This is my own bulletin board

Student enrollment
As an experiment, I can try to teach programming (but NOT reverse engineering and NOT infosec). With a strong geeky bias towards theoretical computer science and discrete mathematics and everything around it. No matter what your age, education and background. Free of charge so far (in demo version mode), in my spare time, remotely.

Please send me your source codes which you are most proud of: <first_name @ last_name . com> or <first_name . last_name @ gmail . com>, so I could measure your level of experience. Please also mention your age, education/work and topics you would like to study.

This book is probably outdated already. (Unless it was just downloaded from https://beginners.re/.)
The book is changing too often, content being added, bugs are (hopefully) being fixed. The latest version is always at https://beginners.re/.
This PDF you currently reading was compiled at April 10, 2020.

If you have printed this book on paper, can you please send me a picture of it, for collection? <first_name @ last_name . com> or <first_name . last_name @ gmail . com>. The collection, so far: https://yurichev.com/news/20200222_printed_RE4B/.

My dear readers! From time to time, I have questions, I don’t know who (or where) to ask. Or I’m just lazy... Can you please help me?

I have a 2TB HDD that is constantly busy/noisy. Even if no OS booted! That drives me crazy. A strange thing – when I turn on “long” SMART test (smartctl --test=long /dev/sda), it is almost silent, but for a 5 hours. When test finish, the HDD is constantly noisy again. While everything is fine in SMART info. The HDD is new. Maybe this is a kind of maintenance I should just wait for the end of? Nothing suspicious in SMART info: https://pastebin.com/5eFePXgG.

Can anyone help me with Low Fragmentation Heap in Windows?

Survey: how do you use DLL injection aside of API interception?

I’ve got a huge graph, for example, $10^6$ nodes (vertices). I want to visualize it somehow, so that a user would navigate it using mouse. Clicked on link (edge) and moved to another node (vertex). Like in IDA. Maybe, using JavaScript. Are there any mature open-source solutions?

Do you remember “The Incredible Machine” videogame for DOS? Do you know about Rube Goldberg machines? What can be used today for simulating them? Some kind of physics engine, maybe?

The ERGO BT-590 bluetooth headphones have a touch control that is too sensitive and easy to hit with clothing. How to force Android ignore messages from headphones about buttons?

If you know something, please help me: <first_name @ last_name . com> or <first_name . last_name @ gmail . com>. 
Understanding Assembly Language
Understanding Assembly Language

(Reverse Engineering for Beginners)

Why two titles? Read here: on page xiii.

Dennis Yurichev
<first_name @ last_name . com>

©2013-2020, Dennis Yurichev.
This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/.
Text version (April 10, 2020).
The latest version (and Russian edition) of this text is accessible at beginners.re.

Call for translators!

You may want to help me with translating this work into languages other than English and Russian. Just send me any piece of translated text (no matter how short) and I’ll put it into my LaTeX source code.
Read here.
The language statistics is available right here: https://beginners.re/.
Speed isn’t important, because this is an open-source project, after all. Your name will be mentioned as a project contributor. Korean, Chinese, and Persian languages are reserved by publishers. English and Russian versions I do by myself, but my English is still that horrible, so I’m very grateful for any notes about grammar, etc. Even my Russian is flawed, so I’m grateful for notes about Russian text as well!
So do not hesitate to contact me: <first_name @ last_name . com> or <first_name . last_name @ gmail . com>.
# Abridged contents

1 Code Patterns  
2 Important fundamentals  
3 Slightly more advanced examples  
4 Java  
5 Finding important/interesting stuff in the code  
6 OS-specific  
7 Tools  
8 Case studies  
9 Examples of reversing proprietary file formats  
10 Dynamic binary instrumentation  
11 Other things  
12 Books/blogs worth reading  
13 Communities  

Afterword  

Appendix  

Acronyms Used  

Glossary  

Index
# Contents

1 Code Patterns ................................. 1
   1.1 The method .................................. 1
   1.2 Some basics .................................. 2
      1.2.1 A short introduction to the CPU ......... 2
      1.2.2 Numeral Systems .......................... 3
      1.2.3 Converting From One Radix To Another .... 3
   1.3 An Empty Function ......................... 5
      1.3.1 x86 ..................................... 5
      1.3.2 ARM .................................... 6
      1.3.3 MIPS .................................... 6
      1.3.4 Empty Functions in Practice .............. 6
   1.4 Returning Values .......................... 7
      1.4.1 x86 ..................................... 7
      1.4.2 ARM .................................... 7
      1.4.3 MIPS .................................... 8
   1.5 Hello, world! ............................... 8
      1.5.1 x86 ..................................... 8
      1.5.2 x86-64 .................................. 14
      1.5.3 ARM .................................... 18
      1.5.4 MIPS .................................... 24
      1.5.5 Conclusion ................................ 28
      1.5.6 Exercises ................................ 28
   1.6 Function prologue and epilogue .......... 29
      1.6.1 Recursion ................................ 29
   1.7 An Empty Function: redux ................. 29
   1.8 Returning Values: redux .................. 29
   1.9 Stack ........................................ 30
      1.9.1 Why does the stack grow backwards? ..... 30
      1.9.2 What is the stack used for? ............... 36
      1.9.3 A typical stack layout .................... 36
      1.9.4 Noise in stack ............................ 41
      1.9.5 Exercises ................................ 41
   1.10 Almost empty function .................... 41
   1.11 printf() with several arguments ........ 42
      1.11.1 x86 ..................................... 42
      1.11.2 ARM .................................... 53
      1.11.3 MIPS .................................... 58
      1.11.4 Conclusion ................................ 64
      1.11.5 By the way ................................ 65
   1.12 scanf() ..................................... 65
      1.12.1 Simple example .......................... 66
      1.12.2 The classic mistake ....................... 75
      1.12.3 Global variables ......................... 76
      1.12.4 scanf() ................................ 85
      1.12.5 Exercise ................................ 96
   1.13 Worth noting: global vs. local variables .. 96
   1.14 Accessing passed arguments .............. 97
      1.14.1 x86 ..................................... 97
      1.14.2 x64 ..................................... 99
      1.14.3 ARM .................................... 102
      1.14.4 MIPS .................................... 105
   1.15 More about results returning ............ 106
      1.15.1 Attempt to use the result of a function returning void ............ 106
1.15.2 What if we do not use the function result? ................................................. 107
1.15.3 Returning a structure ................................................................. 107
1.16 Pointers ................................................................. 109
  1.16.1 Returning values ................................................................. 109
  1.16.2 Swap input values ................................................................. 118
1.17 GOTO operator ................................................................. 119
  1.17.1 Dead code ................................................................. 122
  1.17.2 Exercise ................................................................. 123
1.18 Conditional jumps ................................................................. 123
  1.18.1 Simple example ................................................................. 123
  1.18.2 Calculating absolute value ...................................................... 140
  1.18.3 Ternary conditional operator .................................................... 142
  1.18.4 Getting minimal and maximal values .......................................... 145
  1.18.5 Conclusion ................................................................. 149
  1.18.6 Exercise ................................................................. 151
1.19 Software cracking ................................................................. 151
1.20 Impossible shutdown practical joke (Windows 7) ...................... 152
1.21 switch()/case/default ................................................................. 153
  1.21.1 Small number of cases ...................................................... 153
  1.21.2 A lot of cases ................................................................. 166
  1.21.3 When there are several case statements in one block ................... 178
  1.21.4 Fall-through ................................................................. 182
  1.21.5 Exercises ................................................................. 183
1.22 Loops ................................................................. 183
  1.22.1 Simple example ................................................................. 183
  1.22.2 Memory blocks copying routine .............................................. 194
  1.22.3 Condition check ................................................................. 197
  1.22.4 Conclusion ................................................................. 198
  1.22.5 Exercises ................................................................. 199
1.23 More about strings ................................................................. 200
  1.23.1 strlen() ................................................................. 200
  1.23.2 Boundaries of strings ...................................................... 210
1.24 Replacing arithmetic instructions to other ones ................................................................. 211
  1.24.1 Multiplication ................................................................. 211
  1.24.2 Division ................................................................. 216
  1.24.3 Exercise ................................................................. 217
1.25 Floating-point unit ................................................................. 217
  1.25.1 IEEE 754 ................................................................. 217
  1.25.2 x86 ................................................................. 217
  1.25.3 ARM, MIPS, x86/x64 SIMD .................................................. 217
  1.25.4 C/C++ ................................................................. 217
  1.25.5 Simple example ................................................................. 218
  1.25.6 Passing floating point numbers via arguments ......................... 228
  1.25.7 Comparison example ...................................................... 230
  1.25.8 Some constants ................................................................. 264
  1.25.9 Copying ................................................................. 264
  1.25.10 Stack, calculators and reverse Polish notation ....................... 264
  1.25.11 80 bits? ................................................................. 264
  1.25.12 x64 ................................................................. 264
  1.25.13 Exercises ................................................................. 264
1.26 Arrays ................................................................. 264
  1.26.1 Simple example ................................................................. 264
  1.26.2 Buffer overflow ................................................................. 271
  1.26.3 Buffer overflow protection methods ......................... 279
  1.26.4 One more word about arrays .............................................. 282
  1.26.5 Array of pointers to strings .................................................. 283
  1.26.6 Multidimensional arrays ...................................................... 290
  1.26.7 Pack of strings as a two-dimensional array ......................... 297
  1.26.8 Conclusion ................................................................. 301
  1.26.9 Exercises ................................................................. 301
1.27 Example: a bug in Angband ................................................................. 301
1.28 Manipulating specific bit(s) ................................................................. 303
  1.28.1 Specific bit checking ...................................................... 303
  1.28.2 Setting and clearing specific bits .......................................... 307
1.29 Linear congruential generator .......................... 336
1.29.1 x86 ........................................ 337
1.29.2 x64 ........................................ 338
1.29.3 32-bit ARM .................................. 338
1.29.4 MIPS ........................................ 339
1.29.5 Thread-safe version of the example ................. 341

1.30 Structures ......................................... 341
1.30.1 MSVC: SYSTEMTIME example ...................... 341
1.30.2 Let’s allocate space for a structure using malloc() 344
1.30.3 UNIX: struct tm ................................ 346
1.30.4 Fields packing in structure ......................... 356
1.30.5 Nested structures ................................ 363
1.30.6 Bit fields in a structure .......................... 366
1.30.7 Exercises ....................................... 373

1.31 The classic struct bug ................................ 373

1.32 Unions ............................................. 374
1.32.1 Pseudo-random number generator example ........ 374
1.32.2 Calculating machine epsilon ......................... 377
1.32.3 FSCALE instruction replacement .................... 379
1.32.4 Fast square root calculation ......................... 380

1.33 Pointers to functions ................................ 381
1.33.1 MSVC .......................................... 381
1.33.2 GCC ........................................... 388
1.33.3 Danger of pointers to functions ..................... 392

1.34 64-bit values in 32-bit environment .................... 392
1.34.1 Returning of 64-bit value .......................... 392
1.34.2 Arguments passing, addition, subtraction .......... 393
1.34.3 Multiplication, division ........................... 396
1.34.4 Shifting right .................................... 400
1.34.5 Converting 32-bit value into 64-bit one ............. 401

1.35 LARGE_INTEGER structure case ....................... 402

1.36 SIMD .............................................. 404
1.36.1 Vectorization ..................................... 405
1.36.2 SIMD strlen() implementation ...................... 414

1.37 64 bits .............................................. 418
1.37.1 x86-64 .......................................... 418
1.37.2 ARM ............................................ 424
1.37.3 Float point numbers ............................... 424
1.37.4 64-bit architecture criticism ......................... 425

1.38 Working with floating point numbers using SIMD .... 425
1.38.1 Simple example ................................... 425
1.38.2 Passing floating point number via arguments ..... 433
1.38.3 Comparison example ................................. 434
1.38.4 Calculating machine epsilon: x64 and SIMD ......... 435
1.38.5 Pseudo-random number generator example revisited 436
1.38.6 Summary .......................................... 436

1.39 ARM-specific details .................................. 437
1.39.1 Number sign (#) before number ...................... 437
1.39.2 Addressing modes ................................ 437
1.39.3 Loading a constant into a register ................... 438
1.39.4 Relocs in ARM64 .................................. 439

1.40 MIPS-specific details ................................ 441
1.40.1 Loading a 32-bit constant into register .............. 441
1.40.2 Further reading about MIPS ........................ 442

2 Important fundamentals .................................. 443
2.1 Integral datatypes ..................................... 443
2.1.1 Bit ............................................. 443

1 Floating-Point Unit
2.1.2 Nibble AKA nybble
2.1.3 Byte
2.1.4 Wide char
2.1.5 Signed integer vs unsigned
2.1.6 Word
2.1.7 Address register
2.1.8 Numbers
2.2 Signed number representations
2.2.1 Using IMUL over MUL
2.2.2 Couple of additions about two's complement form
2.2.3 -1
2.3 Integer overflow
2.4 AND
2.4.1 Checking if a value is on 2ⁿ boundary
2.4.2 KOI-8R Cyrillic encoding
2.5 AND and OR as subtraction and addition
2.5.1 ZX Spectrum ROM text strings
2.6 XOR (exclusive OR)
2.6.1 Logical difference
2.6.2 Everyday speech
2.6.3 Encryption
2.6.4 RAID³⁴
2.6.5 XOR swap algorithm
2.6.6 XOR linked list
2.6.7 Switching value trick
2.6.8 Zobrist hashing / tabulation hashing
2.6.9 By the way
2.6.10 AND/OR/XOR as MOV
2.7 Population count
2.8 Endianness
2.8.1 Big-endian
2.8.2 Little-endian
2.8.3 Example
2.8.4 Bi-endian
2.8.5 Converting data
2.9 Memory
2.10 CPU
2.10.1 Branch predictors
2.10.2 Data dependencies
2.11 Hash functions
2.11.1 How do one-way functions work?
3 Slightly more advanced examples
3.1 Zero register
3.2 Double negation
3.3 const correctness
3.3.1 Overlapping const strings
3.4 strstr() example
3.5 qsort() revisited
3.6 Temperature converting
3.6.1 Integer values
3.6.2 Floating-point values
3.7 Fibonacci numbers
3.7.1 Example #1
3.7.2 Example #2
3.7.3 Summary
3.8 CRC32 calculation example
3.9 Network address calculation example
3.9.1 calc_network_address()
3.9.2 form_IP()
3.9.3 print_as_IP()
3.9.4 form_netmask() and set_bit()
3.9.5 Summary

²Redundant Array of Independent Disks
5.2 Communication with outer world (function level) ........................................... 695
5.3 Communication with the outer world (win32) ................................................. 695
  5.3.1 Often used functions in the Windows API ................................................. 696
  5.3.2 Extending trial period ................................................................. 696
  5.3.3 Removing nag dialog box ................................................................. 696
  5.3.4 tracer: Intercepting all functions in specific module ................................. 696
5.4 Strings ................................................................................................. 697
  5.4.1 Text strings ......................................................................................... 697
  5.4.2 Finding strings in binary ........................................................................ 701
  5.4.3 Error/debug messages ........................................................................... 702
  5.4.4 Suspicious magic strings ........................................................................ 703
5.5 Calls to assert() ......................................................................................... 703
5.6 Constants ................................................................................................. 704
  5.6.1 Magic numbers ......................................................................................... 704
  5.6.2 Specific constants .................................................................................... 706
  5.6.3 Searching for constants ........................................................................... 706
5.7 Finding the right instructions ........................................................................ 706
5.8 Suspicious code patterns .............................................................................. 707
  5.8.1 XOR instructions ...................................................................................... 707
  5.8.2 Hand-written assembly code .................................................................... 708
5.9 Using magic numbers while tracing .............................................................. 709
5.10 Loops ........................................................................................................ 709
  5.10.1 Some binary file patterns ........................................................................ 710
  5.10.2 Memory “snapshots” comparing .............................................................. 717
5.11 ISA\(^5\) detection ....................................................................................... 719
  5.11.1 Incorrectly disassembled code ............................................................... 719
  5.11.2 Correctly disassembled code .................................................................. 724
5.12 Other things ............................................................................................... 724
  5.12.1 General idea .......................................................................................... 724
  5.12.2 Order of functions in binary code .......................................................... 724
  5.12.3 Tiny functions ......................................................................................... 724
  5.12.4 C++ ........................................................................................................ 724
  5.12.5 Crash on purpose ................................................................................... 724

6 OS-specific .................................................................................................... 725
  6.1 Arguments passing methods (calling conventions) ........................................... 725
    6.1.1 cdecl ...................................................................................................... 725
    6.1.2 stdcall .................................................................................................. 725
    6.1.3 fastcall .................................................................................................. 726
    6.1.4 thiscall .................................................................................................. 727
    6.1.5 x86-64 .................................................................................................. 728
    6.1.6 Return values of float and double type .................................................. 730
    6.1.7 Modifying arguments ............................................................................. 730
    6.1.8 Taking a pointer to function argument .................................................. 731
    6.1.9 Python ctypes problem (x86 assembly homework) ................................ 733
    6.1.10 Cdecl example: a DLL ......................................................................... 733
  6.2 Thread Local Storage .................................................................................... 734
    6.2.1 Linear congruential generator revisited .................................................. 734
  6.3 System calls (syscall-s) ................................................................................ 738
    6.3.1 Linux ...................................................................................................... 739
    6.3.2 Windows ................................................................................................. 739
  6.4 Linux ............................................................................................................ 739
    6.4.1 Position-independent code ..................................................................... 739
    6.4.2 LD_PRELOAD hack in Linux .................................................................. 742
  6.5 Windows NT ............................................................................................... 744
    6.5.1 CRT (win32) .......................................................................................... 744
    6.5.2 Win32 PE ............................................................................................... 748
    6.5.3 Windows SEH ........................................................................................ 755
    6.5.4 Windows NT: Critical section ............................................................... 777

7 Tools ............................................................................................................. 780
  7.1 Binary analysis ............................................................................................ 780
    7.1.1 Disassemblers ......................................................................................... 780

\(^5\)Instruction Set Architecture
12.1 Books and other materials .................................................. 998
  12.1.1 Reverse Engineering .................................................. 998
  12.1.2 Windows .................................................................... 998
  12.1.3 C/C++ ......................................................................... 998
  12.1.4 x86 / x86-64 ................................................................. 999
  12.1.5 ARM ......................................................................... 999
  12.1.6 Assembly language ....................................................... 999
  12.1.7 Java ......................................................................... 999
  12.1.8 UNIX ......................................................................... 999
  12.1.9 Programming in general ................................................. 999
  12.1.10 Cryptography ............................................................... 1000

13 Communities .......................................................................... 1001

Afterword .................................................................................. 1003
  13.1 Questions? ....................................................................... 1003

Appendix ...................................................................................... 1005
  .1 x86 .................................................................................. 1005
     .1.1 Terminology ................................................................. 1005
     .1.2 General purpose registers ............................................. 1005
     .1.3 FPU registers ............................................................... 1009
     .1.4 SIMD registers ........................................................... 1011
     .1.5 Debugging registers ..................................................... 1011
     .1.6 Instructions ................................................................. 1012
     .1.7 npad ........................................................................ 1024
  .2 ARM .................................................................................. 1025
     .2.1 Terminology ................................................................. 1025
     .2.2 Versions .................................................................... 1026
     .2.3 32-bit ARM (AArch32) .................................................. 1026
     .2.4 64-bit ARM (AArch64) .................................................. 1027
     .2.5 Instructions ................................................................. 1027
  .3 MIPS .................................................................................. 1028
     .3.1 Registers ....................................................................... 1028
     .3.2 Instructions ................................................................. 1029
  .4 Some GCC library functions .................................................. 1029
  .5 Some MSVC library functions ............................................... 1029
  .6 Cheatsheets ....................................................................... 1029
     .6.1 IDA ........................................................................ 1029
     .6.2 OllyDbg ...................................................................... 1030
     .6.3 MSVC .................................................................... 1030
     .6.4 GCC ....................................................................... 1030
     .6.5 GDB ....................................................................... 1031

Acronyms Used ......................................................................... 1033

Glossary ....................................................................................... 1038

Index .......................................................................................... 1040
Preface

What is with two titles?
The book was named “Reverse Engineering for Beginners” in 2014-2018, but I always suspected this makes readership too narrow.

Infosec people know about “reverse engineering”, but I’ve rarely hear the “assembler” word from them. Likewise, the “reverse engineering” term is somewhat cryptic to a general audience of programmers, but they know about “assembler”.

In July 2018, as an experiment, I’ve changed the title to “Assembly Language for Beginners” and posted the link to Hacker News website, and the book was received generally well.

So let it be, the book now has two titles.

However, I’ve changed the second title to “Understanding Assembly Language”, because someone had already written “Assembly Language for Beginners” book. Also, people say “for Beginners” sounds a bit sarcastic for a book of ~1000 pages.

The two books differ only by title, filename (UAL-XX.pdf versus RE4B-XX.pdf), URL and a couple of the first pages.

About reverse engineering

There are several popular meanings of the term “reverse engineering”:
1) The reverse engineering of software; researching compiled programs
2) The scanning of 3D structures and the subsequent digital manipulation required in order to duplicate them
3) Recreating DBMS\textsuperscript{8} structure

This book is about the first meaning.

Prerequisites

Basic knowledge of the C PL\textsuperscript{9}. Recommended reading: 12.1.3 on page 998.

Exercises and tasks

...can be found at: http://challenges.re.

Praise for this book

https://beginners.re/#praise.

Universities

The book is recommended at least at these universities: https://beginners.re/#uni.

Thanks

For patiently answering all my questions: Slava “Avid” Kazakov, SkullC0DEr.


\textsuperscript{7}https://news.ycombinator.com/item?id=17549050
\textsuperscript{8}Database Management Systems
\textsuperscript{9}Programming Language
\textsuperscript{10}goto-vlad@github
\textsuperscript{11}https://github.com/pixjuan
\textsuperscript{12}https://github.com/73696e65
For helping me in other ways: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar, Peter Sovietov, Misha “tiphareth” Verbitsky.

For translating the book into Simplified Chinese: Antiy Labs (antiy.cn), Archer.
For translating the book into Korean: Byungho Min.
For translating the book into Dutch: Cedric Sambre (AKA Midas).
For translating the book into Spanish: Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida, Diogo Mussi, Patricio Galdames.
For translating the book into Portuguese: Thales Stevan de A. Gois, Diogo Mussi, Luiz Filipe.
For translating the book into French: Florent Besnard, Marc Remy, Baudouin Landais, Téo Dacquet, BlueSkeye@GitHub.
For translating the book into German: Dennis Siekmeier, Julius Angres, Dirk Loser, Clemens Tamme.
For translating the book into Polish: Kateryna Rozanova, Aleksander Miestewicz, Wiktoria Lewicka.
For translating the book into Japanese: shmz@github.
Vasil Kolev did a great amount of work in proofreading and correcting many mistakes.
Thanks also to all the folks on github.com who have contributed notes and corrections.
Many \TeX packages were used: I would like to thank the authors as well.

Donors

Those who supported me during the time when I wrote significant part of the book:

2 * Oleg Vygovsky (50+100 UAH), Daniel Bilar (550), James Truscott ($4.5), Luis Rocha ($63), Joris van de Vis ($127), Richard S Shultz ($20), Jang Minchang ($20), Shade Atlas (5 AUD), Yao Xiao ($10), Pawel Szczur (40 CHF), Justin Simms ($20), Shawn the Rock ($27), Ki Chan Ahn ($50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gisason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao ($50), Philippe Teuwen ($4), Martin Haebeler ($10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai (€15), Bayna Al Zaabi ($75), Redfive B.V. (€25), Joona Oskari Heikkilä (€5), Marshall Bishop (500), Nicolas Werner ($12), Jeremy Brown ($100), Alexandre Borges ($25), Vladimir Dikovski ($50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent ($30), Sri Harsha Kandrakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe ($14), Maxim Dyakonov (€3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov ($100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei ($40), Z0vsky (€10), Yu Dai ($10), Anonymous (€15), Vladislav Chelnokov ($25), Nenad Noveljic ($50), Ryan Smith ($25), Andreas Schommer (€5), Nikolay Gavrilov ($300).

Thanks a lot to every donor!

13https://github.com/TheRenaissance
14https://github.com/pinkrab
15https://github.com/paolostivanin
16https://github.com/Internaut401
17https://github.com/besnardf
18https://github.com/mremy
19https://github.com/T30rix
20https://github.com/BlueSkeye
21https://github.com/JAngres
22https://github.com/PolymathMonkey
23https://github.com/shmz
24https://vasil.ludost.net/
mini-FAQ

Q: Is this book simpler/easier than others?
A: No, it is at about the same level as other books of this subject.

Q: I’m too frightened to start reading this book, there are more than 1000 pages. “…for Beginners” in the name sounds a bit sarcastic.
A: All sorts of listings are the bulk of the book. The book is indeed for beginners, there is a lot missing (yet).

Q: What are the prerequisites for reading this book?
A: A basic understanding of C/C++ is desirable.

Q: Should I really learn x86/x64/ARM and MIPS at once? Isn’t it too much?
A: Starters can read about just x86/x64, while skipping or skimming the ARM and MIPS parts.

Q: Can I buy a Russian or English hard copy/paper book?

Q: Is there an epub or mobi version?
A: No. The book is highly dependent on TeX/LaTeX-specific hacks, so converting to HTML (epub/mobi are a set of HTMLs) would not be easy.

Q: Why should one learn assembly language these days?
A: Unless you are an OS developer, you probably don’t need to code in assembly—the latest compilers (2010s) are much better at performing optimizations than humans. Also, the latest CPUs are very complex devices, and assembly knowledge doesn’t really help towards understand their internals.

That being said, there are at least two areas where a good understanding of assembly can be helpful: First and foremost, for security/malware research. It is also a good way to gain a better understanding of your compiled code while debugging. This book is therefore intended for those who want to understand assembly language rather than to code in it, which is why there are many examples of compiler output contained within.

Q: I clicked on a hyperlink inside a PDF-document, how do I go back?
A: In Adobe Acrobat Reader click Alt+LeftArrow. In Evince click “<” button.

Q: May I print this book / use it for teaching?
A: Of course! That’s why the book is licensed under the Creative Commons license (CC BY-SA 4.0).

Q: Why is this book free? You’ve done great job. This is suspicious, as with many other free things.
A: In my own experience, authors of technical literature write mostly for self-advertisement purposes. It’s not possible to make any decent money from such work.

Q: How does one get a job in reverse engineering?
A: There are hiring threads that appear from time to time on reddit, devoted to RE. Try looking there.

A somewhat related hiring thread can be found in the “netsec” subreddit.

Q: Compilers’ versions in the book are outdated already...
A: No need to follow all steps precisely. Use the compilers you already have installed on your OS. Also, there is: Compiler Explorer.

Q: I have a question...
A: Send it to me by email (<first_name @ last_name . com> or <first_name . last_name @ gmail . com>).

---

26Operating System
27A very good text on this topic: [Agner Fog, The microarchitecture of Intel, AMD and VIA CPUs, (2016)]
28Central Processing Unit
29reddit.com/r/ReverseEngineering/
About the Korean translation

In January 2015, the Acorn publishing company (www.acornpub.co.kr) in South Korea did a huge amount of work in translating and publishing this book (as it was in August 2014) into Korean.

It’s available now at their website.

The translator is Byungho Min (twitter/tais9). The cover art was done by the artistic Andy Nechaevsky, a friend of the author: facebook/andydinka. Acorn also holds the copyright to the Korean translation.

So, if you want to have a real book on your shelf in Korean and want to support this work, it is now available for purchase.

About the Persian/Farsi translation

In 2016 the book was translated by Mohsen Mostafa Jokar (who is also known to Iranian community for his translation of Radare manual[^30]). It is available on the publisher’s website[^31] (Pendare Pars).

Here is a link to a 40-page excerpt: https://beginners.re/farsi.pdf.


About the Chinese translation

In April 2017, translation to Chinese was completed by Chinese PTPress. They are also the Chinese translation copyright holders.


The principal translator is Archer, to whom the author owes very much. He was extremely meticulous (in a good sense) and reported most of the known mistakes and bugs, which is very important in literature such as this book. The author would recommend his services to any other author!

The guys from Antiy Labs has also helped with translation. Here is preface written by them.

[^31]: http://goo.gl/2Tzx0H
Chapter 1

Code Patterns

1.1 The method

When the author of this book first started learning C and, later, C++, he used to write small pieces of code, compile them, and then look at the assembly language output. This made it very easy for him to understand what was going on in the code that he had written. He did this so many times that the relationship between the C/C++ code and what the compiler produced was imprinted deeply in his mind. It’s now easy for him to imagine instantly a rough outline of a C code’s appearance and function. Perhaps this technique could be helpful for others.

By the way, there is a great website where you can do the same, with various compilers, instead of installing them on your box. You can use it as well: https://godbolt.org/.

Exercises

When the author of this book studied assembly language, he also often compiled small C functions and then rewrote them gradually to assembly, trying to make their code as short as possible. This probably is not worth doing in real-world scenarios today, because it’s hard to compete with the latest compilers in terms of efficiency. It is, however, a very good way to gain a better understanding of assembly. Feel free, therefore, to take any assembly code from this book and try to make it shorter. However, don’t forget to test what you have written.

Optimization levels and debug information

Source code can be compiled by different compilers with various optimization levels. A typical compiler has about three such levels, where level zero means that optimization is completely disabled. Optimization can also be targeted towards code size or code speed. A non-optimizing compiler is faster and produces more understandable (albeit verbose) code, whereas an optimizing compiler is slower and tries to produce code that runs faster (but is not necessarily more compact). In addition to optimization levels, a compiler can include some debug information in the resulting file, producing code that is easy to debug. One of the important features of the ‘debug’ code is that it might contain links between each line of the source code and its respective machine code address. Optimizing compilers, on the other hand, tend to produce output where entire lines of source code can be optimized away and thus not even be present in the resulting machine code. Reverse engineers can encounter either version, simply because some developers turn on the compiler’s optimization flags and others do not. Because of this, we’ll try to work on examples of both debug and release versions of the code featured in this book, wherever possible.

Sometimes some pretty ancient compilers are used in this book, in order to get the shortest (or simplest) possible code snippet.

1 In fact, he still does this when he can’t understand what a particular bit of code does. A recent example from the year 2019: p += p+(i&1)+2; from the “SAT0W” SAT-solver by D.Knuth.
1.2 Some basics

1.2.1 A short introduction to the CPU

The CPU is the device that executes the machine code a program consists of.

A short glossary:

**Instruction**: A primitive CPU command. The simplest examples include: moving data between registers, working with memory, primitive arithmetic operations. As a rule, each CPU has its own instruction set architecture (ISA).

**Machine code**: Code that the CPU directly processes. Each instruction is usually encoded by several bytes.

**Assembly language**: Mnemonic code and some extensions, like macros, that are intended to make a programmer’s life easier.

**CPU register**: Each CPU has a fixed set of general purpose registers (GPR\(^2\)). \(\approx 8\) in x86, \(\approx 16\) in x86-64, and also \(\approx 16\) in ARM. The easiest way to understand a register is to think of it as an untyped temporary variable. Imagine if you were working with a high-level PL and could only use eight 32-bit (or 64-bit) variables. Yet a lot can be done using just these!

One might wonder why there needs to be a difference between machine code and a PL. The answer lies in the fact that humans and CPUs are not alike—it is much easier for humans to use a high-level PL like C/C++, Java, or Python, but it is easier for a CPU to use a much lower level of abstraction. Perhaps it would be possible to invent a CPU that can execute high-level PL code, but it would be many times more complex than the CPUs we know of today. In a similar fashion, it is very inconvenient for humans to write in assembly language, due to it being so low-level and difficult to write in without making a huge number of annoying mistakes. The program that converts the high-level PL code into assembly is called a compiler.\(^3\)

A couple of words about different ISAs

The x86 ISA has always had variable-length instructions, so when the 64-bit era came, the x64 extensions did not impact the ISA very significantly. In fact, the x86 ISA still contains a lot of instructions that first appeared in 16-bit 8086 CPU, yet are still found in the CPUs of today. ARM is a RISC\(^4\) CPU designed with constant-length instructions in mind, which had some advantages in the past. In the very beginning, all ARM instructions were encoded in 4 bytes\(^5\). This is now referred to as “ARM mode”. Then they realized it wasn’t as frugal as they first imagined. In fact, the most common CPU instructions\(^6\) in real world applications can be encoded using less information. They therefore added another ISA, called Thumb, in which each instruction was encoded in just 2 bytes. This is now referred to as “Thumb mode”. However, not all ARM instructions can be encoded in just 2 bytes, so the Thumb instruction set is somewhat limited. It is worth noting that code compiled for ARM mode and Thumb mode can coexist within one single program.

The ARM creators thought Thumb could be extended, giving rise to Thumb-2, which appeared in ARMv7. Thumb-2 still uses 2-byte instructions, but has some new instructions which have the size of 4 bytes. There is a common misconception that Thumb-2 is a mix of ARM and Thumb. This is incorrect. Rather, Thumb-2 was extended to fully support all processor features so it could compete with ARM mode—a goal that was clearly achieved, as the majority of applications for iPod/iPhone/iPad are compiled for the Thumb-2 instruction set. (Though, admittedly, this is largely due to the fact that Xcode does this by default). Later the 64-bit ARM came out. This ISA has 4-byte instructions, and lacked the need of any additional Thumb mode. However, the 64-bit requirements affected the ISA, resulting in us now having three ARM instruction sets: ARM mode, Thumb mode (including Thumb-2) and ARM64. These ISAs intersect partially, but it can be said that they are different ISAs, rather than variations of the same one. Therefore, we will try to add fragments of code in all three ARM ISAs in this book. There are, by the way, many other RISC ISAs with fixed length 32-bit instructions, such as MIPS, PowerPC and Alpha AXP.

\(^2\)General Purpose Registers

\(^3\)Old-school Russian literature also uses the term “translator”.

\(^4\)Reduced Instruction Set Computing

\(^5\)Fixed-length instructions are handy because one can calculate the next (or previous) instruction address without effort. This feature will be discussed in the switch() operator (1.21.2 on page 173) section.

\(^6\)e.g. MOV/PUSH/CALL/jcc

---

---
1.2.2 Numeral Systems

Nowadays octal numbers seem to be used for exactly one purpose—file permissions on POSIX systems—but hexadecimal numbers are widely used to emphasize the bit pattern of a number over its numeric value.

Alan A. A. Donovan, Brian W. Kernighan — The Go Programming Language

Humans have become accustomed to a decimal numeral system, probably because almost everyone has 10 fingers. Nevertheless, the number “10” has no significant meaning in science and mathematics. The natural numeral system in digital electronics is binary: 0 is for an absence of current in the wire, and 1 for presence. 10 in binary is 2 in decimal, 100 in binary is 4 in decimal, and so on.

If the numeral system has 10 digits, it has a radix (or base) of 10. The binary numeral system has a radix of 2.

Important things to recall:

1) A number is a number, while a digit is a term from writing systems, and is usually one character
2) The value of a number does not change when converted to another radix; only the writing notation for that value has changed (and therefore the way of representing it in RAM\(^7\)).

1.2.3 Converting From One Radix To Another

Positional notation is used in almost every numerical system. This means that a digit has weight relative to where it is placed inside of the larger number. If 2 is placed at the rightmost place, it’s 2, but if it’s placed one digit before rightmost, it’s 20.

What does 1234 stand for?

\[10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4 = 1234\]  or  \[1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234\]

It’s the same story for binary numbers, but the base is 2 instead of 10. What does 0b101011 stand for?

\[2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43\]  or  \[32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43\]

There is such a thing as non-positional notation, such as the Roman numeral system. 8. Perhaps, humankind switched to positional notation because it’s easier to do basic operations (addition, multiplication, etc.) on paper by hand.

Binary numbers can be added, subtracted and so on in the very same as taught in schools, but only 2 digits are available.

Binary numbers are bulky when represented in source code and dumps, so that is where the hexadecimal numeral system can be useful. A hexadecimal radix uses the digits 0..9, and also 6 Latin characters: A..F. Each hexadecimal digit takes 4 bits or 4 binary digits, so it’s very easy to convert from binary number to hexadecimal and back, even manually, in one’s mind.

<table>
<thead>
<tr>
<th>hexadecimal</th>
<th>binary</th>
<th>decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
<td>14</td>
</tr>
</tbody>
</table>

\(^7\)Random-Access Memory

How can one tell which radix is being used in a specific instance?

Decimal numbers are usually written as is, i.e., 1234. Some assemblers allow an identifier on decimal radix numbers, in which the number would be written with a "d" suffix: 1234d.

Binary numbers are sometimes prepended with the “0b” prefix: 0b100110111 (GCC\(^9\) has a non-standard language extension for this\(^10\)). There is also another way: using a "b" suffix, for example: 100110111b. This book tries to use the “0b” prefix consistently throughout the book for binary numbers.

Hexadecimal numbers are prepended with “0x” prefix in C/C++ and other PLs: 0x1234ABCD. Alternatively, they are given a “h” suffix: 1234ABCDh. This is common way of representing them in assemblers and debuggers. In this convention, if the number is started with a Latin (A..F) digit, a 0 is added at the beginning: 0ABCDEFh. There was also convention that was popular in 8-bit home computers era, using $ prefix, like $ABCD. The book will try to stick to “0x” prefix throughout the book for hexadecimal numbers.

Should one learn to convert numbers mentally? A table of 1-digit hexadecimal numbers can easily be memorized. As for larger numbers, it’s probably not worth tormenting yourself. Perhaps the most visible hexadecimal numbers are in URL\(^11\)s. This is the way that non-Latin characters are encoded. For example: https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9 is the URL of Wiktionary article about “naïveté” word.

### Octal Radix

Another numeral system heavily used in the past of computer programming is octal. In octal there are 8 digits (0..7), and each is mapped to 3 bits, so it’s easy to convert numbers back and forth. It has been superseded by the hexadecimal system almost everywhere, but, surprisingly, there is a *NIX utility, used often by many people, which takes octal numbers as argument: chmod.

As many *NIX users know, chmod argument can be a number of 3 digits. The first digit represents the rights of the owner of the file (read, write and/or execute), the second is the rights for the group to which the file belongs, and the third is for everyone else. Each digit that chmod takes can be represented in binary form:

<table>
<thead>
<tr>
<th>decimal</th>
<th>binary</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>111</td>
<td>rwx</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>rw-</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>r-x</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>r--</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>-wx</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>-w-</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>--x</td>
</tr>
<tr>
<td>0</td>
<td>000</td>
<td>---</td>
</tr>
</tbody>
</table>

So each bit is mapped to a flag: read/write/execute.

The importance of chmod here is that the whole number in argument can be represented as octal number. Let’s take, for example, 644. When you run chmod 644 file, you set read/write permissions for owner, read permissions for group and again, read permissions for everyone else. If we convert the octal number 644 to binary, it would be 110100100, or, in groups of 3 bits, 110 100 100.

Now we see that each triplet describe permissions for owner/group/others: first is rw-, second is r-- and third is r--.

The octal numeral system was also popular on old computers like PDP-8, because word there could be 12, 24 or 36 bits, and these numbers are all divisible by 3, so the octal system was natural in that environment. Nowadays, all popular computers employ word/address sizes of 16, 32 or 64 bits, and these numbers are all divisible by 4, so the hexadecimal system is more natural there.

\(^9\)GNU Compiler Collection
\(^10\)https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html
\(^11\)Uniform Resource Locator
The octal numeral system is supported by all standard C/C++ compilers. This is a source of confusion sometimes, because octal numbers are encoded with a zero prepended, for example, 0377 is 255. Sometimes, you might make a typo and write “09" instead of 9, and the compiler would report an error. GCC might report something like this:
error: invalid digit "9" in octal constant.

Also, the octal system is somewhat popular in Java. When the IDA shows Java strings with non-printable characters, they are encoded in the octal system instead of hexadecimal. The JAD Java decompiler behaves the same way.

**Divisibility**

When you see a decimal number like 120, you can quickly deduce that it’s divisible by 10, because the last digit is zero. In the same way, 123400 is divisible by 100, because the two last digits are zeros.

Likewise, the hexadecimal number 0x1230 is divisible by 0x10 (or 16), 0x123000 is divisible by 0x1000 (or 4096), etc.

The binary number 0b1000101000 is divisible by 0b1000 (8), etc.

This property can often be used to quickly realize if an address or a size of some block in memory is padded to some boundary. For example, sections in PE files are almost always started at addresses ending with 3 hexadecimal zeros: 0x41000, 0x10001000, etc. The reason behind this is the fact that almost all PE sections are padded to a boundary of 0x1000 (4096) bytes.

**Multi-Precision Arithmetic and Radix**

Multi-precision arithmetic can use huge numbers, and each one may be stored in several bytes. For example, RSA keys, both public and private, span up to 4096 bits, and maybe even more.

In [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] we find the following idea: when you store a multi-precision number in several bytes, the whole number can be represented as having a radix of $2^8 = 256$, and each digit goes to the corresponding byte. Likewise, if you store a multi-precision number in several 32-bit integer values, each digit goes to each 32-bit slot, and you may think about this number as stored in radix of $2^{32}$.

**How to Pronounce Non-Decimal Numbers**

Numbers in a non-decimal base are usually pronounced by digit by digit: “one-zero-zero-one-one-...”.

Words like “ten” and “thousand” are usually not pronounced, to prevent confusion with the decimal base system.

**Floating point numbers**

To distinguish floating point numbers from integers, they are usually written with “.0” at the end, like 0.0, 123.0, etc.

### 1.3 An Empty Function

The simplest possible function is arguably one that does nothing:

```c
void f()
{
    return;
};
```

Let’s compile it!

---

1^12Portable Executable
1.3.1 x86

Here’s what both the GCC and MSVC compilers produce on the x86 platform:

Listing 1.2: Optimizing GCC/MSVC (assembly output)

f:
  ret

There is just one instruction: RET, which returns execution to the caller.

1.3.2 ARM

Listing 1.3: Optimizing Keil 6/2013 (ARM mode) assembly output

f
  PROC
  BX   lr
  ENDP

The return address is not saved on the local stack in the ARM ISA, but rather in the link register, so the BX LR instruction causes execution to jump to that address—effectively returning execution to the caller.

1.3.3 MIPS

There are two naming conventions used in the world of MIPS when naming registers: by number (from $0 to $31) or by pseudo name ($V0, $A0, etc.).

The GCC assembly output below lists registers by number:

Listing 1.4: Optimizing GCC 4.4.5 (assembly output)

j
  $31
  nop

...while IDA\textsuperscript{13} does it by pseudo name:

Listing 1.5: Optimizing GCC 4.4.5 (IDA)

j
  $ra
  nop

The first instruction is the jump instruction (J or JR) which returns the execution flow to the caller, jumping to the address in the $31 (or $RA) register.

This is the register analogous to LR\textsuperscript{14} in ARM.

The second instruction is NOP\textsuperscript{15}, which does nothing. We can ignore it for now.

A Note About MIPS Instructions and Register Names

Register and instruction names in the world of MIPS are traditionally written in lowercase. However, for the sake of consistency, this book will stick to using uppercase letters, as it is the convention followed by all the other ISAs featured in this book.

1.3.4 Empty Functions in Practice

Despite the fact empty functions seem useless, they are quite frequent in low-level code.

First of all, they are quite popular in debugging functions, like this one:

Listing 1.6: C/C++ code

void dbg_print (const char *fmt, ...)
{
  #ifdef _DEBUG
  // open log file

\textsuperscript{13} Interactive Disassembler and Debugger developed by Hex-Rays
\textsuperscript{14} Link Register
\textsuperscript{15} No Operation
In a non-debug build (as in a “release”), _DEBUG is not defined, so the dbg_print() function, despite still being called during execution, will be empty.

Similarly, a popular method of software protection is to make one build for legal customers, and another demo build. A demo build can lack some important functions, as with this example:

```c
void save_file()
{
    // a real saving code
    ...
}
```

The save_file() function can be called when the user clicks File->Save on the menu. The demo version may be delivered with this menu item disabled, but even if a software cracker will enable it, only an empty function with no useful code will be called.

IDA marks such functions with names like nullsub_00, nullsub_01, etc.

## 1.4 Returning Values

Another simple function is the one that simply returns a constant value:

```c
int f()
{
    return 123;
}
```

Let's compile it.

### 1.4.1 x86

Here’s what both the GCC and MSVC compilers produce (with optimization) on the x86 platform:

```assembly
f:
    mov    eax, 123
    ret
```

There are just two instructions: the first places the value 123 into the EAX register, which is used by convention for storing the return value, and the second one is RET, which returns execution to the caller.

The caller will take the result from the EAX register.

### 1.4.2 ARM

There are a few differences on the ARM platform:
ARM uses the register R0 for returning the results of functions, so 123 is copied into R0.

It is worth noting that MOV is a misleading name for the instruction in both the x86 and ARM ISAs. The data is not in fact moved, but copied.

### 1.4.3 MIPS

The GCC assembly output below lists registers by number:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>li</td>
<td>$2, 123</td>
<td></td>
</tr>
</tbody>
</table>

...while IDA does it by their pseudo names:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>jr</td>
<td>$ra</td>
<td></td>
</tr>
<tr>
<td>li</td>
<td>$v0, 0x7B</td>
<td></td>
</tr>
</tbody>
</table>

The $2 (or $V0) register is used to store the function’s return value. LI stands for “Load Immediate” and is the MIPS equivalent to MOV.

The other instruction is the jump instruction (J or JR) which returns the execution flow to the caller.

You might be wondering why the positions of the load instruction (LI) and the jump instruction (J or JR) are swapped. This is due to a RISC feature called “branch delay slot”.

The reason this happens is a quirk in the architecture of some RISC ISAs and isn’t important for our purposes—we must simply keep in mind that in MIPS, the instruction following a jump or branch instruction is executed before the jump/branch instruction itself.

As a consequence, branch instructions always swap places with the instruction executed immediately beforehand.

In practice, functions which merely return 1 (true) or 0 (false) are very frequent.

The smallest ever of the standard UNIX utilities, /bin/true and /bin/false return 0 and 1 respectively, as an exit code. (Zero as an exit code usually means success, non-zero means error.)

### 1.5 Hello, world!

Let’s use the famous example from the book [Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, 2ed, (1988)]:

```c
#include <stdio.h>

int main()
{
    printf("hello, world\n");
    return 0;
}
```

### 1.5.1 x86

**MSVC**

Let’s compile it in MSVC 2010:
The /Fa option instructs the compiler to generate an assembly listing file.

Listing 1.14: MSVC 2010

```
CONST SEGMENT
$SG3830 DB 'hello, world', 0AH, 00H
CONST ENDS
PUBLIC _main
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_main PROC
push ebp
mov ebp, esp
push OFFSET $SG3830
call _printf
add esp, 4
xor eax, eax
pop ebp
ret 0
_main ENDP
_TEXT ENDS
```

MSVC produces assembly listings in Intel-syntax. The differences between Intel-syntax and AT&T-syntax will be discussed in 1.5.1 on page 11.

The compiler generated the file, 1.obj, which is to be linked into 1.exe. In our case, the file contains two segments: CONST (for data constants) and _TEXT (for code).

The string hello, world in C/C++ has type const char[] [Bjarne Stroustrup, The C++ Programming Language, 4th Edition, (2013)p176, 7.3.2], but it does not have its own name. The compiler needs to deal with the string somehow, so it defines the internal name $SG3830 for it.

That is why the example may be rewritten as follows:

```
#include <stdio.h>

const char $SG3830[]="hello, world\n";

int main()
{
    printf($SG3830);
    return 0;
}
```

Let’s go back to the assembly listing. As we can see, the string is terminated by a zero byte, which is standard for C/C++ strings. More about C/C++ strings: 5.4.1 on page 697.

In the code segment, _TEXT, there is only one function so far: main(). The function main() starts with prologue code and ends with epilogue code (like almost any function) \(^{16}\).

After the function prologue we see the call to the printf() function: CALL _printf. Before the call, a string address (or a pointer to it) containing our greeting is placed on the stack with the help of the PUSH instruction.

When the printf() function returns the control to the main() function, the string address (or a pointer to it) is still on the stack. Since we do not need it anymore, the stack pointer (the ESP register) needs to be corrected.

ADD ESP, 4 means add 4 to the ESP register value.

Why 4? Since this is a 32-bit program, we need exactly 4 bytes for address passing through the stack. If it was x64 code we would need 8 bytes. ADD ESP, 4 is effectively equivalent to POP register but without using any register\(^{17}\).

For the same purpose, some compilers (like the Intel C++ Compiler) may emit POP ECX instead of ADD (e.g., such a pattern can be observed in the Oracle RDBMS code as it is compiled with the Intel C++ Compiler). You can read more about it in the section about function prologues and epilogues (1.6 on page 28). CPU flags, however, are modified

\(^{16}\)You can read more about it in the section about function prologues and epilogues (1.6 on page 28).

\(^{17}\)CPU flags, however, are modified
This instruction has almost the same effect but the ECX register contents will be overwritten. The Intel C++ compiler supposedly uses POP ECX since this instruction's opcode is shorter than ADD ESP, x (1 byte for POP against 3 for ADD).

Here is an example of using POP instead of ADD from Oracle RDBMS:

<table>
<thead>
<tr>
<th>Listing 1.15: Oracle RDBMS 10.2 Linux (app.o file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:0800029A push ebx</td>
</tr>
<tr>
<td>.text:0800029B call qksfroChild</td>
</tr>
<tr>
<td>.text:080002A0 pop ecx</td>
</tr>
</tbody>
</table>

However, MSVC can do the same.

<table>
<thead>
<tr>
<th>Listing 1.16: MineSweeper from Windows 7 32-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:0102106F push 0</td>
</tr>
<tr>
<td>.text:01021071 call ds:time</td>
</tr>
<tr>
<td>.text:01021077 pop ecx</td>
</tr>
</tbody>
</table>

After calling printf(), the original C/C++ code contains the statement return 0 —return 0 as the result of the main() function.

In the generated code this is implemented by the instruction XOR EAX, EAX.

XOR is in fact just "eXclusive OR" but the compilers often use it instead of MOV EAX, 0—again because it is a slightly shorter opcode (2 bytes for XOR against 5 for MOV).

Some compilers emit SUB EAX, EAX, which means SUBtract the value in the EAX from the value in EAX. That in any case will result in zero.

The last instruction RET returns the control to the caller. Usually, this is C/C++ CRT code which in turn returns control to the OS.

GCC

Now let’s try to compile the same C/C++ code in the GCC 4.4.1 compiler in Linux: gcc 1.c -o 1. Next, with the assistance of the IDA disassembler, let’s see how the main() function was created. IDA, like MSVC, uses Intel-syntax.

<table>
<thead>
<tr>
<th>Listing 1.17: code in IDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>main proc near</td>
</tr>
<tr>
<td>var_10 = dword ptr -10h</td>
</tr>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>and esp, 0FFFFFFFF0h</td>
</tr>
<tr>
<td>sub esp, 10h</td>
</tr>
<tr>
<td>mov eax, offset aHelloWorld ; &quot;hello, world\n&quot;</td>
</tr>
<tr>
<td>mov [esp+10h+var_10], eax</td>
</tr>
<tr>
<td>call _printf</td>
</tr>
<tr>
<td>mov eax, 0</td>
</tr>
<tr>
<td>leave</td>
</tr>
<tr>
<td>retn</td>
</tr>
<tr>
<td>main endp</td>
</tr>
</tbody>
</table>

The result is almost the same. The address of the hello, world string (stored in the data segment) is loaded in the EAX register first, and then saved onto the stack.

In addition, the function prologue has AND ESP, 0FFFFFFFF0h —this instruction aligns the ESP register value on a 16-byte boundary. This results in all values in the stack being aligned the same way (The CPU performs better if the values it is dealing with are located in memory at addresses aligned on a 4-byte or 16-byte boundary).

SUB ESP, 10h allocates 16 bytes on the stack. Although, as we can see hereafter, only 4 are necessary here.

---

18 Wikipedia
19 C Runtime library
20 We could also have GCC produce assembly listings in Intel-syntax by applying the options -S -masm=intel.
This is because the size of the allocated stack is also aligned on a 16-byte boundary.

The string address (or a pointer to the string) is then stored directly onto the stack without using the PUSH instruction. `var_10`—is a local variable and is also an argument for `printf()`. Read about it below.

Then the `printf()` function is called.

Unlike MSVC, when GCC is compiling without optimization turned on, it emits `MOV EAX, 0` instead of a shorter opcode.

The last instruction, `LEAVE`—is the equivalent of the `MOV ESP, EBP` and `POP EBP` instruction pair—in other words, this instruction sets the stack pointer (ESP) back and restores the EBP register to its initial state. This is necessary since we modified these register values (ESP and EBP) at the beginning of the function (by executing `MOV EBP, ESP` / `AND ESP, ...`).

**GCC: AT&T syntax**

Let’s see how this can be represented in assembly language AT&T syntax. This syntax is much more popular in the UNIX-world.

```plaintext
Listing 1.18: let’s compile in GCC 4.7.3

gcc -S 1_1.c

We get this:

```plaintext
Listing 1.19: GCC 4.7.3

```

```plaintext
Listing 1.20: GCC 4.7.3

This GCC option can be used to eliminate “unnecessary” macros: `-fno-asynchronous-unwind-tables`

---

21 This GCC option can be used to eliminate “unnecessary” macros: `-fno-asynchronous-unwind-tables`
Some of the major differences between Intel and AT&T syntax are:

- Source and destination operands are written in opposite order.
  - In Intel-syntax: `<instruction> <destination operand> <source operand>`.
  - In AT&T syntax: `<instruction> <source operand> <destination operand>`.

  Here is an easy way to memorize the difference: when you deal with Intel-syntax, you can imagine that there is an equality sign (=) between operands and when you deal with AT&T-syntax imagine there is a right arrow (→).

- AT&T: Before register names, a percent sign must be written (%) and before numbers a dollar sign ($). Parentheses are used instead of brackets.

- AT&T: A suffix is added to instructions to define the operand size:
  - q — quad (64 bits)
  - l — long (32 bits)
  - w — word (16 bits)
  - b — byte (8 bits)

To go back to the compiled result: it is almost identical to what was displayed by IDA. There is one subtle difference: `0xFFFFFF0h` is presented as `-16`. It's the same thing: 16 in the decimal system is `0x10` in hexadecimal. `-0x10` is equal to `0xFFFFFFF0` (for a 32-bit data type).

One more thing: the return value is set to 0 by using the usual MOV, not XOR. MOV just loads a value to a register. Its name is a misnomer (as the data is not moved but rather copied). In other architectures, this instruction is named “LOAD” or “STORE” or something similar.

String patching (Win32)

We can easily find the “hello, world” string in the executable file using Hiew:

![Figure 1.1: Hiew](image)

And we can try to translate our message into Spanish:

22By the way, in some C standard functions (e.g., memcpy(), strcpy()) the arguments are listed in the same way as in Intel-syntax: first the pointer to the destination memory block, and then the pointer to the source memory block.
The Spanish text is one byte shorter than English, so we also added the 0x0A byte at the end (\n) with a zero byte.

It works.

What if we want to insert a longer message? There are some zero bytes after original English text. It’s hard to say if they can be overwritten: they may be used somewhere in CRT code, or maybe not. Anyway, only overwrite them if you really know what you’re doing.

**String patching (Linux x64)**

Let’s try to patch a Linux x64 executable using rada.re:

```
Listing 1.21: rada.re session

dennis@bigbox ~/tmp % gcc hw.c
dennis@bigbox ~/tmp % radare2 a.out
  -- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f
Searching 5 bytes in [0x400000-0x601040]
hits: 1
0x004005c4 hit0_0 .HHhello, world;0.
[0x004005c4]> oo+
File a.out reopened in read-write mode
[0x004005c4]> w hola, mundo\x00
[0x004005c4]> q
```

![Image of the program execution](image)

Figure 1.2: Hiew
Here’s what’s going on: I searched for the “hello” string using the `/` command, then I set the `cursor` (seek, in rada.re terms) to that address. Then I want to be sure that this is really that place: `px` dumps bytes there. `oo+` switches rada.re to `read-write` mode. `w` writes an ASCII string at the current `seek`. Note the `\00` at the end—this is a zero byte. `q` quits.

**This is a real story of software cracking**

An image processing software, when not registered, added watermarks, like “This image was processed by evaluation version of [software name]”, across a picture. We tried at random: we found that string in the executable file and put spaces instead of it. Watermarks disappeared. Technically speaking, they continued to appear. With the help of Qt functions, the watermark was still added to the resulting image. But adding spaces didn’t alter the image itself...

**Software localization of MS-DOS era**

This method was a common way to translate MS-DOS software to Russian language back to 1980’s and 1990’s. This technique is available even for those who are not aware of machine code and executable file formats. The new string shouldn’t be bigger than the old one, because there’s a risk of overwriting another value or code there. Russian words and sentences are usually slightly longer than its English counterparts, so that is why localized software has a lot of weird acronyms and hardly readable abbreviations.

![Localized Norton Commander 5.51](image)

Figure 1.3: *Localized* Norton Commander 5.51

Perhaps this also happened to other languages during that era, in other countries.

As for Delphi strings, the string’s size must also be corrected, if needed.

### 1.5.2 x86-64

**MSVC: x86-64**

Let's also try 64-bit MSVC:

```
Listing 1.22: MSVC 2012 x64

$SG2989 DB 'hello, world', 0AH, 00H

main PROC
sub rsp, 40
lea rcx, OFFSET FLAT:$SG2989
call printf
xor eax, eax
add rsp, 40
ret 0
main ENDP
```

In x86-64, all registers were extended to 64-bit, and now their names have an `R-` prefix. In order to use the stack less often (in other words, to access external memory/cache less often), there is a popular way to pass function arguments via registers (*fastcall* [6.1.3 on page 726](#)). I.e., a part of the function’s arguments are passed in registers, and the rest—via the stack. In Win64, 4 function arguments are passed in the RCX, RDX, R8, and R9 registers. That is what we see here: a pointer to the string for `printf()` is now passed not in the stack, but rather in the RCX register. The pointers are 64-bit now, so they are passed in the 64-bit registers (which have the `R-` prefix). However, for backward compatibility, it is still possible to access the 32-bit parts, using the `E-` prefix. This is how the RAX/EAX/AX/AL register looks like in x86-64:
The main() function returns an int-typed value, which in C/C++ is still 32-bit, for better backward compatibility and portability, so that is why the EAX register is cleared at the function end (i.e., the 32-bit part of the register) instead of with RAX. There are also 40 bytes allocated in the local stack. This is called the “shadow space”, which we’ll talk about later: 1.14.2 on page 100.

 GCC: x86-64

Let’s also try GCC in 64-bit Linux:

Listing 1.23: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
    sub    rsp, 8
    mov    edi, OFFSET FLAT:.LC0 ; "hello, world\n"
    xor    eax, eax ; number of vector registers passed
    call   printf
    xor    eax, eax
    add    rsp, 8
    ret
```


The first 6 arguments are passed in the RDI, RSI, RDX, RCX, R8, and R9 registers, and the rest—via the stack.

So the pointer to the string is passed in EDI (the 32-bit part of the register). Why doesn’t it use the 64-bit part, RDI?

It is important to keep in mind that all MOV instructions in 64-bit mode that write something into the lower 32-bit register part also clear the higher 32-bits (as stated in Intel manuals: 12.1.4 on page 999).

I.e., the MOV EAX, 011223344h writes a value into RAX correctly, since the higher bits will be cleared.

If we open the compiled object file (.o), we can also see all the instructions’ opcodes 24:

Listing 1.24: GCC 4.4.6 x64

```
.text:00000000004004D0  main proc near
.text:00000000004004D0  sub    rsp, 8
.text:00000000004004D4  BF E8 05 40 00  mov    edi, offset format ; "hello, world\n"
.text:00000000004004D9  31 C0  xor    eax, eax
.text:00000000004004DB  E8 D8 FE FF FF  call   _printf
.text:00000000004004E0  31 C0  xor    eax, eax
.text:00000000004004E2  48 83 C4 08  add    rbp, 8
.text:00000000004004E6  C3  ret
.text:00000000004004E6  main endp
```

As we can see, the instruction that writes into EDI at 0x4004D4 occupies 5 bytes. The same instruction writing a 64-bit value into RDI occupies 7 bytes. Apparently, GCC is trying to save some space. Besides, it can be sure that the data segment containing the string will not be allocated at the addresses higher than 4GiB.

We also see that the EAX register has been cleared before the printf() function call. This is done because according to ABI 25 standard mentioned above, the number of used vector registers is to be passed in EAX in *NIX systems on x86-64.

---

23 Also available as https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf
24 This must be enabled in Options → Disassembly → Number of opcode bytes
25 Application Binary Interface
Address patching (Win64)

If our example was compiled in MSVC 2013 using /MD switch (meaning a smaller executable due to MSVCR*.DLL file linkage), the main() function comes first, and can be easily found:

![Hiew](image)

Figure 1.4: Hiew

As an experiment, we can increment address by 1:
Hiew shows “ello, world”. And when we run the patched executable, this very string is printed.

**Pick another string from binary image (Linux x64)**

The binary file I’ve got when I compile our example using GCC 5.4.0 on Linux x64 box has many other text strings. They are mostly imported function names and library names.

Run objdump to get the contents of all sections of the compiled file:

```
$ objdump -s a.out

a.out:    file format elf64-x86-64

Contents of section .interp:
400238 2f6c6962 36342f6c 642d6c69 6e75782d /lib64/ld-linux-x86-64.so.2.
400248 7838362d 36342e73 6f2e3200 x86-64.so.2.

Contents of section .note.ABI-tag:
400254 04000000 10000000 01000000 474e5500 ............GNU.
400264 00000000 02000000 06000000 20000000 ...............

Contents of section .note.gnu.build-id:
400274 04000000 14000000 03000000 474e5500 ............GNU.
400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10 .F.x[.......^...
400294 cf3f7ae4 .?z.
```

It’s not a problem to pass address of the text string “/lib64/ld-linux-x86-64.so.2” to printf():

```c
#include <stdio.h>

int main()
```
It’s hard to believe, but this code prints the aforementioned string.

If you would change the address to 0x400260, the “GNU” string would be printed. This address is true for my specific GCC version, GNU toolset, etc. On your system, the executable may be slightly different, and all addresses will also be different. Also, adding/removing code to/from this source code will probably shift all addresses back or forward.

1.5.3 ARM

For my experiments with ARM processors, several compilers were used:

- Apple Xcode 4.6.3 IDE with the LLVM-GCC 4.2 compiler.

32-bit ARM code is used (including Thumb and Thumb-2 modes) in all cases in this book, if not mentioned otherwise. When we talk about 64-bit ARM here, we call it ARM64.

Non-optimizing Keil 6/2013 (ARM mode)

Let’s start by compiling our example in Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

The `armcc` compiler produces assembly listings in Intel-syntax, but it has high-level ARM-processor related macros, but it is more important for us to see the instructions “as is” so let’s see the compiled result in IDA.

```
Listing 1.25: Non-optimizing Keil 6/2013 (ARM mode) IDA
```

In the example, we can easily see each instruction has a size of 4 bytes. Indeed, we compiled our code for ARM mode, not for Thumb.

The very first instruction, STMFD SP!, {R4,LR}, works as an x86 PUSH instruction, writing the values of two registers (R4 and LR) into the stack.

Indeed, in the output listing from the `armcc` compiler, for the sake of simplification, actually shows the PUSH {r4,lr} instruction. But that is not quite precise. The PUSH instruction is only available in Thumb mode. So, to make things less confusing, we’re doing this in IDA.

This instruction first decrements the SP so it points to the place in the stack that is free for new entries, then it saves the values of the R4 and LR registers at the address stored in the modified SP.

This instruction (like the PUSH instruction in Thumb mode) is able to save several register values at once which can be very useful. By the way, this has no equivalent in x86. It can also be noted that the STMFD instruction is a generalization of the PUSH instruction (extending its features), since it can work with any register, not just with SP. In other words, STMFD may be used for storing a set of registers at the specified memory address.

---

26 It is indeed so: Apple Xcode 4.6.3 uses open-source GCC as front-end compiler and LLVM code generator
27 e.g. ARM mode lacks PUSH/POP instructions
28 STMFD
29 stack pointer. SP/ESP/RSP in x86/x64. SP in ARM.
The ADR R0, aHelloWorld instruction adds or subtracts the value in the PC\textsuperscript{31} register to the offset where the hello, world string is located. How is the PC register used here, one might ask? This is called “position-independent code”\textsuperscript{32}.

Such code can be executed at a non-fixed address in memory. In other words, this is PC-relative addressing. The ADR instruction takes into account the difference between the address of this instruction and the address where the string is located. This difference (offset) is always to be the same, no matter at what address our code is loaded by the OS. That’s why all we need is to add the address of the current instruction (from PC) in order to get the absolute memory address of our C-string.

BL __2printf\textsuperscript{33} instruction calls the printf() function. Here’s how this instruction works:

- store the address following the BL instruction (0xC) into the LR;
- then pass the control to printf() by writing its address into the PC register.

When printf() finishes its execution it must have information about where it needs to return the control to. That’s why each function passes control to the address stored in the LR register.

That is a difference between “pure” RISC-processors like ARM and CISC\textsuperscript{34}-processors like x86, where the return address is usually saved in the LR register. The very first instruction saves its value in the stack because the same register will be used by our main() function when calling printf(). In the function’s end, this value can be written directly to the PC register, thus passing control to where our function has been called.

Since main() is usually the primary function in C/C++, the control will be returned to the OS loader or to a point in a CRT, or something like that.

All that allows omitting the BX LR instruction at the end of the function.

DCB is an assembly language directive defining an array of bytes or ASCII strings, akin to the DB directive in the x86-assembly language.

Non-optimizing Keil 6/2013 (Thumb mode)

Let’s compile the same example using Keil in Thumb mode:

```
armcc.exe --thumb --c90 -O0 1.c
```

We are getting (in IDA):

Listing 1.26: Non-optimizing Keil 6/2013 (Thumb mode) + IDA

```
.text:00000000 main
.text:00000000 10 B5   PUSH   {R4,LR}
.text:00000002 C0 A0   ADR    R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9   BL    __2printf
.text:00000008 00 20   MOVS   R0, #0
.text:0000000A 10 BD   POP    {R4,PC}
.text:00000304 68 65 6C+aHelloWorld DCB "hello, world",0 ; DATA XREF: main+2
```

31Program Counter. IP/EIP/RIP in x86/64. PC in ARM.
32Read more about it in relevant section (6.4.1 on page 739)
33Branch with Link
34Complex Instruction Set Computing
35Meaning MOVe
36LDMFD\textsuperscript{37} is an inverse instruction of STMFD
We can easily spot the 2-byte (16-bit) opcodes. This is, as was already noted, Thumb. The BL instruction, however, consists of two 16-bit instructions. This is because it is impossible to load an offset for the printf() function while using the small space in one 16-bit opcode. Therefore, the first 16-bit instruction loads the higher 10 bits of the offset and the second instruction loads the lower 11 bits of the offset.

As was noted, all instructions in Thumb mode have a size of 2 bytes (or 16 bits). This implies it is impossible for a Thumb-instruction to be at an odd address whatsoever. Given the above, the last address bit may be omitted while encoding instructions.

In summary, the BL Thumb-instruction can encode an address in \( \text{current}_{PC} \pm \approx 2M \).

As for the other instructions in the function: PUSH and POP work here just like the described STMFD/LDMFD only the SP register is not mentioned explicitly here. ADR works just like in the previous example. MOVS writes 0 into the R0 register in order to return zero.

**Optimizing Xcode 4.6.3 (LLVM) (ARM mode)**

Xcode 4.6.3 without optimization turned on produces a lot of redundant code so we’ll study optimized output, where the instruction count is as small as possible, setting the compiler switch -O3.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000028C</td>
<td>MOV R0, #0</td>
<td>(MOVe T op) instruction writes 0 into higher 16 bits of the register. The issue here is that the generic MOV instruction in ARM mode may write only the lower 16 bits of the register.</td>
</tr>
<tr>
<td>0000028C</td>
<td>MOV R0, #0x1686</td>
<td>The MOV instruction just writes the number 0x1686 into the R0 register. This is the offset pointing to the “Hello world!” string.</td>
</tr>
<tr>
<td>0000028C</td>
<td>ADD R0, PC, R0</td>
<td>The PC register in order to return zero.</td>
</tr>
<tr>
<td>0000028C</td>
<td>MOVT R0, #0</td>
<td>The R7 register (as it is standardized in <a href="http://go.yurichev.com/17276">iOS ABI Function Call Guide, (2010)</a> is a frame pointer. More on that below.</td>
</tr>
<tr>
<td>0000028C</td>
<td>BL puts</td>
<td>The ADD R0, PC, R0 instruction adds the value in the PC to the value in the R0, to calculate the absolute address of the “Hello world!” string. As we already know, it is “position-independent code” so this correction is essential here.</td>
</tr>
<tr>
<td>0000028C</td>
<td>_hello_world</td>
<td>The BL instruction calls the puts() function instead of printf().</td>
</tr>
<tr>
<td>0000028C</td>
<td>cstring:0003B62 48 65 6C 6C+aHelloWorld_0 DCB &quot;Hello world!&quot;,0</td>
<td>LLVM has replaced the first printf() call with puts(). Indeed: printf() with a sole argument is almost analogous to puts().</td>
</tr>
</tbody>
</table>

The instructions STMFD and LDMFD are already familiar to us.

Keep in mind, all instruction opcodes in ARM mode are limited in size to 32 bits. Of course, this limitation is not related to moving data between registers. That’s why an additional instruction MOVT exists for writing into the higher bits (from 16 to 31 inclusive). Its usage here, however, is redundant because the MOV R0, #0x1686 instruction above cleared the higher part of the register. This is supposedly a shortcoming of the compiler.

The ADD R0, PC, R0 instruction above cleared the higher part of the register. This is supposedly a shortcoming of the compiler.

The BL instruction calls the puts() function instead of printf().

LLVM has replaced the first printf() call with puts(). Indeed: printf() with a sole argument is almost analogous to puts().

Almost, because the two functions are producing the same result only in case the string does not contain printf format identifiers starting with %. In case it does, the effect of these two functions would be different.

Why did the compiler replace the printf() with puts()? Presumably because puts() is faster. Because it just passes characters to stdout without comparing every one of them with the % symbol.

The instructions STMFD and LDMFD are already familiar to us.

Keep in mind, all instruction opcodes in ARM mode are limited in size to 32 bits. Of course, this limitation is not related to moving data between registers. That’s why an additional instruction MOVT exists for writing into the higher bits (from 16 to 31 inclusive). Its usage here, however, is redundant because the MOV R0, #0x1686 instruction above cleared the higher part of the register. This is supposedly a shortcoming of the compiler.

The ADD R0, PC, R0 instruction adds the value in the PC to the value in the R0, to calculate the absolute address of the “Hello world!” string. As we already know, it is “position-independent code” so this correction is essential here.
Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

By default Xcode 4.6.3 generates code for Thumb-2 in this manner:

```
Listing 1.28: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)
__text:00002B6C  hello_world
    _text:00002B6C 80 B5  PUSH {R7,LR} {R0, #0x13D8}
    _text:00002B6E 41 F2 D8 30  MOVW R0, #0x13D8
    _text:00002B72 6F 46  MOV R7, SP
    _text:00002B74 C0 F2 00 00  MOVT.W R0, #0
    _text:00002B78 78 44  ADD R0, SP
    _text:00002B7A 01 F0 38  EA  BLX _puts
__cstring:00003E70 48 65 6C 6F 20+aHelloWorld  DCB "Hello world!",0xA,0
```

The BL and BLX instructions in Thumb mode, as we recall, are encoded as a pair of 16-bit instructions. In Thumb-2 these surrogate opcodes are extended in such a way so that new instructions may be encoded here as 32-bit instructions.

That is obvious considering that the opcodes of the Thumb-2 instructions always begin with 0xFx or 0xEx. But in the IDA listing the opcode bytes are swapped because for ARM processor the instructions are encoded as follows: last byte comes first and after that comes the first one (for Thumb and Thumb-2 modes) or for instructions in ARM mode the fourth byte comes first, then the third, then the second and finally the first (due to different endianness).

So that is how bytes are located in IDA listings:
- for ARM and ARM64 modes: 4-3-2-1;
- for Thumb mode: 2-1;
- for 16-bit instructions pair in Thumb-2 mode: 2-1-4-3.

So as we can see, the MOVW, MOVT.W and BLX instructions begin with 0xFx.

One of the Thumb-2 instructions is MOVW R0, #0x13D8 —it stores a 16-bit value into the lower part of the R0 register, clearing the higher bits.

Also, MOVT.W R0, #0 works just like MOVT from the previous example only it works in Thumb-2.

Among the other differences, the BLX instruction is used in this case instead of the BL.

The difference is that, besides saving the RA in the LR register and passing control to the puts() function, the processor is also switching from Thumb/Thumb-2 mode to ARM mode (or back).

This instruction is placed here since the instruction to which control is passed looks like (it is encoded in ARM mode):
```
    _symbolstub1:00003FEC _puts ; CODE XREF: _hello_world+E
    _symbolstub1:00003FEC 44 F0 9F E5  LDR PC, =_imp_puts
```

This is essentially a jump to the place where the address of puts() is written in the imports’ section.

So, the observant reader may ask: why not call puts() right at the point in the code where it is needed? Because it is not very space-efficient.

Almost any program uses external dynamic libraries (like DLL in Windows, .so in *NIX or .dylib in Mac OS X). The dynamic libraries contain frequently used library functions, including the standard C-function puts().

In an executable binary file (Windows PE .exe, ELF or Mach-O) an import section is present. This is a list of symbols (functions or global variables) imported from external modules along with the names of the modules themselves.

The OS loader loads all modules it needs and, while enumerating import symbols in the primary module, determines the correct addresses of each symbol.

\(^{41}\) Return Address
In our case, \_imp\_puts is a 32-bit variable used by the OS loader to store the correct address of the function in an external library. Then the LDR instruction just reads the 32-bit value from this variable and writes it into the PC register, passing control to it.

So, in order to reduce the time the OS loader needs for completing this procedure, it is good idea to write the address of each symbol only once, to a dedicated place.

Besides, as we have already figured out, it is impossible to load a 32-bit value into a register while using only one instruction without a memory access.

Therefore, the optimal solution is to allocate a separate function working in ARM mode with the sole goal of passing control to the dynamic library and then to jump to this short one-instruction function (the so-called thunk function) from the Thumb-code.

By the way, in the previous example (compiled for ARM mode) the control is passed by the BL to the same thunk function. The processor mode, however, is not being switched (hence the absence of an “X” in the instruction mnemonic).

More about thunk-functions

Thunk-functions are hard to understand, apparently, because of a misnomer. The simplest way to understand it as adaptors or convertors of one type of jack to another. For example, an adaptor allowing the insertion of a British power plug into an American wall socket, or vice-versa. Thunk functions are also sometimes called wrappers.

Here are a couple more descriptions of these functions:

“\text{A piece of coding which provides an address:’’}, \text{ according to P. Z. Ingerman, who invented thunks in 1961 as a way of binding actual parameters to their formal definitions in Algol-60 procedure calls. If a procedure is called with an expression in the place of a formal parameter, the compiler generates a thunk which computes the expression and leaves the address of the result in some standard location.}

... .

Microsoft and IBM have both defined, in their Intel-based systems, a “16-bit environment” (with bletcherous segment registers and 64K address limits) and a “32-bit environment” (with flat addressing and semi-real memory management). The two environments can both be running on the same computer and OS (thanks to what is called, in the Microsoft world, WOW which stands for Windows On Windows). MS and IBM have both decided that the process of getting from 16- to 32-bit and vice versa is called a “thunk”; for Windows 95, there is even a tool, THUNK.EXE, called a “thunk compiler”.

(The Jargon File)

Another example we can find in LAPACK library—a “Linear Algebra PACKage” written in FORTRAN. C/C++ developers also want to use LAPACK, but it’s insane to rewrite it to C/C++ and then maintain several versions. So there are short C functions callable from C/C++ environment, which are, in turn, call FORTRAN functions, and do almost anything else:

\begin{verbatim}
double Blas_Dot_Prod(const LaVectorDouble &dx, const LaVectorDouble &dy)
{
    assert(dx.size()==dy.size());
    integer n = dx.size();
    integer incx = dx.inc(), incy = dy.inc();
    return F77NAME(ddot)(&n, &dx(0), &incx, &dy(0), &incy);
}
\end{verbatim}

Also, functions like that are called “wrappers”.

**ARM64**

**GCC**

Let’s compile the example using GCC 4.8.1 in ARM64:
Listing 1.29: Non-optimizing GCC 4.8.1 + objdump

Listing 1.30: main() returning a value of uint64_t type

There are no Thumb and Thumb-2 modes in ARM64, only ARM, so there are 32-bit instructions only. The Register count is doubled: .2.4 on page 1027. 64-bit registers have X- prefixes, while its 32-bit parts—W-. The STP instruction (Store Pair) saves two registers in the stack simultaneously: X29 and X30.

Of course, this instruction is able to save this pair at an arbitrary place in memory, but the SP register is specified here, so the pair is saved in the stack. ARM64 registers are 64-bit ones, each has a size of 8 bytes, so one needs 16 bytes for saving two registers.

The exclamation mark ("!") after the operand means that 16 is to be subtracted from SP first, and only then are values from register pair to be written into the stack. This is also called pre-index. About the difference between post-index and pre-index read here: 1.39.2 on page 437.

Hence, in terms of the more familiar x86, the first instruction is just an analogue to a pair of PUSH X29 and PUSH X30. X29 is used as FP in ARM64, and X30 as LR, so that’s why they are saved in the function prologue and restored in the function epilogue.

The second instruction copies SP in X29 (or FP). This is made so to set up the function stack frame. ADRP and ADD instructions are used to fill the address of the string “Hello!” into the X0 register, because the first function argument is passed in this register. There are no instructions, whatsoever, in ARM that can store a large number into a register (because the instruction length is limited to 4 bytes, read more about it here: 1.39.3 on page 438). So several instructions must be utilized. The first instruction (ADRP) writes the address of the 4KiB page, where the string is located, into X0, and the second one (ADD) just adds the remainder to the address. More about that in: 1.39.4 on page 439.

0x400000 + 0x648 = 0x400648, and we see our “Hello!” C-string in the .rodata data segment at this address.

puts() is called afterwards using the BL instruction. This was already discussed: 1.5.3 on page 20.

MOV writes 0 into W0. W0 is the lower 32 bits of the 64-bit X0 register:

<table>
<thead>
<tr>
<th>High 32-bit part</th>
<th>low 32-bit part</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0</td>
<td>W0</td>
</tr>
</tbody>
</table>

The function result is returned via X0 and main() returns 0, so that’s how the return result is prepared. But why use the 32-bit part?

Because the int data type in ARM64, just like in x86-64, is still 32-bit, for better compatibility.

So if a function returns a 32-bit int, only the lower 32 bits of X0 register have to be filled.

In order to verify this, let’s change this example slightly and recompile it. Now main() returns a 64-bit value.
The result is the same, but that’s how MOV at that line looks like now:

```
Listing 1.31: Non-optimizing GCC 4.8.1 + objdump
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Register(s)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>4005a4</td>
<td>d2800000</td>
<td>mov x0, #0x0</td>
<td>// #0</td>
</tr>
</tbody>
</table>

LDP (Load Pair) then restores the X29 and X30 registers.

There is no exclamation mark after the instruction: this implies that the values are first loaded from the stack, and only then is SP increased by 16. This is called *post-index*.

A new instruction appeared in ARM64: RET. It works just as BX LR, only a special *hint* bit is added, informing the CPU that this is a return from a function, not just another jump instruction, so it can execute it more optimally.

Due to the simplicity of the function, optimizing GCC generates the very same code.

### 1.5.4 MIPS

**A word about the “global pointer”**

One important MIPS concept is the “global pointer”. As we may already know, each MIPS instruction has a size of 32 bits, so it’s impossible to embed a 32-bit address into one instruction: a pair has to be used for this (like GCC did in our example for the text string address loading). It’s possible, however, to load data from the address in the range of `register−32768...register + 32767` using one single instruction (because 16 bits of signed offset could be encoded in a single instruction). So we can allocate some register for this purpose and also allocate a 64KiB area of most used data. This allocated register is called a “global pointer” and it points to the middle of the 64KiB area. This area usually contains global variables and addresses of imported functions like `printf()`, because the GCC developers decided that getting the address of some function must be as fast as a single instruction execution instead of two. In an ELF file this 64KiB area is located partly in sections .sbss ("small BSS") for uninitialized data and .sdata ("small data") for initialized data. This implies that the programmer may choose what data he/she wants to be accessed fast and place it into .sdata/.sbss. Some old-school programmers may recall the MS-DOS memory model 11.6 on page 986 or the MS-DOS memory managers like XMS/EMS where all memory was divided in 64KiB blocks.

This concept is not unique to MIPS. At least PowerPC uses this technique as well.

#### Optimizing GCC

Let’s consider the following example, which illustrates the “global pointer” concept.

```
Listing 1.32: Optimizing GCC 4.4.5 (assembly output)
```

```
$LC0:
; \\
; \000 is zero byte in octal base:
.asci "Hello, world!\012\000"
main:
; function prologue.
; set the GP:
    lui $28,%hi(__gnu_local_gp)
    addiu $sp,$sp,-32
    addiu $28,$28,%lo(__gnu_local_gp)
; save the RA to the local stack:
    sw $31,28($sp)
; load the address of the puts() function from the GP to $25:
    lw $25,%call16(puts)$28)
; load the address of the text string to $4 ($a0):
    lui $4,%hi($LC0)
; jump to puts(), saving the return address in the link register:
    jalr $25
    addiu $4,$4,%lo($LC0) ; branch delay slot
; restore the RA:
    lw $31,28($sp)
; copy 0 from $zero to $v0:
```

[^4]: Block Started by Symbol
Another interesting fact is that MIPS lacks an instruction that transfers data between registers. In fact, the $GP register is set in the function prologue to point to the middle of this area. The JALR ("Jump and Link Register") jumps to the address stored in the $25 register (address of the function prologue). The address of the text string is loaded to $4 using LUI ("Load Upper Immediate") and ADDIU ("Add Immediate Unsigned Word") instruction pair. LUI sets the high 16 bits of the register (hence “upper” word in instruction name) and ADDIU adds the lower 16 bits of the address.

ADDIU follows JALR (haven’t you forgot branch delay slots yet?). The register $4 is also called $A0, which is used for passing the first function argument.

JALR ("Jump and Link Register") jumps to the address stored in the $25 register (address of puts()) while saving the address of the next instruction (LW) in RA. This is very similar to ARM. Oh, and one important thing is that the address saved in RA is not the address of the next instruction (because it’s in a delay slot and is executed before the jump instruction), but the address of the instruction after the next one (after the delay slot). Hence, $PC + 8 is written to RA during the execution of JALR, in our case, this is the address of the LW instruction next to ADDIU.

LW ("Load Word") at line 20 restores RA from the local stack (this instruction is actually part of the function epilogue).

MOVE at line 22 copies the value from the $0 ($ZERO) register to $2 ($V0).

MIPS has a constant register, which always holds zero. Apparently, the MIPS developers came up with the idea that zero is in fact the busiest constant in the computer programming, so let’s just use the $0 register every time zero is needed.

Another interesting fact is that MIPS lacks an instruction that transfers data between registers. In fact, MOVE DST, SRC is ADD DST, SRC, $ZERO ($DST = $SRC + 0), which does the same. Apparently, the MIPS developers wanted to have a compact opcode table. This does not mean an actual addition happens at each MOVE instruction. Most likely, the CPU optimizes these pseudo instructions and the ALU is never used.

J at line 24 jumps to the address in RA, which is effectively performing a return from the function. ADDIU after J is in fact executed before J (remember branch delay slots?) and is part of the function epilogue. Here is also a listing generated by IDA. Each register here has its own pseudo name:

Listing 1.33: Optimizing GCC 4.4.5 (IDA)

---

As we see, the $GP register is set in the function prologue to point to the middle of this area. The RA register is also saved in the local stack. puts() is also used here instead of printf(). The address of the puts() function is loaded into $25 using LW the instruction ("Load Word"). Then the address of the text string is loaded to $4 using LUI ("Load Upper Immediate") and ADDIU ("Add Immediate Unsigned Word") instruction pair. LUI sets the high 16 bits of the register (hence “upper” word in instruction name) and ADDIU adds the lower 16 bits of the address.

ADDIU follows JALR (haven’t you forgot branch delay slots yet?). The register $4 is also called $A0, which is used for passing the first function argument.

JALR ("Jump and Link Register") jumps to the address stored in the $25 register (address of puts() while saving the address of the next instruction (LW) in RA. This is very similar to ARM. Oh, and one important thing is that the address saved in RA is not the address of the next instruction (because it’s in a delay slot and is executed before the jump instruction), but the address of the instruction after the next one (after the delay slot). Hence, $PC + 8 is written to RA during the execution of JALR, in our case, this is the address of the LW instruction next to ADDIU.

LW ("Load Word") at line 20 restores RA from the local stack (this instruction is actually part of the function epilogue).

MOVE at line 22 copies the value from the $0 ($ZERO) register to $2 ($V0).

MIPS has a constant register, which always holds zero. Apparently, the MIPS developers came up with the idea that zero is in fact the busiest constant in the computer programming, so let’s just use the $0 register every time zero is needed.

Another interesting fact is that MIPS lacks an instruction that transfers data between registers. In fact, MOVE DST, SRC is ADD DST, SRC, $ZERO ($DST = $SRC + 0), which does the same. Apparently, the MIPS developers wanted to have a compact opcode table. This does not mean an actual addition happens at each MOVE instruction. Most likely, the CPU optimizes these pseudo instructions and the ALU is never used.

J at line 24 jumps to the address in RA, which is effectively performing a return from the function. ADDIU after J is in fact executed before J (remember branch delay slots?) and is part of the function epilogue. Here is also a listing generated by IDA. Each register here has its own pseudo name:

Listing 1.33: Optimizing GCC 4.4.5 (IDA)

---

$44$The MIPS registers table is available in appendix 3.1 on page 1028

$45$Arithmetic Logic Unit
The instruction at line 15 saves the GP value into the local stack, and this instruction is missing mysteriously from the GCC output listing, maybe by a GCC error. The GP value has to be saved indeed, because each function can use its own 64KiB data window. The register containing the puts() address is called $T9, because registers prefixed with T- are called “temporaries” and their contents may not be preserved.

Non-optimizing GCC

Non-optimizing GCC is more verbose.

We see here that register FP is used as a pointer to the stack frame. We also see 3 NOPs. The second and third of which follow the branch instructions. Perhaps the GCC compiler always adds NOPs (because of branch delay slots) after branch instructions and then, if optimization is turned on, maybe eliminates them. So in this case they are left here.

Here is also IDA listing:

```assembly
Listing 1.35: Non-optimizing GCC 4.4.5 (IDA)

.text:000000000 main:

; function prologue.
; save the RA ($31) and FP in the stack:
addiu $sp,$sp,-32
sw $31,28($sp)
sw $fp,24($sp)
; set the FP (stack frame pointer):
move $fp,$sp
; set the GP:
lui $28,%hi(__gnu_local_gp)
addiu $28,$28,%lo(__gnu_local_gp)
; load the address of the text string:
  ; load the address of puts() using the GP:
lw $2,%call16(puts($28)
  ; call puts():
move $25,$2
jalr $25
nop ; branch delay slot
; restore the GP from the local stack:
lw $28,16($fp)
; set register $2 ($V0) to zero:
move $2,$0
; function epilogue.
; restore the SP:
move $sp,$fp
; restore the RA:
lw $31,28($sp)
; restore the FP:
lw $fp,24($sp)
addiu $sp,$sp,32
; jump to the RA:
j $31
nop ; branch delay slot
```

46 Apparently, functions generating listings are not so critical to GCC users, so some unfixed cosmetic bugs may still exist.
Interestingly, IDA recognized the LUI/ADDIU instructions pair and coalesced them into one LA (“Load Address”) pseudo instruction at line 15. We may also see that this pseudo instruction has a size of 8 bytes! This is a pseudo instruction (or macro) because it’s not a real MIPS instruction, but rather a handy name for an instruction pair.

Another thing is that IDA doesn’t recognize NOP instructions, so here they are at lines 22, 26 and 41. It is **OR $AT, $ZERO**. Essentially, this instruction applies the OR operation to the contents of the $AT register with zero, which is, of course, an idle instruction. MIPS, like many other ISAs, doesn’t have a separate NOP instruction.

**Role of the stack frame in this example**

The address of the text string is passed in the register. Why setup a local stack anyway? The reason for this lies in the fact that the values of registers RA and GP have to be saved somewhere (because printf() is called), and the local stack is used for this purpose. If this was a leaf function, it would have been possible to get rid of the function prologue and epilogue, for example: 1.4.3 on page 8.

**Optimizing GCC: load it into GDB**

*Listing 1.36: sample GDB session*

```
root@debian-mips:~# gcc hw.c -O3 -o hw
root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian...
Reading symbols from /root/hw...(no debugging symbols found)...done.
```
1.5.5 Conclusion

The main difference between x86/ARM and x64/ARM64 code is that the pointer to the string is now 64-bits in length. Indeed, modern CPUs are now 64-bit due to both the reduced cost of memory and the greater demand for it by modern applications. We can add much more memory to our computers than 32-bit pointers are able to address. As such, all pointers are now 64-bit.

1.5.6 Exercises

- [http://challenges.re/48](http://challenges.re/48)
- [http://challenges.re/49](http://challenges.re/49)

1.6 Function prologue and epilogue

A function prologue is a sequence of instructions at the start of a function. It often looks something like the following code fragment:

```assembly
push ebp
mov ebp, esp
sub esp, X
```

What these instructions do: save the value of the EBP register on the stack, set the value of the EBP register to the value of the ESP and then allocate space on the stack for local variables.

The value in the EBP stays the same over the period of the function execution and is to be used for local variables and arguments access. For the same purpose one can use ESP, but since it changes over time this approach is not too convenient.

The function epilogue frees the allocated space in the stack, returns the value in the EBP register back to its initial state and returns the control flow to the caller:

```assembly
mov esp, ebp
pop ebp
```
Function prologues and epilogues are usually detected in disassemblers for function delimitation.

1.6.1 Recursion

Epilogues and prologues can negatively affect the recursion performance. More about recursion in this book: 3.7.3 on page 481.

1.7 An Empty Function: redux

Let's back to the empty function example 1.3 on page 5. Now that we know about function prologue and epilogue, this is an empty function 1.1 on page 5 compiled by non-optimizing GCC:

Listing 1.37: Non-optimizing GCC 8.2 x64 (assembly output)

```assembly
f:
    push rbp
    mov rbp, rsp
    nop
    pop rbp
    ret
```

It's RET, but function prologue and epilogue, probably, wasn't optimized and left as is. NOP is seems another compiler artefact. Anyway, the only effective instruction here is RET. All other instructions can be removed (or optimized).

1.8 Returning Values: redux

Again, when we know about function prologue and epilogue, let's recompile an example returning a value ( 1.4 on page 7, 1.8 on page 7) using non-optimizing GCC:

Listing 1.38: Non-optimizing GCC 8.2 x64 (assembly output)

```assembly
f:
    push rbp
    mov rbp, rsp
    mov eax, 123
    pop rbp
    ret
```

Effective instructions here are MOV and RET, others are – prologue and epilogue.

1.9 Stack

The stack is one of the most fundamental data structures in computer science 47. AKA 48 LIFO 49. Technically, it is just a block of memory in process memory along with the ESP or RSP register in x86 or x64, or the SP register in ARM, as a pointer within that block.

The most frequently used stack access instructions are PUSH and POP (in both x86 and ARM Thumb-mode). PUSH subtracts from ESP/RSP/SP 4 in 32-bit mode (or 8 in 64-bit mode) and then writes the contents of its sole operand to the memory address pointed by ESP/RSP/SP.

POP is the reverse operation: retrieve the data from the memory location that SP points to, load it into the instruction operand (often a register) and then add 4 (or 8) to the stack pointer.

After stack allocation, the stack pointer points at the bottom of the stack. PUSH decreases the stack pointer and POP increases it. The bottom of the stack is actually at the beginning of the memory allocated for the stack block. It seems strange, but that's the way it is.

47 wikipedia.org/wiki/Call_stack
48 Also Known As
49 Last In First Out
ARM supports both descending and ascending stacks.

For example the `STMFD/LDMFD, STMED/LMED` instructions are intended to deal with a descending stack (grows downwards, starting with a high address and progressing to a lower one). The `STMFA/LDMFA, STMEA/LMEIA` instructions are intended to deal with an ascending stack (grows upwards, starting from a low address and progressing to a higher one).

### 1.9.1 Why does the stack grow backwards?

Intuitively, we might think that the stack grows upwards, i.e. towards higher addresses, like any other data structure.

The reason that the stack grows backward is probably historical. When the computers were big and occupied a whole room, it was easy to divide memory into two parts, one for the heap and one for the stack. Of course, it was unknown how big the heap and the stack would be during program execution, so this solution was the simplest possible.

![Stack and Heap Diagram](image)

In D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System, (1974)* we can read:

> The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a nonshared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

This reminds us how some students write two lecture notes using only one notebook: notes for the first lecture are written as usual, and notes for the second one are written from the end of notebook, by flipping it. Notes may meet each other somewhere in between, in case of lack of free space.

### 1.9.2 What is the stack used for?

**Save the function’s return address**

*x86*

When calling another function with a CALL instruction, the address of the point exactly after the CALL instruction is saved to the stack and then an unconditional jump to the address in the CALL operand is executed.

The CALL instruction is equivalent to a

PUSH `address_after_call` / JMP `operand` instruction pair.

RET fetches a value from the stack and jumps to it — that is equivalent to a POP `tmp` / JMP `tmp` instruction pair.

**Overflowing the stack is straightforward.** Just run eternal recursion:

```c
void f()
{
    f();
}
```

---

50Store Multiple Empty Descending (ARM instruction)
51Load Multiple Empty Descending (ARM instruction)
52Store Multiple Full Ascending (ARM instruction)
53Load Multiple Full Ascending (ARM instruction)
54Store Multiple Empty Ascending (ARM instruction)
55Load Multiple Empty Ascending (ARM instruction)
56Also available as http://go.yurichev.com/17270
MSVC 2008 reports the problem:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.
ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all control paths, function will cause \n  \ runtime stack overflow
...but generates the right code anyway:

?f@YAXXZ PROC ; f
  ; Line 2
  push   ebp
  mov    ebp, esp
  ; Line 3
  call   ?f@YAXXZ ; f
  ; Line 4
  pop    ebp
  ret    0
?f@YAXXZ ENDP ; f

...Also if we turn on the compiler optimization (/Ox option) the optimized code will not overflow the stack and will work correctly\(^57\) instead:

?f@YAXXZ PROC ; f
  ; Line 2
  $LL3@f:
  ; Line 3
  jmp    SHORT $LL3@f
?f@YAXXZ ENDP ; f
```

GCC 4.4.1 generates similar code in both cases without, however, issuing any warning about the problem.

**ARM**

ARM programs also use the stack for saving return addresses, but differently. As mentioned in “Hello, world!” (1.5.3 on page 18), the RA is saved to the LR (link register). If one needs, however, to call another function and use the LR register one more time, its value has to be saved. Usually it is saved in the function prologue.

Often, we see instructions like `PUSH R4-R7, LR` along with this instruction in epilogue `POP R4-R7, PC`—thus register values to be used in the function are saved in the stack, including LR.

Nevertheless, if a function never calls any other function, in RISC terminology it is called a leaf function\(^58\). As a consequence, leaf functions do not save the LR register (because they don’t modify it). If such function is small and uses a small number of registers, it may not use the stack at all. Thus, it is possible to call leaf functions without using the stack, which can be faster than on older x86 machines because external RAM is not used for the stack\(^59\). This can be also useful for situations when memory for the stack is not yet allocated or not available.

Some examples of leaf functions: 1.14.3 on page 103, 1.14.3 on page 103, 1.282 on page 313, 1.298 on page 330, 1.28.5 on page 331, 1.192 on page 209, 1.190 on page 207, 1.209 on page 225.

**Passing function arguments**

The most popular way to pass parameters in x86 is called “cdecl”:

\(^{57}\)irony here

\(^{58}\)infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

\(^{59}\)Some time ago, on PDP-11 and VAX, the CALL instruction (calling other functions) was expensive; up to 50% of execution time might be spent on it, so it was considered that having a big number of small functions is an anti-pattern [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12

**Callee** functions get their arguments via the stack pointer.

Therefore, this is how the argument values are located in the stack before the execution of the f() function’s very first instruction:

<table>
<thead>
<tr>
<th></th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>argument#1, marked in IDA as arg_0</td>
</tr>
<tr>
<td>ESP+4</td>
<td>argument#2, marked in IDA as arg_4</td>
</tr>
<tr>
<td>ESP+8</td>
<td>argument#3, marked in IDA as arg_8</td>
</tr>
<tr>
<td>ESP+0xC</td>
<td>...</td>
</tr>
</tbody>
</table>

For more information on other calling conventions see also section (6.1 on page 725).

By the way, the **callee** function does not have any information about how many arguments were passed. C functions with a variable number of arguments (like printf()) determine their number using format string specifiers (which begin with the % symbol).

If we write something like:

```c
printf("%d %d %d", 1234);
```

printf() will print 1234, and then two random numbers[^60], which were lying next to it in the stack.

That’s why it is not very important how we declare the main() function: as main(), main(int argc, char *argv[]) or main(int argc, char *argv[], char *envp[]).

In fact, the **CRT**-code is calling main() roughly as:

```c
push envp
push argv
push argc
call main
```

If you declare main() as main() without arguments, they are, nevertheless, still present in the stack, but are not used. If you declare main() as main(int argc, char *argv[]), you will be able to use first two arguments, and the third will remain “invisible” for your function. Even more, it is possible to declare main(int argc), and it will work.

Another related example: 6.1.10.

**Alternative ways of passing arguments**

It is worth noting that nothing obliges programmers to pass arguments through the stack. It is not a requirement. One could implement any other method without using the stack at all.

A somewhat popular way among assembly language newbies is to pass arguments via global variables, like:

```
Listing 1.39: Assembly code
```

```
... 
mov X, 123 
mov Y, 456 
call do_something 
... 
```

[^60]: Not random in strict sense, but rather unpredictable: 1.9.4 on page 36
But this method has obvious drawback: do_something() function cannot call itself recursively (or via another function), because it has to zap its own arguments. The same story with local variables: if you hold them in global variables, the function couldn’t call itself. And this is also not thread-safe. A method to store such information in stack makes this easier—it can hold as many function arguments and/or values, as much space it has.


MS-DOS had a way of passing all function arguments via registers, for example, this is piece of code for ancient 16-bit MS-DOS prints “Hello, world!”:

```
mov dx, msg       ; address of message
mov ah, 9         ; 9 means "print string" function
int 21h           ; DOS "syscall"

mov ah, 4ch        ; "terminate program" function
int 21h            ; DOS "syscall"

msg db 'Hello, World!'\$
```

This is quite similar to 6.1.3 on page 726 method. And also it’s very similar to calling syscalls in Linux (6.3.1 on page 739) and Windows.

If a MS-DOS function is going to return a boolean value (i.e., single bit, usually indicating error state), CF flag was often used.

For example:

```
mov ah, 3ch       ; create file
lea dx, filename
mov cl, 1
int 21h
jc error
mov file_handle, ax
...
error:
...
```

In case of error, CF flag is raised. Otherwise, handle of newly created file is returned via AX.

This method is still used by assembly language programmers. In Windows Research Kernel source code (which is quite similar to Windows 2003) we can find something like this (file base/ntos/ke/i386/cpu.asm):

```
public Get386Stepping
Get386Stepping proc
    call MultiplyTest            ; Perform multiplication test
    jnc short G3s00              ; if nc, muttest is ok
    mov ax, 0
    ret
G3s00:
    call Check386B0              ; Check for B0 stepping
    jnc short G3s05              ; if nc, it's B1/later
    mov ax, 100h
    ret
G3s05:
    call Check386D1              ; Check for D1 stepping
    jc short G3s10               ; if c, it is NOT D1
    mov ax, 301h
    ret
```

\(^{61}\)Correctly implemented, each thread would have its own stack with its own arguments/variables.
G3s10:

```
mov    ax, 101h  ; assume it is B1 stepping
ret
...
```

```
MultiplyTest proc
```

```
xor    cx,cx   ; 64K times is a nice round number
mlt00:  push   cx
    call    Multiply    ; does this chip's multiply work?
pop     cx
    jc      short mltx  ; if c, No, exit
    loop    mlt00      ; if nc, YEs, loop to try again
    clc      
mltx:    ret
```

```
MultiplyTest endp
```

**Local variable storage**

A function could allocate space in the stack for its local variables just by decreasing the stack pointer towards the stack bottom.

Hence, it's very fast, no matter how many local variables are defined. It is also not a requirement to store local variables in the stack. You could store local variables wherever you like, but traditionally this is how it's done.

**x86: malloc() function**

It is worth noting the `alloca()` function. This function works like `malloc()`, but allocates memory directly on the stack. The allocated memory chunk does not have to be freed via a `free()` function call, since the function epilogue (1.6 on page 28) returns ESP back to its initial state and the allocated memory is just dropped. It is worth noting how `alloca()` is implemented. In simple terms, this function just shifts ESP downwards toward the stack bottom by the number of bytes you need and sets ESP as a pointer to the allocated block.

Let's try:

```
#define __GNUC__
#include <alloca.h>  // GCC
#else
#include <malloc.h>  // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#define __GNUC__
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3);  // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3);  // MSVC
#endif
    puts (buf);
}
```

_`snprintf()` function works just like `printf()`, but instead of dumping the result into `stdout` (e.g., to terminal or console), it writes it to the buf buffer. Function `puts()` copies the contents of `buf` to `stdout`. Of course, these two function calls might be replaced by one `printf()` call, but we have to illustrate small buffer usage.

---

62In MSVC, the function implementation can be found in `alloca16.asm` and `chkstk.asm` in `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`
MSVC

Let's compile (MSVC 2010):

Listing 1.40: MSVC 2010

```plaintext
... 
    mov  eax, 600 ; 00000258H
    call __alloca_probe_16
    mov  esi, esp

    push 3
    push 2
    push 1
    push OFFSET $SG2672
    push 600 ; 00000258H
    push esi
    call __snprintf
    push esi
    call _puts
    add  esp, 28
... 
```

The sole `alloca()` argument is passed via EAX (instead of pushing it into the stack).

GCC + Intel syntax

GCC 4.4.1 does the same without calling external functions:

Listing 1.41: GCC 4.7.3

```plaintext
.LC0:
  .string "hi! %d, %d, %d\n"
f:
    push  ebp
    mov   ebp, esp
    push  ebx
    sub   esp, 660
    lea   ebx, [esp+39]
    and   ebx, -16
    mov   DWORD PTR [esp], ebx ; s
    mov   DWORD PTR [esp+20], 3
    mov   DWORD PTR [esp+16], 2
    mov   DWORD PTR [esp+12], 1
    mov   DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
    mov   DWORD PTR [esp+4], 600 ; maxlen
    call __snprintf
    mov   DWORD PTR [esp], ebx ; s
    call  _puts
    mov   ebx, DWORD PTR [ebp-4]
    leave
    ret
```

GCC + AT&T syntax

Let's see the same code, but in AT&T syntax:

Listing 1.42: GCC 4.7.3

```plaintext
.LC0:

63 It is because `alloca()` is rather a compiler intrinsic (11.3 on page 981) than a normal function. One of the reasons we need a separate function instead of just a couple of instructions in the code, is because the MSVC64 `alloca()` implementation also has code which reads from the memory just allocated, in order to let the OS map physical memory to this VM65 region. After the `alloca()` call, ESP points to the block of 600 bytes and we can use it as memory for the buf array.
```
The code is the same as in the previous listing.

By the way, movl $3, 20(%esp) corresponds to mov DWORD PTR [esp+20], 3 in Intel-syntax. In the AT&T syntax, the register+offset format of addressing memory looks like offset(%register).

(Windows) SEH

SEH records are also stored on the stack (if they are present). Read more about it: (6.5.3 on page 755).

Buffer overflow protection

More about it here (1.26.2 on page 271).

Automatic deallocation of data in stack

Perhaps the reason for storing local variables and SEH records in the stack is that they are freed automatically upon function exit, using just one instruction to correct the stack pointer (it is often ADD). Function arguments, as we could say, are also deallocated automatically at the end of function. In contrast, everything stored in the heap must be deallocated explicitly.

1.9.3 A typical stack layout

A typical stack layout in a 32-bit environment at the start of a function, before the first instruction execution looks like this:

| ESP-0xC | local variable#2, marked in IDA as var_8 |
| ESP-8   | local variable#1, marked in IDA as var_4 |
| ESP-4   | saved value of EBP                      |
| ESP     | Return Address                           |
| ESP+4   | argument#1, marked in IDA as arg_0       |
| ESP+8   | argument#2, marked in IDA as arg_4       |
| ESP+0xC  | argument#3, marked in IDA as arg_8       |
| ...     | ...                                       |

1.9.4 Noise in stack

When one says that something seems random, what one usually means in practice is that one cannot see any regularities in it.

---

Stephen Wolfram, A New Kind of Science.
Often in this book “noise” or “garbage” values in the stack or memory are mentioned. Where do they come from? These are what has been left there after other functions’ executions. Short example:

```c
#include <stdio.h>

void f1()
{
    int a=1, b=2, c=3;
};

void f2()
{
    int a, b, c;
    printf ("%d, %d, %d\n", a, b, c);
};

int main()
{
    f1();
    f2();
};
```

Compiling ...

```
Listing 1.43: Non-optimizing MSVC 2010

$SG2752 DB ' %d, %d, %d', 0AH, 00H

_c$ = -12 ; size = 4
_b$ = -8 ; size = 4
_a$ = -4 ; size = 4

_f1 PROC
    push ebp
    mov ebp, esp
    sub esp, 12
    mov DWORD PTR _a$[ebp], 1
    mov DWORD PTR _b$[ebp], 2
    mov DWORD PTR _c$[ebp], 3
    mov esp, ebp
    pop ebp
    ret 0

_f1 ENDP

_c$ = -12 ; size = 4
_b$ = -8 ; size = 4
_a$ = -4 ; size = 4

_f2 PROC
    push ebp
    mov ebp, esp
    sub esp, 12
    mov eax, DWORD PTR _c$[ebp]
    push eax
    mov ecx, DWORD PTR _b$[ebp]
    push ecx
    mov edx, DWORD PTR _a$[ebp]
    push edx
    push OFFSET $SG2752 ; '%d, %d, %d'
    call DWORD PTR __imp__printf
    add esp, 16
    mov esp, ebp
    pop ebp
    ret 0

_f2 ENDP

_main PROC
    push ebp
    mov ebp, esp
    call _f1
    call _f2
    xor eax, eax
    pop ebp
```
The compiler will grumble a little bit...

```
c:\Polygon\c>cl st.c /Fast.asm /MD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.40219.01 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

st.c
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'c' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'b' used
c:\polygon\c\st.c(11) : warning C4700: uninitialized local variable 'a' used
Microsoft (R) Incremental Linker Version 10.00.40219.01
Copyright (C) Microsoft Corporation. All rights reserved.
/out:st.exe
st.obj
```

But when we run the compiled program ...

```
c:\Polygon\c>st
1, 2, 3
```

Oh, what a weird thing! We did not set any variables in f2(). These are “ghosts” values, which are still in the stack.
Let's load the example into OllyDbg:

![OllyDbg](image)

Figure 1.6: OllyDbg: f1()

When f1() assigns the variables $a$, $b$ and $c$, their values are stored at the address 0x1FF860 and so on.
And when `f2()` executes:

![Image of OllyDbg](image)

Figure 1.7: OllyDbg: `f2()`

... `a`, `b` and `c` of `f2()` are located at the same addresses! No one has overwritten the values yet, so at that point they are still untouched. So, for this weird situation to occur, several functions have to be called one after another and `SP` has to be the same at each function entry (i.e., they have the same number of arguments). Then the local variables will be located at the same positions in the stack. Summarizing, all values in the stack (and memory cells in general) have values left there from previous function executions. They are not random in the strict sense, but rather have unpredictable values. Is there another option? It would probably be possible to clear portions of the stack before each function execution, but that’s too much extra (and unnecessary) work.

**MSVC 2013**

The example was compiled by MSVC 2010. But the reader of this book made attempt to compile this example in MSVC 2013, ran it, and got all 3 numbers reversed:

```
c:\Polygon\c>st
3, 2, 1
```

Why? I also compiled this example in MSVC 2013 and saw this:

**Listing 1.44: MSVC 2013**

```
as  = -12 ; size = 4
bs  = -8  ; size = 4
cs  = -4  ; size = 4
_f2 PROC
...
_f2 ENDP
```

```
cs  = -12 ; size = 4
bs  = -8  ; size = 4
as  = -4  ; size = 4
_f1 PROC
```
Unlike MSVC 2010, MSVC 2013 allocated a/b/c variables in function f2() in reverse order. And this is completely correct, because C/C++ standards has no rule, in which order local variables must be allocated in the local stack, if at all. The reason of difference is because MSVC 2010 has one way to do it, and MSVC 2013 has supposedly something changed inside of compiler guts, so it behaves slightly different.

1.9.5 Exercises

- http://challenges.re/51
- http://challenges.re/52

1.10 Almost empty function

This is a real piece of code I found in Boolector:

```c
// forward declaration. the function is residing in some other module:
int boolector_main (int argc, char **argv);

// executable
int main (int argc, char **argv)
{
    return boolector_main (argc, argv);
}
```

Why would anyone do so? I don’t know, but my best guess is that boolector_main() may be compiled in some kind of DLL or dynamic library, and be called from a test suite. Surely, a test suite can prepare argc/argv variables as CRT would do it.

Interestingly enough, how this compiles:

**Listing 1.45: Non-optimizing GCC 8.2 x64 (assembly output)**

```
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     DWORD PTR -4[rbp], edi
    mov     QWORD PTR -16[rbp], rsi
    mov     rdx, QWORD PTR -16[rbp]
    mov     eax, DWORD PTR -4[rbp]
    mov     rsi, rdx
    mov     edi, eax
    call    boolector_main
    leave
    ret
```

This is OK, prologue, unnecessary (not optimized) shuffling of two arguments, CALL, epilogue, RET. But let’s see optimizing version:

**Listing 1.46: Optimizing GCC 8.2 x64 (assembly output)**

```
main:
    jmp    boolector_main
```

As simple as that: stack/registers are untouched and boolector_main() has the same arguments set. So all we need to do is pass execution to another address.

This is close to thunk function.

We will see something more advanced later: 1.11.2 on page 53, 1.21.1 on page 154.
1.11 printf() with several arguments

Now let’s extend the Hello, world! (1.5 on page 8) example, replacing printf() in the main() function body with this:

```c
#include <stdio.h>
int main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
    return 0;
}
```

1.11.1 x86

x86: 3 arguments

MSVC

When we compile it with MSVC 2010 Express we get:

```
$SG3830 DB 'a=%d; b=%d; c=%d', 00H
...
push 3
push 2
push 1
push OFFSET $SG3830
call _printf
add esp, 16 ; 00000010H
```

Almost the same, but now we can see the printf() arguments are pushed onto the stack in reverse order. The first argument is pushed last.

By the way, variables of int type in 32-bit environment have 32-bit width, that is 4 bytes.

So, we have 4 arguments here. 4 + 4 = 16 —they occupy exactly 16 bytes in the stack: a 32-bit pointer to a string and 3 numbers of type int.

When the stack pointer (ESP register) has changed back by the ADD ESP, X instruction after a function call, often, the number of function arguments could be deduced by simply dividing X by 4.

Of course, this is specific to the cdecl calling convention, and only for 32-bit environment.

See also the calling conventions section (6.1 on page 725).

In certain cases where several functions return right after one another, the compiler could merge multiple “ADD ESP, X” instructions into one, after the last call:

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

Here is a real-world example:

```
Listing 1.47: x86
.text:100113E7    push   3
.text:100113E9    call   sub_100018B0 ; takes one argument (3)
```
<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>100113EE</td>
<td>call</td>
<td>sub_10001900; takes no arguments at all</td>
<td></td>
</tr>
<tr>
<td>100113F3</td>
<td>call</td>
<td>sub_10006A90; takes no arguments at all</td>
<td></td>
</tr>
<tr>
<td>100113F8</td>
<td>push</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>100113FA</td>
<td>call</td>
<td>sub_100018B0; takes one argument (1)</td>
<td></td>
</tr>
<tr>
<td>100113FF</td>
<td>add</td>
<td>esp, 8; drops two arguments from stack at once</td>
<td></td>
</tr>
</tbody>
</table>
Now let's try to load this example in OllyDbg. It is one of the most popular user-land win32 debuggers. We can compile our example in MSVC 2012 with /MD option, which means to link with MSVCR*.DLL, so we can see the imported functions clearly in the debugger.

Then load the executable in OllyDbg. The very first breakpoint is in ntdll.dll, press F9 (run). The second breakpoint is in CRT-code. Now we have to find the main() function.

Find this code by scrolling the code to the very top (MSVC allocates the main() function at the very beginning of the code section):

![Figure 1.8: OllyDbg: the very start of the main() function](image)

Click on the PUSH EBP instruction, press F2 (set breakpoint) and press F9 (run). We have to perform these actions in order to skip CRT-code, because we aren't really interested in it yet.
Press F8 (step over) 6 times, i.e. skip 6 instructions:

Figure 1.9: OllyDbg: before printf() execution

Now the PC points to the CALL printf instruction. OllyDbg, like other debuggers, highlights the value of the registers which were changed. So each time you press F8, EIP changes and its value is displayed in red. ESP changes as well, because the arguments values are pushed into the stack.

Where are the values in the stack? Take a look at the right bottom debugger window:

Figure 1.10: OllyDbg: stack after the argument values have been pushed (The red rectangular border was added by the author in a graphics editor)

We can see 3 columns there: address in the stack, value in the stack and some additional OllyDbg comments. OllyDbg understands printf()-like strings, so it reports the string here and the 3 values attached to it.

It is possible to right-click on the format string, click on “Follow in dump”, and the format string will appear in the debugger left-bottom window, which always displays some part of the memory. These memory values can be edited. It is possible to change the format string, in which case the result of our example would be different. It is not very useful in this particular case, but it could be good as an exercise so you start building a feel of how everything works here.
Press F8 (step over).

We see the following output in the console:

```
a=1; b=2; c=3
```

Let's see how the registers and stack state have changed:

![Figure 1.11: OllyDbg after printf() execution](image)

Register EAX now contains 0xD (13). That is correct, since printf() returns the number of characters printed. The value of EIP has changed: indeed, now it contains the address of the instruction coming after CALL printf. ECX and EDX values have changed as well. Apparently, the printf() function's hidden machinery used them for its own needs.

A very important fact is that neither the ESP value, nor the stack state have been changed! We clearly see that the format string and corresponding 3 values are still there. This is indeed the cdecl calling convention behavior: callee does not return ESP back to its previous value. The caller is responsible to do so.
Press F8 again to execute ADD ESP, 10 instruction:

Figure 1.12: OllyDbg: after ADD ESP, 10 instruction execution

ESP has changed, but the values are still in the stack! Yes, of course; no one needs to set these values to zeros or something like that. Everything above the stack pointer (SP) is noise or garbage and has no meaning at all. It would be time consuming to clear the unused stack entries anyway, and no one really needs to.

GCC

Now let’s compile the same program in Linux using GCC 4.4.1 and take a look at what we have got in IDA:

```
main proc near
var_10 = dword ptr -10h
var_C = dword ptr -0Ch
var_8 = dword ptr -8
var_4 = dword ptr -4

    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
sub esp, 10h
    mov eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
    mov [esp+10h+var_4], 3
    mov [esp+10h+var_8], 2
    mov [esp+10h+var_C], 1
    mov [esp+10h+var_10], eax
call _printf
    mov eax, 0
leave
retn
main endp
```

It's noticeable that the difference between the MSVC code and the GCC code is only in the way the arguments are stored on the stack. Here the GCC is working directly with the stack without the use of PUSH/POP.

GCC and GDB
Let's try this example also in **GDB** in Linux.

- **g option instructs the compiler to include debug information in the executable file.**

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu ...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 1.48: let's set breakpoint on printf()

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Run. We don’t have the printf() function source code here, so **GDB** can’t show it, but may do so.

```
(gdb) run
Starting program: /home/dennis/polygon/1
Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
29 printf.c: No such file or directory.
```

Print 10 stack elements. The most left column contains addresses on the stack.

```
(gdb) x/10w $esp
0xbffff11c: 0x0804844a 0x080484f0 0x00000001 0x00000002
0xbffff12c: 0x00000003 0x08048460 0x00000000 0x00000000
0xbffff13c: 0xb7e29905 0x00000001
```

The very first element is the **RA** (0x0804844a). We can verify this by disassembling the memory at this address:

```
(gdb) x/5i 0x0804844a
0x804844a <main+45>: mov $0x0,%eax
0x804844f <main+50>: leave
0x8048450 <main+51>: ret
0x8048451: xchg %ax,%ax
0x8048453: xchg %ax,%ax
```

The two **XCHG** instructions are idle instructions, analogous to **NOPs**.

The second element (0x080484f0) is the format string address:

```
(gdb) x/s 0x080484f0
0x080484f0: "a=%d; b=%d; c=%d"
```

Next 3 elements (1, 2, 3) are the printf() arguments. The rest of the elements could be just “garbage” on the stack, but could also be values from other functions, their local variables, etc. We can ignore them for now.

Run “finish”. The command instructs GDB to “execute all instructions until the end of the function”. In this case: execute till the end of printf().

```
(gdb) finish
Run till exit from #0 __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at printf.c:29
main () at 1.c:6
6 return 0;
Value returned is $2 = 13
```

**GDB** shows what printf() returned in EAX (13). This is the number of characters printed out, just like in the OllyDbg example.

We also see “return 0;” and the information that this expression is in the 1.c file at the line 6. Indeed, the 1.c file is located in the current directory, and **GDB** finds the string there. How does **GDB** know which C-code line is being currently executed? This is due to the fact that the compiler, while generating

---

68 GNU Debugger
debugging information, also saves a table of relations between source code line numbers and instruction addresses. GDB is a source-level debugger, after all.

Let's examine the registers. 13 in EAX:

(gdb) info registers
eax  0xd  13
ecx  0x0  0
edx  0x0  0
ebx  0xb7fc0000  -1208221696
esp  0xbfffff120  0xbfffff120
ebp  0xbfffff138  0xbfffff138
esi  0x0  0
edi  0x0  0
eip  0x804844a  0x804844a <main+45>
...

Let's disassemble the current instructions. The arrow points to the instruction to be executed next.

(gdb) disas
Dump of assembler code for function main:
0x0804841d <+0>: push %ebp  
0x0804841e <+1>: mov %esp,%ebp  
0x08048420 <+3>: and $0xfffffffff0,%esp  
0x08048423 <+6>: sub $0x10,%esp  
0x08048426 <+9>: movl $0x3,0xc(%esp)  
0x0804842e <+17>: movl $0x2,0x8(%esp)  
0x08048436 <+25>: mov $0x1,0x4(%esp)  
0x0804843e <+33>: movl $0x80484f0,0(%esp)  
0x08048444 <+40>: call 0x80482f0 <printf@plt>  
0x0804844a <+45>: mov $0x0,eax  
0x0804844f <+50>: leave  
0x08048450 <+51>: ret
End of assembler dump.

GDB uses AT&T syntax by default. But it is possible to switch to Intel syntax:

(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
0x0804841d <+0>: push ebp  
0x0804841e <+1>: mov ebp,esp  
0x08048420 <+3>: and esp,0xfffffffff0  
0x08048423 <+6>: sub esp,0x10  
0x08048426 <+9>: mov DWORD PTR [esp+0xc],0x3  
0x0804842e <+17>: mov DWORD PTR [esp+0x8],0x2  
0x08048436 <+25>: mov DWORD PTR [esp+0x4],0x1  
0x0804843e <+33>: mov DWORD PTR [esp],0x80484f0  
0x08048444 <+40>: call 0x80482f0 <printf@plt>  
0x0804844a <+45>: mov eax,0x0  
0x0804844f <+50>: leave  
0x08048450 <+51>: ret
End of assembler dump.

Execute next instruction. GDB shows ending bracket, meaning, it ends the block.

(gdb) step
7
};

Let's examine the registers after the MOV EAX, 0 instruction execution. Indeed EAX is zero at that point.

(gdb) info registers
eax  0x0  0
ecx  0x0  0
edx  0x0  0
ebx  0xb7fc0000  -1208221696
esp  0xbfffff120  0xbfffff120
ebp  0xbfffff138  0xbfffff138
esi  0x0  0
edi  0x0  0
x64: 8 arguments

To see how other arguments are passed via the stack, let’s change our example again by increasing the number of arguments to 9 (printf() format string + 8 int variables):

```c
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
}
```

MSVC

As it was mentioned earlier, the first 4 arguments has to be passed through the RCX, RDX, R8, R9 registers in Win64, while all the rest—via the stack. That is exactly what we see here. However, the MOV instruction, instead of PUSH, is used for preparing the stack, so the values are stored to the stack in a straightforward manner.

Listing 1.49: MSVC 2012 x64

```assembly
$SG2923 DB 'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d', 0aH, 00H
main PROC sub rsp, 88
    mov DWORD PTR [rsp+64], 8
    mov DWORD PTR [rsp+56], 7
    mov DWORD PTR [rsp+48], 6
    mov DWORD PTR [rsp+40], 5
    mov DWORD PTR [rsp+32], 4
    mov r9d, 3
    mov r8d, 2
    mov edx, 1
    lea rcx, OFFSET FLAT:$SG2923
    call printf
    ; return 0
    xor eax, eax
    add rsp, 88
    ret 0
main ENDP
TEXT ENDS
END
```

The observant reader may ask why are 8 bytes allocated for int values, when 4 is enough? Yes, one has to recall: 8 bytes are allocated for any data type shorter than 64 bits. This is established for the convenience’s sake: it makes it easy to calculate the address of arbitrary argument. Besides, they are all located at aligned memory addresses. It is the same in the 32-bit environments: 4 bytes are reserved for all data types.

GCC

The picture is similar for x86-64 *NIX OS-es, except that the first 6 arguments are passed through the RDI, RSI, RDX, RCX, R8, R9 registers. All the rest—via the stack. GCC generates the code storing the string pointer into EDI instead of RDI—we noted that previously: 1.5.2 on page 15.

We also noted earlier that the EAX register has been cleared before a printf() call: 1.5.2 on page 15.
Listing 1.50: Optimizing GCC 4.4.6 x64

.LC0:
.string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"

main:
    sub    rsp, 40
    mov    r9d, 5
    mov    r8d, 4
    mov    ecx, 3
    mov    edx, 2
    mov    esi, 1
    mov    edi, OFFSET FLAT:.LC0
    xor    eax, eax ; number of vector registers passed
    mov    DWORD PTR [rsp+16], 8
    mov    DWORD PTR [rsp+8], 7
    mov    DWORD PTR [rsp], 6
    call    printf

    ; return 0
    xor    eax, eax
    add    rsp, 40
    ret

GCC + GDB

Let's try this example in GDB.

$ gcc -g 2.c -o 2

$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
...
Reading symbols from /home/dennis/polygon/2...done.

Listing 1.51: let's set the breakpoint to printf(), and run

(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n") at /uni2926/printf.c:29
29 printf.c: No such file or directory.

Registers RSI/RDX/RCX/R8/R9 have the expected values. RIP has the address of the very first instruction of the printf() function.

(gdb) info registers
rax 0x0 0
rbx 0x0 0
rcx 0x3 3
rdx 0x2 2
rsi 0x1 1
rdi 0x400628 4195880
rbp 0x7fffffffdf60 0x7fffffffdf60
rsp 0x7fffffffdf38 0x7fffffffdf38
r8 0x4 4
r9 0x5 5
r10 0x7fffffffdfce0 140737488346336
r11 0x7fffffff7a65f60 140737348263776
r12 0x400440 4195392
r13 0x7fffffff0e40 140737488347200
r14 0x0 0
r15 0x0 0
Let's inspect the format string

```
Listing 1.52: let's inspect the format string
(gdb) x/s $rdi
0x400628: "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
```

Let's dump the stack with the x/g command this time—\(g\) stands for giant words, i.e., 64-bit words.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x0000000000400576 0x0000000000000006
0x7fffffffdf48: 0x0000000000000007 0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000 0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5 0x0000000000000000
0x7fffffffdf78: 0x00007fffffffde08 0x0000000000000000
```

The very first stack element, just like in the previous case, is the RA. 3 values are also passed through the stack: 6, 7, 8. We also see that 8 is passed with the high 32-bits not cleared: 0x00007fff00000008. That's OK, because the values are of \(\text{int}\) type, which is 32-bit. So, the high register or stack element part may contain "random garbage".

If you take a look at where the control will return after the `printf()` execution, GDB will show the entire `main()` function:

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x0000000000400576
Dump of assembler code for function main:
 0x000000000040052d <+0>:  push rbp
 0x000000000040052e <+1>:  mov rbp,rsp
 0x0000000000400531 <+4>:  sub rsp,0x20
 0x0000000000400535 <+8>:  mov DWORD PTR [rsp+0x10],0x8
 0x000000000040053d <+16>: mov DWORD PTR [rsp+0x8],0x7
 0x0000000000400545 <+24>: mov DWORD PTR [rsp],0x6
 0x000000000040054c <+31>: mov r9d,0x5
 0x0000000000400552 <+37>: mov r8d,0x4
 0x0000000000400558 <+43>: mov ecx,0x3
 0x000000000040055d <+48>: mov edx,0x2
 0x0000000000400562 <+53>: mov esi,0x1
 0x0000000000400567 <+58>: mov edi,0x400628
 0x000000000040056c <+63>: mov eax,0x0
 0x0000000000400571 <+68>: call 0x400410 <printf@plt>
 0x0000000000400576 <+73>: mov eax,0x0
 0x000000000040057b <+78>: leave
 0x000000000040057c <+79>: ret
End of assembler dump.
```

Let's finish executing `printf()`, execute the instruction zeroing EAX, and note that the EAX register has a value of exactly zero. RIP now points to the LEAVE instruction, i.e., the penultimate one in the `main()` function.

```
(gdb) finish
Run till exit from #0  _printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n") at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
main () at 2.c:6
6  return 0;
Value returned is $1 = 39
(gdb) next
7  }
(gdb) info registers
rax 0x0 0
rbx 0x0 0
rcx 0x26 38
rdx 0x7fffffffdd59f0 140737351866864
rsi 0x7fffffffde08 2147483609
rdi 0x0 0
rbp 0x7fffffffde06 0x7fffffffde06
rsp 0x7fffffffde040 0x7fffffffde040
r8 0x7fffffffdd26a0 140737351853728
```

52
1.11.2 ARM

ARM: 3 arguments

ARM’s traditional scheme for passing arguments (calling convention) behaves as follows: the first 4 arguments are passed through the R0-R3 registers; the remaining arguments via the stack. This resembles the arguments passing scheme in fastcall (6.1.3 on page 726) or win64 (6.1.5 on page 728).

32-bit ARM

Non-optimizing Keil 6/2013 (ARM mode)

![Listing 1.53: Non-optimizing Keil 6/2013 (ARM mode)](image)

So, the first 4 arguments are passed via the R0-R3 registers in this order: a pointer to the printf() format string in R0, then 1 in R1, 2 in R2 and 3 in R3. The instruction at 0x18 writes 0 to R0—this is return 0 C-statement. There is nothing unusual so far.

Optimizing Keil 6/2013 generates the same code.

Optimizing Keil 6/2013 (Thumb mode)

![Listing 1.54: Optimizing Keil 6/2013 (Thumb mode)](image)

There is no significant difference from the non-optimized code for ARM mode.

Optimizing Keil 6/2013 (ARM mode) + let’s remove return

Let’s rework example slightly by removing return 0:
#include <stdio.h>

void main()
{
    printf("a=%d; b=%d; c=%d\n", 1, 2, 3);
};

The result is somewhat unusual:

---

Listing 1.55: Optimizing Keil 6/2013 (ARM mode)

This is the optimized (-O3) version for ARM mode and this time we see B as the last instruction instead of the familiar BL. Another difference between this optimized version and the previous one (compiled without optimization) is the lack of function prologue and epilogue (instructions preserving the R0 and LR registers values). The B instruction just jumps to another address, without any manipulation of the LR register, similar to JMP in x86. Why does it work? Because this code is, in fact, effectively equivalent to the previous. There are two main reasons: 1) neither the stack nor SP (the stack pointer) is modified; 2) the call to printf() is the last instruction, so there is nothing going on afterwards. On completion, the printf() function simply returns the control to the address stored in LR. Since the LR currently stores the address of the point from where our function has been called then the control from printf() will be returned to that point. Therefore we do not have to save LR because we do not have necessity to modify LR. And we do not have necessity to modify LR because there are no other function calls except printf(). Furthermore, after this call we do not to do anything else! That is the reason such optimization is possible.

This optimization is often used in functions where the last statement is a call to another function. A similar example is presented here: 1.21.1 on page 155.

A somewhat simpler case was described above: 1.10 on page 41.

---

ARM64

Non-optimizing GCC (Linaro) 4.9

Listing 1.56: Non-optimizing GCC (Linaro) 4.9

The first instruction STP (Store Pair) saves FP (X29) and LR (X30) in the stack. The second ADD X29, SP, 0 instruction forms the stack frame. It is just writing the value of SP into X29.

Next, we see the familiar ADRP/ADD instruction pair, which forms a pointer to the string. lo12 meaning low 12 bits, i.e., linker will write low 12 bits of LC1 address into the opcode of ADD instruction. %d in printf() string format is a 32-bit int, so the 1, 2 and 3 are loaded into 32-bit register parts.
Optimizing GCC (Linaro) 4.9 generates the same code.

**ARM: 8 arguments**

Let’s use again the example with 9 arguments from the previous section: 1.11.1 on page 50.

```c
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
}
```

Optimizing Keil 6/2013: ARM mode

This code can be divided into several parts:

- **Function prologue:**
  
The very first `STR LR, [SP,#var_4]` instruction saves LR on the stack, because we are going to use this register for the `printf()` call. Exclamation mark at the end indicates pre-index.

  This implies that SP is to be decreased by 4 first, and then LR will be saved at the address stored in SP. This is similar to PUSH in x86. Read more about it at: 1.39.2 on page 437.

  The second `SUB SP, SP, #0x14` instruction decreases SP (the stack pointer) in order to allocate 0x14 (20) bytes on the stack. Indeed, we have to pass 5 32-bit values via the stack to the `printf()` function, and each one occupies 4 bytes, which is exactly 5 * 4 = 20. The other 4 32-bit values are to be passed through registers.

- **Passing 5, 6, 7 and 8 via the stack:** they are stored in the R0, R1, R2 and R3 registers respectively. Then, the `ADD R12, SP, #0x18+var_14` instruction writes the stack address where these 4 variables are to be stored, into the R12 register. `var_14` is an assembly macro, equal to -0x14, created by IDA to conveniently display the code accessing the stack. The `var_?` macros generated by IDA reflect local variables in the stack.

  So, SP+4 is to be stored into the R12 register.

  The next `STMIA R12, R0-R3` instruction writes registers R0-R3 contents to the memory pointed by R12. `STMIA` abbreviates *Store Multiple Increment After*. *Increment After* implies that R12 is to be increased by 4 after each register value is written.
• Passing 4 via the stack: 4 is stored in R0 and then this value, with the help of the
   STR R0, [SP,#0x18+var_18] instruction is saved on the stack. var_18 is -0x18, so the offset is to
   be 0, thus the value from the R0 register (4) is to be written to the address written in SP.

• Passing 1, 2 and 3 via registers: The values of the first 3 numbers (a, b, c) (1, 2, 3 respectively) are
   passed through the R1, R2 and R3 registers right before the printf() call, and the other 5 values are
   passed via the stack:
   • printf() call.

• Function epilogue:

The ADD SP, SP, #0x14 instruction restores the SP pointer back to its former value, thus annulling
everything what has been stored on the stack. Of course, what has been stored on the stack will
stay there, but it will all be rewritten during the execution of subsequent functions.

The LDR PC, [SP+4+var_4],#4 instruction loads the saved LR value from the stack into the PC reg-
ister, thus causing the function to exit. There is no exclamation mark—indeed, PC is loaded first from
the address stored in SP (4+var_4 = 4+(-4) = 0, so this instruction is analogous to LDR PC, [SP],#4),
and then SP is increased by 4. This is referred as post-index69. Why does IDA display the instruction
like that? Because it wants to illustrate the stack layout and the fact that var_4 is allocated for saving the
LR value in the local stack. This instruction is somewhat similar to POP PC in x8670.

Optimizing Keil 6/2013: Thumb mode

```
.text:0000001C  printf_main2
.text:0000001C  var_18 = -0x18
.text:0000001C  var_14 = -0x14
.text:0000001C  var_8 = -8

.text:0000001C  00 B5  PUSH  {LR}
.text:0000001C  08 23  MOVS  R3, #8
.text:00000020  85 B0  SUB  SP, SP, #0x14
.text:00000022  04 93  STR  R3, [SP,#0x18+var_8]
.text:00000024  07 22  MOVS  R2, #7
.text:00000026  06 21  MOVS  R1, #6
.text:00000028  05 20  MOVS  R0, #5
.text:0000002A  01 AB  ADD  R3, SP, #0x18+var_14
.text:0000002C  07 C3  STMIA  R3!, {R0-R2}
.text:0000002E  04 20  MOVS  R0, #4
.text:00000030  00 90  STR  R0, [SP,#0x18+var_18]
.text:00000032  03 23  MOVS  R3, #3
.text:00000034  02 22  MOVS  R2, #2
.text:00000036  01 21  MOVS  R1, #1
.text:00000038  A0 A0  ADR  R0, aADBDCDDDEDFDGD ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%a"
... .text:0000003A  06 F0 D9 F8  BL  __2printf
.text:0000003E  loc_3E ; CODE XREF: example13_f+16
.text:0000003E  05 B0  ADD  SP, SP, #0x14
.text:00000040  00 BD  POP  {PC}
```

The output is almost like in the previous example. However, this is Thumb code and the values are packed
into stack differently: 8 goes first, then 5, 6, 7, and 4 goes third.

Optimizing Xcode 4.6.3 (LLVM): ARM mode

```
__text:0000290C  __printf_main2
__text:0000290C  var_1C = -0x1C
__text:0000290C  var_C = -0xC
__text:0000290C  __text:0000290C  80 40 2D E9  STMFD  SP!, {R7,LR}
__text:00002910  0D 70 A0 E1  MOV  R7, SP
```

---

69Read more about it: 1.39.2 on page 437.
70It is impossible to set IP/EIP/RIP value using POP in x86, but anyway, you got the analogy right.
Another thing that catches the eye is that the instructions are arranged seemingly random. For example, almost the same as what we have already seen, with the exception of
presumably, the compiler tries to generate code in such a manner (wherever it is possible). It would be possible to do it in one point.

However, the optimizing compiler may have its own reasons on how to order the instructions so to achieve higher efficiency during the execution.

Almost the same as what we have already seen, with the exception of STMFA (Store Multiple Full Ascending) instruction, which is a synonym of STMIW (Store Multiple Increment Before) instruction. This instruction increases the value in the SP register and only then writes the next register value into the memory, rather than performing those two actions in the opposite order.

Another thing that catches the eye is that the instructions are arranged seemingly random. For example, the value in the R0 register is manipulated in three places, at addresses 0x2918, 0x2920 and 0x2928, when it would be possible to do it in one point.

However, the optimizing compiler may have its own reasons on how to order the instructions so to achieve higher efficiency during the execution.

Usually, the processor attempts to simultaneously execute instructions located side-by-side. For example, instructions like MOVT R0, #0 and ADD R0, PC, R0 cannot be executed simultaneously since they both modify the R0 register. On the other hand, MOVT R0, #0 and MOVT R2, #4 instructions can be executed simultaneously since the effects of their execution are not conflicting with each other. Presumably, the compiler tries to generate code in such a manner (wherever it is possible).

Optimizing Xcode 4.6.3 (LLVM): Thumb-2 mode

```
.text:00002914 14 D0 4D E2     SUB   SP, SP, #0x14
.text:00002918 70 05 01 E3     MOV   R0, #0x1570
.text:0000291C 07 C0 A0 E3     MOV   R12, #7
.text:00002920 00 00 40 E3     MOVT  R0, #0
.text:00002924 04 20 A0 E3     MOV   R2, #4
.text:00002928 00 00 8F E0     ADD   R0, PC, R0
.text:0000292C 06 30 A0 E3     MOV   R3, #6
.text:00002930 05 10 A0 E3     MOV   R1, #5
.text:00002934 00 20 B8 E5     STR   R2, [SP,#0x1C+var_1C]
.text:00002938 0A 10 8D E9     STMFA  SP, {R1,R3,R12}
.text:0000293C 08 90 A0 E3     MOV   R9, #8
.text:00002940 01 10 A0 E3     MOV   R1, #1
.text:00002944 02 20 A0 E3     MOV   R2, #2
.text:00002948 03 30 A0 E3     MOV   R3, #3
.text:0000294C 10 90 8D E5     STR   R9, [SP,#0x1C+var_C]
.text:00002950 A4 05 00 E8     BL   _printf
.text:00002954 07 D0 A0 E1     MOV   SP, R7
.text:00002958 80 80 BD E8     LDMFD  SP!, {R7,PC}
```

```
.text:000029A0 _printf_main2
.text:000029A0
.text:000029A0     var_1C = -0x1C
.text:000029A0     var_18 = -0x18
.text:000029A0     var_C = -0xC
.text:000029A0
.text:000029A0 80 B5     PUSH   {R7,LR}
.text:000029A2 6F 46     MOV   R7, SP
.text:000029A4 85 B0     SUB   SP, SP, #0x14
.text:000029A6 41 F2 D8 20     MOVW  R0, #0x12D8
.text:000029AA 4F F0 07 0C     MOVL   R12, #7
.text:000029B2 04 22     MOVS  R2, #4
.text:000029B4 78 44     ADD   R0, PC ; char *
.text:000029B6 06 23     MOVS  R3, #6
.text:000029B8 05 21     MOVS  R1, #5
.text:000029BA 0D F1 04 0E     ADDL  LR, SP, #0x1C+var_18
.text:000029B4 00 92     STR   R2, [SP,#0x1C+var_1C]
.text:000029BC 00 98 00 09     STMIA   LR, {R1,R3,R12}
.text:000029C0 4F F0 08 09     MOVL   R9, #8
.text:000029C4 8E E8 0A 10     STMPIA   LR, {R1,R3,R12}
.text:000029C8 01 21     MOVS  R1, #1
.text:000029CA 02 22     MOVS  R2, #2
.text:000029CC 03 23     MOVS  R3, #3
.text:000029CE CD F8 10 90     STRL   R9, [SP,#0x1C+var_C]
.text:000029D0 01 F0 0A EA     BLX   _printf
.text:000029D4 05 B0     ADD   SP, SP, #0x14
.text:000029D8 80 BD     POP   {R7,PC}
```
The output is almost the same as in the previous example, with the exception that Thumb-instructions are used instead.

**ARM64**

**Non-optimizing GCC (Linaro) 4.9**

Listing 1.57: Non-optimizing GCC (Linaro) 4.9

```assembly
.LC2:
  .string "%a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
.f3:
; grab more space in stack:
  sub  sp, sp, #32
; save FP and LR in stack frame:
  stp  x29, x30, [sp,16]
; set stack frame (FP=SP):
  add  x29, sp, 16
  adrp  x0, .LC2 ; "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n"
  add  x0, x0, :lo12:.LC2
  mov  w1, 8 ; 9th argument
  str  w1, [sp] ; store 9th argument in the stack
  mov  w1, 1
  mov  w2, 2
  mov  w3, 3
  mov  w4, 4
  mov  w5, 5
  mov  w6, 6
  mov  w7, 7
  bl   printf
  sub  sp, x29, #16
; restore FP and LR
  ldp  x29, x30, [sp,16]
  add  sp, sp, 32
ret
```

The first 8 arguments are passed in X- or W-registers: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64), (2013)*][1]. A string pointer requires a 64-bit register, so it’s passed in X0. All other values have a int 32-bit type, so they are stored in the 32-bit part of the registers (W-). The 9th argument (8) is passed via the stack. Indeed: it’s not possible to pass large number of arguments through registers, because the number of registers is limited.

Optimizing GCC (Linaro) 4.9 generates the same code.

### 1.11.3 MIPS

**3 arguments**

**Optimizing GCC 4.4.5**

The main difference with the “Hello, world!” example is that in this case printf() is called instead of puts() and 3 more arguments are passed through the registers $5...$7 (or $A0...$A2). That is why these registers are prefixed with A-, which implies they are used for function arguments passing.

Listing 1.58: Optimizing GCC 4.4.5 (assembly output)

```assembly
.LC0:
  .ascii "a=%d; b=%d; c=%d\000"
main:
; function prologue:
  lui   $28,%hi(__gnu_local_gp)
  addiu $sp,$sp,-32
  addiu $28,$28,%lo(__gnu_local_gp)
  sw    $31,28($sp)
```

[1]: Also available as [http://go.yurichev.com/17287](http://go.yurichev.com/17287)
Listing 1.59: Optimizing GCC 4.4.5 (IDA)

IDA has coalesced pair of LUI and ADDIU instructions into one LA pseudo instruction. That's why there are no instruction at address 0x1C: because LA occupies 8 bytes.

Non-optimizing GCC 4.4.5

Non-optimizing GCC is more verbose:

Listing 1.60: Non-optimizing GCC 4.4.5 (assembly output)

$LC0:
    .ascii "a=%d; b=%d; c=%d\000"
main:
    ; function prologue:
    addiu $sp,$sp,-32 ; branch delay slot
Listing 1.61: Non-optimizing GCC 4.4.5 (IDA)

```
sw $31,28($sp)
sw $fp,24($sp)
move $fp,$sp
lui $28,%hi(_gnu_local_gp)
addiu $28,$28,%lo(_gnu_local_gp)

; load address of the text string:
lui $2,%hi($LC0)
addiu $2,$2,%lo($LC0)

; set 1st argument of printf():
move $4,$2

; set 2nd argument of printf():
li $5,1 # 0x1

; set 3rd argument of printf():
li $6,2 # 0x2

; set 4th argument of printf():
li $7,3 # 0x3

; get address of printf():
lw $2,%call16(printf)(28)
nop

; call printf():
mov $25,$2
jalr $25
nop

; function epilogue:
lw $28,16($fp)

Listing 1.61: Non-optimizing GCC 4.4.5 (IDA)

.text:00000000 main:
.text:00000000
.text:00000000 var_10 = -0x10
.text:00000000 var_8 = -8
.text:00000000 var_4 = -4
.text:00000000

; function prologue:
.addi $sp, -0x20
.sw $ra, 0x20+var_4($sp)
.sw $fp, 0x20+var_8($sp)
.move $fp, $sp
.la $gp, _gnu_local_gp
.sw $gp, 0x20+var_10($sp)

; load address of the text string:
.la $v0, aADBDCD # "a=%d; b=%d; c=%d"

; set 1st argument of printf():
.move $a0, $v0

; set 2nd argument of printf():
.li $a1, 1

; set 3rd argument of printf():
.li $a2, 2

; set 4th argument of printf():
.li $a3, 3

; get address of printf():
lw $v0, (printf & 0xFFFF)($gp)
or $at, $zero

; call printf():
mov $t9, $v0
.jalr $t9
.or $at, $zero ; NOP

; function epilogue:
lw $gp, 0x20+var_10($fp)
```
8 arguments

Let's use again the example with 9 arguments from the previous section: 1.11.1 on page 50.

```c
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n", 1, 2, 3, 4, 5, 6, 7, 8);
    return 0;
}
```

Optimizing GCC 4.4.5

Only the first 4 arguments are passed in the $A0 ...$A3 registers, the rest are passed via the stack. This is the O32 calling convention (which is the most common one in the MIPS world). Other calling conventions (like N32) may use the registers for different purposes.

SW abbreviates “Store Word” (from register to memory). MIPS lacks instructions for storing a value into memory, so an instruction pair has to be used instead (LI/SW).

Listing 1.62: Optimizing GCC 4.4.5 (assembly output)

```
$LC0:
   .ascii  "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"

main:
    ; function prologue:
    lui    $28,hi(__gnu_local_gp)
    addiu  $sp,$sp,-56
    addiu  $28,$28,lo(__gnu_local_gp)
    sw     $31,52($sp)
    ; pass 5th argument in stack:
    li     $2,4
    sw     $2,16($sp)
    ; pass 6th argument in stack:
    li     $2,5
    sw     $2,20($sp)
    ; pass 7th argument in stack:
    li     $2,6
    sw     $2,24($sp)
    ; pass 8th argument in stack:
    li     $2,7
    lw     $25,call16(printf)(0x28)
    sw     $2,28($sp)
    ; pass 1st argument in $a0:
    lui    $4,hi($LC0)
    ; pass 9th argument in stack:
    li     $2,8
    sw     $2,32($sp)
    addiu  $4,$4,lo($LC0)
    ; pass 2nd argument in $a1:
    li     $5,1
    ; pass 3rd argument in $a2:
    li     $6,2
    ; call printf():
    jalr   $25
    ; pass 4th argument in $a3 (branch delay slot):
```
Listing 1.63: Optimizing GCC 4.4.5 (IDA)

Non-optimizing GCC 4.4.5

Non-optimizing GCC is more verbose:
Listing 1.64: Non-optimizing GCC 4.4.5 (assembly output)

```
$LC0:
.ascii "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\012\000"

main:
    ; function prologue:
    addiu $sp,$sp,-56
    sw $31,52($sp)
    sw $fp,48($sp)
    move $fp,$sp
    lui $28,%hi(_gnu_local_gp)
    addiu $28,$28,%lo(_gnu_local_gp)
    lui $2,%hi($LC0)
    addiu $2,$2,%lo($LC0)
    ; pass 5th argument in stack:
    li $3,4
    sw $3,16($sp)
    ; pass 6th argument in stack:
    li $3,5
    sw $3,20($sp)
    ; pass 7th argument in stack:
    li $3,6
    sw $3,24($sp)
    ; pass 8th argument in stack:
    li $3,7
    sw $3,28($sp)
    ; pass 9th argument in stack:
    li $3,8
    sw $3,32($sp)
    ; pass 1st argument in $a0:
    move $4,$2
    ; pass 2nd argument in $a1:
    li $5,1
    ; pass 3rd argument in $a2:
    li $6,2
    ; pass 4th argument in $a3:
    li $7,3
    ; call printf():
    lw $2,%call16.printf($28)
    nop
    move $25,$2
    jalr $25
    nop
    ; function epilogue:
    lw $28,40($fp)
    ; set return value to 0:
    move $2,$0
    move $sp,$fp
    lw $31,52($sp)
    lw $fp,48($sp)
    addiu $sp,$sp,56
    ; return
    j $31
    nop
```

Listing 1.65: Non-optimizing GCC 4.4.5 (IDA)

```
.text:00000000 main:
.text:00000000
.text:00000000 var_28 = -0x28
.text:00000000 var_24 = -0x24
.text:00000000 var_20 = -0x20
.text:00000000 var_1C = -0x1C
.text:00000000 var_18 = -0x18
.text:00000000 var_10 = -0x10
.text:00000000 var_8 = -8
.text:00000000 var_4 = -4
.text:00000000
    ; function prologue:
.text:00000000 addiu $sp, -0x38
.text:00000004 sw $ra, 0x38+var_4($sp)
```

63
1.11.4 Conclusion

Here is a rough skeleton of the function call:

Listing 1.66: x86

...  
PUSH 3rd argument  
PUSH 2nd argument  
PUSH 1st argument  
CALL function  
; modify stack pointer (if needed)

Listing 1.67: x64 (MSVC)

MOV RCX, 1st argument  
MOV RDX, 2nd argument  
MOV R8, 3rd argument  
MOV R9, 4th argument  
...  
PUSH 5th, 6th argument, etc. (if needed)  
CALL function  
; modify stack pointer (if needed)
By the way, this difference between the arguments passing in x86, x64, fastcall, ARM and MIPS is a good illustration of the fact that the CPU is oblivious to how the arguments are passed to functions. It is also possible to create a hypothetical compiler able to pass arguments via a special structure without using stack at all.

MIPS $A0 …$A3 registers are labeled this way only for convenience (that is in the O32 calling convention). Programmers may use any other register (well, maybe except $ZERO) to pass data or use any other calling convention.

The CPU is not aware of calling conventions whatsoever.

We may also recall how new coming assembly language programmers passing arguments into other functions: usually via registers, without any explicit order, or even via global variables. Of course, it works fine.

1.12 scanf()

Now let’s use scanf().
1.12.1 Simple example

```c
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");
    scanf ("%d", &x);
    printf ("You entered %d...\n", x);
    return 0;
}
```

It’s not clever to use `scanf()` for user interactions nowadays. But we can, however, illustrate passing a pointer to a variable of type `int`.

**About pointers**

Pointers are one of the fundamental concepts in computer science. Often, passing a large array, structure or object as an argument to another function is too expensive, while passing their address is much cheaper. For example, if you are going to print a text string to console, it’s much easier to pass its address into OS kernel.

In addition if the callee function needs to modify something in the large array or structure received as a parameter and return back the entire structure then the situation is close to absurd. So the simplest thing to do is to pass the address of the array or structure to the callee function, and let it change what needs to be changed.

A pointer in C/C++—is simply an address of some memory location.

In x86, the address is represented as a 32-bit number (i.e., it occupies 4 bytes), while in x86-64 it is a 64-bit number (occupying 8 bytes). By the way, that is the reason behind some people’s indignation related to switching to x86-64—all pointers in the x64-architecture require twice as much space, including cache memory, which is “expensive” memory.

It is possible to work with untyped pointers only, given some effort; e.g. the standard C function `memcpy()` that copies a block from one memory location to another, takes 2 pointers of type `void*` as arguments, since it is impossible to predict the type of the data you would like to copy. Data types are not important, only the block size matters.

Pointers are also widely used when a function needs to return more than one value (we are going to get back to this later (3.23 on page 596)).

`scanf()` function—is such a case.

Besides the fact that the function needs to indicate how many values were successfully read, it also needs to return all these values.

In C/C++ the pointer type is only needed for compile-time type checking.

Internally, in the compiled code there is no information about pointer types at all.

**x86**

**MSVC**

Here is what we get after compiling with MSVC 2010:

```
CONST   SEGMENT
$SG3831  DB  'Enter X:\', 0aH, 00H
$SG3832  DB  ' %d\', 00H
$SG3833  DB  'You entered %d...\', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
```
x is a local variable.

According to the C/C++ standard it must be visible only in this function and not from any other external scope. Traditionally, local variables are stored on the stack. There are probably other ways to allocate them, but in x86 that is the way it is.

The goal of the instruction following the function prologue, PUSH ECX, is not to save the ECX state (notice the absence of corresponding POP ECX at the function’s end).

In fact it allocates 4 bytes on the stack for storing the x variable.

x is to be accessed with the assistance of the _x$ macro (it equals to -4) and the EBP register pointing to the current frame.

Over the span of the function’s execution, EBP is pointing to the current stack frame making it possible to access local variables and function arguments via EBP+offset.

It is also possible to use ESP for the same purpose, although that is not very convenient since it changes frequently. The value of the EBP could be perceived as a frozen state of the value in ESP at the start of the function’s execution.

Here is a typical stack frame layout in 32-bit environment:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBP-8</td>
<td>local variable #2, marked in IDA as var_8</td>
</tr>
<tr>
<td>EBP-4</td>
<td>local variable #1, marked in IDA as var_4</td>
</tr>
<tr>
<td>EBP</td>
<td>saved value of EBP</td>
</tr>
<tr>
<td>EBP+4</td>
<td>return address</td>
</tr>
<tr>
<td>EBP+8</td>
<td>argument#1, marked in IDA as arg_0</td>
</tr>
<tr>
<td>EBP+0xC</td>
<td>argument#2, marked in IDA as arg_4</td>
</tr>
<tr>
<td>EBP+0x10</td>
<td>argument#3, marked in IDA as arg_8</td>
</tr>
</tbody>
</table>

The scanf() function in our example has two arguments.

The first one is a pointer to the string containing %d and the second is the address of the x variable.

First, the x variable’s address is loaded into the EAX register by the lea eax, DWORD PTR _x$[ebp] instruction.

LEA stands for load effective address, and is often used for forming an address (.1.6 on page 1014).

We could say that in this case LEA simply stores the sum of the EBP register value and the _x$ macro in the EAX register.

This is the same as lea eax, [ebp-4].
So, 4 is being subtracted from the EBP register value and the result is loaded in the EAX register. Next the EAX register value is pushed into the stack and scanf() is being called.

printf() is being called after that with its first argument — a pointer to the string: You entered %d...

The second argument is prepared with: mov ecx, [ebp-4]. The instruction stores the x variable value and not its address, in the ECX register.

Next the value in the ECX is stored on the stack and the last printf() is being called.
MSVC + OllyDbg

Let’s try this example in OllyDbg. Let’s load it and keep pressing F8 (step over) until we reach our executable file instead of ntdll.dll. Scroll up until main() appears.

Click on the first instruction (PUSH EBP), press F2 (set a breakpoint), then F9 (Run). The breakpoint will be triggered when main() begins.

Let’s trace to the point where the address of the variable x is calculated:

![OllyDbg screenshot](image)

Figure 1.13: OllyDbg: The address of the local variable is calculated

Right-click the EAX in the registers window and then select “Follow in stack”.

This address will appear in the stack window. The red arrow has been added, pointing to the variable in the local stack. At that moment this location contains some garbage (0x6E494714). Now with the help of PUSH instruction the address of this stack element is going to be stored to the same stack on the next position. Let’s trace with F8 until the scanf() execution completes. During the scanf() execution, we input, for example, 123, in the console window:

```
Enter X: 123
```

```
scanf() completed its execution already:

Figure 1.14: OllyDbg: scanf() executed

scanf() returns 1 in EAX, which implies that it has read successfully one value. If we look again at the stack element corresponding to the local variable it now contains 0x7B (123).
Later this value is copied from the stack to the ECX register and passed to printf():

![Image of OllyDbg debugging interface]

**Figure 1.15: OllyDbg: preparing the value for passing to printf()**

**GCC**

Let's try to compile this code in GCC 4.4.1 under Linux:

```assembly
main proc near
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_4 = dword ptr -4

push ebp
mov ebp, esp
and esp, 0FFFFFFFF0h
sub esp, 20h
mov [esp+20h+var_20], offset aEnterX ; "Enter X:"
call _puts
mov eax, offset aD ; "%d"
lea edx, [esp+20h+var_4]
mov [esp+20h+var_1C], edx
mov [esp+20h+var_20], eax
call __isoc99_scanf
mov edx, [esp+20h+var_4]
mov eax, offset aYouEnteredD___ ; "You entered %d...n"
mov [esp+20h+var_1C], edx
mov [esp+20h+var_20], eax
call _printf
mov eax, 0
leave
retn

main endp
```

GCC has replaced the printf() call with call to puts(). The reason for this was explained in (1.5.3 on page 20).

As in the MSVC example—the arguments are placed on the stack using the MOV instruction.

**By the way**
This simple example is a demonstration of the fact that the compiler translates a list of expressions in C/C++-block into a sequential list of instructions. There is nothing in between expressions in C/C++, and so in the resulting machine code, there are nothing between, the control flow slips from one expression to the next one.

x64

The picture here is similar with the difference that the registers, rather than the stack, are used for arguments passing.

MSVC

Listing 1.72: MSVC 2012 x64

```
_DATA SEGMENT
$SG1289 DB 'Enter X:', 0AH, 00H
$SG1291 DB '%d', 00H
$SG1292 DB 'You entered %d...', 0AH, 00H
_DATA ENDS

.TEXTEXT SEGMENT
x$ = 32
main PROC
$LN3:
  sub rsip, 56
  lea rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
  call printf
  lea rdx, QWORD PTR x$[rsp]
  lea rcx, OFFSET FLAT:$SG1291 ; '%d'
  call scanf
  mov edx, DWORD PTR x$[rsp]
  lea rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
  call printf

; return 0
xor eax, eax
ret 0
main ENDP
_TEXTEXT ENDS
```

GCC

Listing 1.73: Optimizing GCC 4.4.6 x64

```
.LC0: .string "Enter X:"
.LC1: .string "%d"
.LC2: .string "You entered %d...\n"

main:
  sub    rsp, 24
  mov    edi, OFFSET FLAT:.LC0 ; "Enter X:"
  call    puts
  lea    rsi, [rsp+12]
  mov    edi, OFFSET FLAT:.LC1 ; "%d"
  xor    eax, eax
  call    __isoc99_scanf
  mov    esi, DWORD PTR [rsp+12]
  mov    edi, OFFSET FLAT:.LC2 ; "You entered %d...
  xor    eax, eax
  call    printf

; return 0
```
In order for scanf() to be able to read item it needs a parameter—pointer to an int. int is 32-bit, so we need 4 bytes to store it somewhere in memory, and it fits exactly in a 32-bit register. A place for the local variable x is allocated in the stack and IDA has named it var_8. It is not necessary, however, to allocate a such since SP (stack pointer) is already pointing to that space and it can be used directly. So, SP’s value is copied to the R1 register and, together with the format-string, passed to scanf().

PUSH/POP instructions behaves differently in ARM than in x86 (it’s the other way around). They are synonyms to STM/STMD/LDM/LDMDIA instructions. And PUSH instruction first writes a value into the stack, and then subtracts SP by 4. POP first adds 4 to SP, and then reads a value from the stack. Hence, after PUSH, SP points to an unused space in stack. It is used by scanf(), and by printf() after.

LDMIA means Load Multiple Registers Increment address After each transfer. STMDB means Store Multiple Registers Decrement address Before each transfer.

Later, with the help of the LDR instruction, this value is moved from the stack to the R1 register in order to be passed to printf().

**ARM64**

Listing 1.74: Non-optimizing GCC 4.9.1 ARM64

```
.LC0:
.string "Enter X:"
.LC1:
.string "%d"
.LC2:
.string "You entered %d...\n"

scann_main:
    ; subtract 32 from SP, then save FP and LR in stack frame:
    stp    x29, x30, [sp, -32]!
    ; set stack frame (FP=SP)
    add    x29, sp, 0
    ; load pointer to the "Enter X:" string:
    adrp   x0, .LC0
    add    x0, x0, :lo12:.LC0
    ; X0=pointer to the "Enter X:" string
    ; print it:
    bl     puts
    ; load pointer to the "%d" string:
    adrp   x0, .LC1
    add    x0, x0, :lo12:.LC1
    ; find a space in stack frame for "x" variable (X1=FP+28):
```

73
add x1, x29, 28
; X1=address of "x" variable
; pass the address to scanf() and call it:
bl __isoc99_scanf
; load 32-bit value from the variable in stack frame:
ldr w1, [x29,28]
; W1=x
; load pointer to the "You entered %d...\n" string
; printf() will take text string from X0 and "x" variable from X1 (or W1)
adrp x0, .LC2
add x0, x0, :lo12:.LC2
bl printf
; return 0
mov w0, 0
; restore FP and LR, then add 32 to SP:
ldp x29, x30, [sp], 32
ret

There is 32 bytes are allocated for stack frame, which is bigger than it needed. Perhaps some memory aligning issue? The most interesting part is finding space for the x variable in the stack frame (line 22). Why 28? Somehow, compiler decided to place this variable at the end of stack frame instead of beginning. The address is passed to scanf(), which just stores the user input value in the memory at that address. This is 32-bit value of type int. The value is fetched at line 27 and then passed to printf().

MIPS

A place in the local stack is allocated for the x variable, and it is to be referred as $sp + 24.

Its address is passed to scanf(), and the user input values is loaded using the LW ("Load Word") instruction and then passed to printf().

Listing 1.75: Optimizing GCC 4.4.5 (assembly output)

$LC0: .ascii "Enter X:\000"
$LC1: .ascii "%d\000"
$LC2: .ascii "You entered %d...\012\000"
main:
; function prologue:
lui $28,hi(__gnu_local_gp)
addiu $sp,$sp,-40
addiu $28,$28,lo(__gnu_local_gp)
sw $31,36($sp)
; call puts():
lw $25,call16(puts)($28)
lui $4,hi($LC0)
jalr $25
addiu $4,$4,lo($LC0) ; branch delay slot
; call scanf():
lw $28,16($sp)
lui $4,hi($LC1)
lw $25,call16(__isoc99_scanf)($28)
; set 2nd argument of scanf(), $a1=$sp+24:
addiu $5,$sp,24
jalr $25
addiu $4,$4,lo($LC1) ; branch delay slot
; call printf():
lw $28,16($sp)
; set 2nd argument of printf(),
; load word at address $sp+24:
lw $5,24($sp)
lw $25,call16(printf)($28)
lui $4,hi($LC2)
jalr $25
addiu $4,$4,lo($LC2) ; branch delay slot
; function epilogue:
1.12.2 The classic mistake

It's a very popular mistake (and/or typo) to pass a value of x instead of pointer to x:

```c
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");
    scanf ("%d", &x); // BUG
    printf ("You entered %d...\n", x);
    return 0;
}
```
So what happens here? x is not initialized and contains some random noise from local stack. When `scanf()` called, it takes string from user, parses it into number and tries to write it into x, treating it as an address in memory. But there is a random noise, so `scanf()` will try to write at random address. Most likely, the process will crash.

Interestingly enough, some CRT libraries in debug build, put visually distinctive patterns into memory just allocated, like 0xCCCCCCCC or 0xBADF00D and so on. In this case, x may contain 0xCCCCCCCC, and `scanf()` would try to write at address 0xCCCCCCCC. And if you’ll notice that something in your process tries to write at address 0xCCCCCCCC, you’ll know that uninitialized variable (or pointer) gets used without prior initialization. This is better than as if newly allocated memory is just cleared by zero bytes.

### 1.12.3 Global variables

What if the x variable from the previous example isn’t local but a global one? Then it would have been accessible from any point, not only from the function body. Global variables are considered **anti-pattern**, but for the sake of the experiment, we could do this.

```c
#include <stdio.h>

// now x is global variable
int x;

int main()
{
    printf("Enter X:\n");
    scanf("%d", &x);
    printf("You entered %d...
", x);
    return 0;
}
```

**MSVC: x86**

```
_DATA SEGMENT
COMM_x:DWORD
$SG2456 DB 'Enter X:', 0AH, 00H
$SG2457 DB '%d', 00H
$SG2458 DB 'You entered %d...', 0AH, 00H
_DATA ENDS
PUBLIC _main
EXTRN _scanf:PROC
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_main PROC
    push ebp
    mov ebp, esp
    push OFFSET $SG2456
    call _printf
    add esp, 4
    push OFFSET _x
    push OFFSET $SG2457
    call _scanf
    add esp, 8
    mov eax, DWORD PTR _x
    push eax
    push OFFSET $SG2458
    call _printf
    add esp, 8
    xor eax, eax
    pop ebp
    ret 0
_main ENDP
_TEXT ENDS
```
In this case the variable is defined in the _DATA segment and no memory is allocated in the local stack. It is accessed directly, not through the stack. Uninitialized global variables take no space in the executable file (indeed, why one needs to allocate space for variables initially set to zero?), but when someone accesses their address, the OS will allocate a block of zeros there. Now let's explicitly assign a value to the variable:

```c
int x=10; // default value
```

We got:

```
_DATA SEGMENT
_x DD 0AH
...
```

Here we see a value 0xA of DWORD type (DD stands for DWORD = 32 bit) for this variable. If you open the compiled .exe in IDA, you can see the variable placed at the beginning of the _DATA segment, and after it you can see text strings.

If you open the compiled .exe from the previous example in IDA, where the value of `x` hasn’t been set, you would see something like this:

```
Listing 1.77: IDA
```

```c
.data:0040FA80 _x dd ? ; DATA XREF: _main+10
.data:0040FA80
.data:0040FA84 dword_40FA84 dd ? ; DATA XREF: _memset+1E
.data:0040FA84 ; unknown libname 1+28
.data:0040FA88 dword_40FA88 dd ? ; DATA XREF: _sbh_find_block+5
.data:0040FA88 ; _sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem dd ? ; DATA XREF: _sbh_find_block+B
.data:0040FA8C ; _sbh_free_block+2CA
.data:0040FA90 dword_40FA90 dd ? ; DATA XREF: _V6_HeapAlloc+13
.data:0040FA90 ; __calloc_impl+72
.data:0040FA94 dword_40FA94 dd ? ; DATA XREF: _sbh_free_block+2FE
```

`x` is marked with ? with the rest of the variables that do not need to be initialized. This implies that after loading the .exe to the memory, a space for all these variables is to be allocated and filled with zeros [ISO/IEC 9899:TC3 (C C99 standard), (2007)6.7.8p10]. But in the .exe file these uninitialized variables do not occupy anything. This is convenient for large arrays, for example.

72That is how a VM behaves
Things are even simpler here:

The variable is located in the data segment. After the PUSH instruction (pushing the address of x) gets executed, the address appears in the stack window. Right-click on that row and select “Follow in dump”. The variable will appear in the memory window on the left. After we have entered 123 in the console, 0x7B appears in the memory window (see the highlighted screenshot regions).

But why is the first byte 7B? Thinking logically, 00 00 00 7B must be there. The cause for this is referred as endianness, and x86 uses little-endian. This implies that the lowest byte is written first, and the highest written last. Read more about it at: 2.8 on page 460. Back to the example, the 32-bit value is loaded from this memory address into EAX and passed to printf().

The memory address of x is 0x00C53394.
In OllyDbg we can review the process memory map (Alt-M) and we can see that this address is inside the .data PE-segment of our program:

![OllyDbg: process memory map](image)

**Figure 1.17: OllyDbg: process memory map**

**GCC: x86**

The picture in Linux is near the same, with the difference that the uninitialized variables are located in the _bss segment. In ELF\(^\text{73}\) file this segment has the following attributes:

- Segment type: Uninitialized
- Segment permissions: Read/Write

If you, however, initialize the variable with some value e.g. 10, it is to be placed in the _data segment, which has the following attributes:

- Segment type: Pure data
- Segment permissions: Read/Write

**MSVC: x64**

```
Listing 1.78: MSVC 2012 x64

DATA SEGMENT
COMM x:DWORD
$SG2924 DB 'Enter X:', 0AH, 00H
$SG2925 DB '%d', 00H
$SG2926 DB 'You entered %d...', 0AH, 00H
_DATA ENDS

_TEXT SEGMENT
main PROC
```

\(^{73}\) Executable File format widely used in *NIX systems including Linux
The code is almost the same as in x86. Please note that the address of the $x$ variable is passed to `scanf()` using a LEA instruction, while the variable's value is passed to the second `printf()` using a MOV instruction. `DWORD PTR`—is a part of the assembly language (no relation to the machine code), indicating that the variable data size is 32-bit and the MOV instruction has to be encoded accordingly.

### ARM: Optimizing Keil 6/2013 (Thumb mode)

So, the $x$ variable is now global and for this reason located in another segment, namely the data segment (.data). One could ask, why are the text strings located in the code segment (.text) and $x$ is located right here? Because it is a variable and by definition its value could change. Moreover it could possibly change
often. While text strings has constant type, they will not be changed, so they are located in the .text segment.

The code segment might sometimes be located in a ROM\textsuperscript{74} chip (keep in mind, we now deal with embedded microelectronics, and memory scarcity is common here), and changeable variables—in RAM.

It is not very economical to store constant variables in RAM when you have ROM.

Furthermore, constant variables in RAM must be initialized, because after powering on, the RAM, obviously, contains random information.

Moving forward, we see a pointer to the x (off\_2C) variable in the code segment, and that all operations with the variable occur via this pointer.

That is because the x variable could be located somewhere far from this particular code fragment, so its address must be saved somewhere in close proximity to the code.

The LDR instruction in Thumb mode can only address variables in a range of 1020 bytes from its location, and in ARM-mode—variables in range of ±4095 bytes.

And so the address of the x variable must be located somewhere in close proximity, because there is no guarantee that the linker would be able to accommodate the variable somewhere nearby the code, it may well be even in an external memory chip!

One more thing: if a variable is declared as const, the Keil compiler allocates it in the .constdata segment. Perhaps thereafter, the linker could place this segment in ROM too, along with the code segment.

\textbf{ARM64}

\begin{verbatim}
1 .comm x,4,4
2 .LC0: .string "Enter X:"
3 .LC1: .string ">%d"
4 .LC2: .string "You entered %d...\n"
5 f5:
6 ; save FP and LR in stack frame:
7   stp  x29, x30, [sp, -16]!
8 ; set stack frame (FP=SP)
9   add  x29, sp, 0
10 ; load pointer to the "Enter X:" string:
11   adrp x0, .LC0
12   add  x0, x0, :lo12:.LC0
13   bl   puts
14 ; load pointer to the "%d" string:
15   adrp x0, .LC1
16   add  x0, x0, :lo12:.LC1
17 ; form address of x global variable:
18   adrp x1, x
19   add  x1, x1, :lo12:x
20   bl   __isoc99_scanf
21 ; form address of x global variable again:
22   adrp x0, x
23   add  x0, x0, :lo12:x
24 ; load value from memory at this address:
25   ldr  w1, [x0]
26 ; load pointer to the "You entered %d...\n" string:
27   adrp x0, .LC2
28   add  x0, x0, :lo12:.LC2
29   bl   printf
30 ; return 0
31   mov  w0, 0
32 ; restore FP and LR:
33   ldp  x29, x30, [sp], 16
34 ret
\end{verbatim}

\textsuperscript{74}Read-Only Memory
In this case the \( x \) variable is declared as global and its address is calculated using the ADRP/ADD instruction pair (lines 21 and 25).

**MIPS**

**Uninitialized global variable**

So now the \( x \) variable is global. Let’s compile to executable file rather than object file and load it into IDA. IDA displays the \( x \) variable in the .sbss ELF section (remember the “Global Pointer”? 1.5.4 on page 24), since the variable is not initialized at the start.

**Listing 1.81: Optimizing GCC 4.4.5 (IDA)**

```plaintext
.text:004006C0 main:
.text:004006C0 .text:004006C0 var_10 = -0x10
.text:004006C0 var_4 = -4
.text:004006C0
; function prologue:
.text:004006C0 lui $gp, 0x42
.text:004006C4 addiu $sp, -0x20
.text:004006C8 li $gp, 0x418940
.text:004006D0 sw $gp, 0x20+var_4($sp)
; call puts():
.text:004006D4 la $t9, puts
.text:004006D8 lui $a0, 0x40
.text:004006DC jalr $t9 ; puts
.text:004006E0 la $a0, aEnterX # "Enter X:" ; branch delay slot
; call scanf():
.text:004006E4 lw $gp, 0x20+var_10($sp)
.text:004006E8 lui $a0, 0x40
.text:004006EC la $t9, __isoc99_scn
; prepare address of x:
.text:004006F0 la $a1, x
.text:004006F4 jalr $t9 ; __isoc99_scn
.text:004006F8 la $a0, aD # "%d" ; branch delay slot
; call printf():
.text:004006FC lw $gp, 0x20+var_10($sp)
.text:00400700 lui $a0, 0x40
; get address of x:
.text:00400704 la $v0, x
.text:00400708 la $t9, printf
; load value from "x" variable and pass it to printf() in $a1:
.text:0040070C lw $a1, (x - 0x41099C)($v0)
.text:00400710 jalr $t9 ; printf
.text:00400714 la $a0, aYouEnteredD___ # "You entered %d...
; branch delay slot
; function epilogue:
.text:00400718 lw $ra, 0x20+var_4($sp)
.text:0040071C move $v0, $zero
.text:00400720 jr $ra
.text:00400724 addiu $sp, 0x20 ; branch delay slot

... .sbss:0041099C # Segment type: Uninitialized .sbss:0041099C .sbss .sbss:0041099C .globl x .sbss:0041099C x: .space 4 .sbss:0041099C
```

IDA reduces the amount of information, so we’ll also do a listing using objdump and comment it:

**Listing 1.82: Optimizing GCC 4.4.5 (objdump)**

```
004006c0 <main>:
 ; function prologue:
4006c0: 3c1c0042 lui gp,0x42
4006c4: 27bdffe0 addiu sp,sp,-32
```
Now we see the $x$ variable address is read from a 64KiB data buffer using GP and adding negative offset to it (line 18). More than that, the addresses of the three external functions which are used in our example (puts(), scanf(), printf()), are also read from the 64KiB global data buffer using GP (lines 9, 16 and 26). GP points to the middle of the buffer, and such offset suggests that all three function’s addresses, and also the address of the $x$ variable, are all stored somewhere at the beginning of that buffer. That make sense, because our example is tiny.

Another thing worth mentioning is that the function ends with two NOPs (MOVE $AT,$AT — an idle instruction), in order to align next function’s start on 16-byte boundary.

**Initialized global variable**

Let’s alter our example by giving the $x$ variable a default value:

```c
int x=10; // default value
```

Now IDA shows that the $x$ variable is residing in the `.data` section:

```
Listing 1.83: Optimizing GCC 4.4.5 (IDA)
```
Why not .sdata? Perhaps that this depends on some GCC option?

Nevertheless, now \textit{x} is in .data, which is a general memory area, and we can take a look how to work with variables there.

The variable's address must be formed using a pair of instructions.

In our case those are LUI (“Load Upper Immediate”) and ADDIU (“Add Immediate Unsigned Word”).

Here is also the objdump listing for close inspection:

\textbf{Listing 1.84: Optimizing GCC 4.4.5 (objdump)}

\begin{verbatim}
004006a0 <main>:
 4006a0:    3c1c0042  lui  gp,0x42
 4006a4:    27bdffe0  addiu  sp,sp,-32
 4006a8:    279c8930  addiu  gp,gp,-30416
 4006ac:    afbf001c  sw  ra,28(sp)
 4006b0:    afb00018  sw  s0,24(sp)
 4006b4:    afbc0010  sw  gp,16(sp)
 4006b8:    8f998034  lw  t9,-32716(gp)
 4006bc:    3c040040  lui  a0,0x40
 4006c0:    0320f809  jalr  t9
 4006c4:    248408d0  addiu  a0,a0,2256
 4006c8:    8fbc0010  lw  gp,16(sp)
 ; prepare high part of x address:
 4006cc:    3c1c0041  lui  s0,0x41
 4006d0:    8f998038  lw  t9,-32712(gp)
 4006d4:    3c040040  lui  a0,0x40
 ; add low part of x address:
 4006d8:    26050920  addiu  a1,s0,2336
 ; now address of x is in $a1.
 4006dc:    0320f809  jalr  t9
 4006e0:    248408dc  addiu  a0,a0,2268
 4006e4:    8fbc0010  lw  gp,16(sp)
 ; high part of x address is still in $s0.
 ; add low part to it and load a word from memory:
 4006e8:    8e050920  lw  a1,2336(s0)
 ; value of x is now in $a1.
 4006ec:    8f99803c  lw  t9,-32708(gp)
 4006f0:    3c040040  lui  a0,0x40
\end{verbatim}
We see that the address is formed using `LUI` and `ADDIU`, but the high part of address is still in the $S0 register, and it is possible to encode the offset in a `LW` ("Load Word") instruction, so one single `LW` is enough to load a value from the variable and pass it to `printf()`. 

Registers holding temporary data are prefixed with T-, but here we also see some prefixed with S-, the contents of which must be preserved before use in other functions (i.e., saved somewhere).

That is why the value of S0 has been set at address 0x4006cc and has been used again at address 0x4006e8, after the `scanf()` call. The `scanf()` function does not change its value.

### 1.12.4 `scanf()`

As was noted before, it is slightly old-fashioned to use `scanf()` today. But if we have to, we have to check if `scanf()` finishes correctly without an error.

```c
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");
    if (scanf ("%d", &x)==1)
        printf ("You entered %d...\n", x);
    else
        printf ("What you entered? Huh?\n");
    return 0;
}
```

By standard, the `scanf()` function returns the number of fields it has successfully read. In our case, if everything goes fine and the user enters a number `scanf()` returns 1, or in case of error (or `EOF`) — 0.

Let’s add some C code to check the `scanf()` return value and print error message in case of an error.

This works as expected:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

### MSVC: x86

Here is what we get in the assembly output (MSVC 2010):

```
lea   eax, DWORD PTR _x$[ebp]
push  eax
push  OFFSET $SG3833 ; '%d', 00H
call  _scanf
add   esp, 8
cmp   eax, 1
```

---

75 `scanf`, `wscanf`: MSDN
76 `End of File`
The caller function (main()) needs the callee function (scanf()) result, so the callee returns it in the EAX register.

We check it with the help of the instruction CMP EAX, 1 (Compare). In other words, we compare the value in the EAX register with 1.

A JNE conditional jump follows the CMP instruction. JNE stands for Jump if Not Equal.

So, if the value in the EAX register is not equal to 1, the CPU will pass the execution to the address mentioned in the JNE operand, in our case $LN2@main. Passing the control to this address results in the CPU executing printf() with the argument 'You entered %d...' and the value of x.

Since in this case the second printf() has not to be executed, there is a JMP preceding it (unconditional jump). It passes the control to the point after the second printf() and just before the XOR EAX, EAX instruction, which implements return 0.

So, it could be said that comparing a value with another is usually implemented by CMP/Jcc instruction pair, where cc is condition code. CMP compares two values and sets processor flags. Jcc checks those flags and decides to either pass the control to the specified address or not.

This could sound paradoxical, but the CMP instruction is in fact SUB (subtract). All arithmetic instructions set processor flags, not just CMP. If we compare 1 and 1, 1 - 1 is 0 so the ZF flag would be set (meaning that the last result is 0). In no other circumstances ZF can be set, except when the operands are equal. JNE checks only the ZF flag and jumps only if it is not set. JNE is in fact a synonym for JNZ (Jump if Not Zero). Assembler translates both JNE and JNZ instructions into the same opcode. So, the CMP instruction can be replaced with a SUB instruction and almost everything will be fine, with the difference that SUB alters the value of the first operand. CMP is SUB without saving the result, but affecting flags.

MSVC: x86: IDA

It is time to run IDA and try to do something in it. By the way, for beginners it is good idea to use /MD option in MSVC, which means that all these standard functions are not be linked with the executable file, but are to be imported from the MSVCR*.DLL file instead. Thus it will be easier to see which standard function are used and where.

While analyzing code in IDA, it is very helpful to leave notes for oneself (and others). In instance, analyzing this example, we see that JNZ is to be triggered in case of an error. So it is possible to move the cursor to the label, press “n” and rename it to “error”. Create another label—into “exit”. Here is my result:

---

77 X86 flags, see also: wikipedia.
Now it is slightly easier to understand the code. However, it is not a good idea to comment on every instruction.

You could also hide(collapse) parts of a function in IDA. To do that mark the block, then press “–” on the numerical pad and enter the text to be displayed instead.

Let's hide two blocks and give them names:

To expand previously collapsed parts of the code, use “+” on the numerical pad.
By pressing “space”, we can see how IDA represents a function as a graph:

```mdasm
; int __cdecl main()
_main proc near

var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push ebp
mov ebp, esp
push ecx
push offset format ; "Enter X:\n"
call ds:printf
add esp, 4
lea eax, [ebp+var_4]
push eax
push offset a0 ; "%d"
call ds:scanf
add esp, 8
cmp eax, 0
jnz short error

error: ; "What you entered? Huh?\n"
push offset aWhat
call ds:printf
add esp, 4

exit:
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
_Main endp
```

Figure 1.18: Graph mode in IDA

There are two arrows after each conditional jump: green and red. The green arrow points to the block which executes if the jump is triggered, and red if otherwise.
It is possible to fold nodes in this mode and give them names as well ("group nodes"). Let's do it for 3 blocks:

```assembly
; int cdecl main()
__main proc near

var_4= dword ptr -4
argc= dword ptr 8
argu= dword ptr 0Ch
envp= dword ptr 10h

push ebp
mov ebp, esp
push ecx
push offset Format ; "Enter X:\n"
call ds:printf
add esp, 4
lea eax, [ebp+var_4]
push eax
push offset aD ; "%d"
call ds:scanf
add esp, 8
cmp eax, 1
jnz short error
```

That is very useful. It could be said that a very important part of the reverse engineers' job (and any other researcher as well) is to reduce the amount of information they deal with.
MSVC \x86 + OllyDbg

Let's try to hack our program in OllyDbg, forcing it to think `scanf()` always works without error. When an address of a local variable is passed into `scanf()`, the variable initially contains some random garbage, in this case 0x6E494714:

Figure 1.20: OllyDbg: passing variable address into `scanf()`
While `scanf()` executes, in the console we enter something that is definitely not a number, like “asdasd”. `scanf()` finishes with 0 in EAX, which indicates that an error has occurred:

```
while (scanf() executes, in the console we enter something that is definitely not a number, like “asdasd”). `scanf()` finishes with 0 in EAX, which indicates that an error has occurred:
```

We can also check the local variable in the stack and note that it has not changed. Indeed, what would `scanf()` write there? It simply did nothing except returning zero.

Let’s try to “hack” our program. Right-click on EAX, Among the options there is “Set to 1”. This is what we need.

We now have 1 in EAX, so the following check is to be executed as intended, and `printf()` will print the value of the variable in the stack.

When we run the program (F9) we can see the following in the console window:

```
Listing 1.85: console window
```

```
Enter X:
asdasd
You entered 1850296084...
```

Indeed, 1850296084 is a decimal representation of the number in the stack (0x6E494714)!
**MSVC: x86 + Hiew**

This can also be used as a simple example of executable file patching. We may try to patch the executable so the program would always print the input, no matter what we enter.

Assuming that the executable is compiled against external MSVCR*.DLL (i.e., with `/MD` option) 78, we see the main() function at the beginning of the .text section. Let’s open the executable in Hiew and find the beginning of the .text section (Enter, F8, F6, Enter, Enter).

We can see this:

![Hiew: main() function](image)

Hiew finds ASCII 79 strings and displays them, as it does with the imported functions’ names.

---

78 that’s what also called “dynamic linking”

79 ASCII Zero (null-terminated ASCII string)
Move the cursor to address .00401027 (where the JNZ instruction, we have to bypass, is located), press F3, and then type “9090” (meaning two NOPs):

![Image of Hiew: replacing JNZ with two NOPs](image)

Then press F9 (update). Now the executable is saved to the disk. It will behave as we wanted.

Two NOPs are probably not the most aesthetical approach. Another way to patch this instruction is to write just 0 to the second opcode byte (jump offset), so that JNZ will always jump to the next instruction.

We could also do the opposite: replace first byte with EB while not touching the second byte (jump offset). We would get an unconditional jump that is always triggered. In this case the error message would be printed every time, no matter the input.

**MSVC: x64**

Since we work here with int-typed variables, which are still 32-bit in x86-64, we see how the 32-bit part of the registers (prefixed with E-) are used here as well. While working with pointers, however, 64-bit register parts are used, prefixed with R-.

```
<table>
<thead>
<tr>
<th>Listing 1.86: MSVC 2012 x64</th>
</tr>
</thead>
</table>

_DATA SEGMENT
$SG2924 DB 'Enter X: ', 0aH, 00H
$SG2926 DB '%d', 00H
```
ARM

ARM: Optimizing Keil 6/2013 (Thumb mode)

Listing 1.87: Optimizing Keil 6/2013 (Thumb mode)

The new instructions here are **CMP** and **BEQ**.80

CMP is analogous to the x86 instruction with the same name, it subtracts one of the arguments from the other and updates the conditional flags if needed.

80(PowerPC, ARM) Branch if Equal
BEQ jumps to another address if the operands were equal to each other, or, if the result of the last computation has been 0, or if the Z flag is 1. It behaves as JZ in x86.

Everything else is simple: the execution flow forks in two branches, then the branches converge at the point where 0 is written into the R0 as a function return value, and then the function ends.

**ARM64**

Listing 1.88: Non-optimizing GCC 4.9.1 ARM64

```assembly
.LC0:  .string "Enter X:"
.LC1:  .string "%d"
.LC2:  .string "You entered %d...
.LC3:  .string "What you entered? Huh?"

f6: ; save FP and LR in stack frame:
  stp  x29, x30, [sp, -32]!
  add  x29, sp, 0
; load pointer to the "Enter X:" string:
  adrp x0, .LC0
  add  x0, x0, :lo12:.LC0
  bl   puts
; load pointer to the "%d" string:
  adrp x0, .LC1
  add  x0, x0, :lo12:.LC1
  ; calculate address of x variable in the local stack
  add  x1, x29, 28
  bl   __isoc99_scanf
; scanf() returned result in W0.
; check it:
cmp  w0, 1
; BNE is Branch if Not Equal
so if W0<>0, jump to L2 will be occurred
bne  .L2
; at this moment W0=1, meaning no error
; load x value from the local stack
ldr  w1, [x29,28]
; load pointer to the "You entered %d...
  adrp x0, .LC2
  add  x0, x0, :lo12:.LC2
  bl   printf
; skip the code, which print the "What you entered? Huh?" string:
b   .L3
  .L2:
; load pointer to the "What you entered? Huh?" string:
  adrp x0, .LC3
  add  x0, x0, :lo12:.LC3
  bl   puts
  .L3:
; return 0
mov   w0, 0
; restore FP and LR:
ldp  x29, x30, [sp], 32
ret
```

Code flow in this case forks with the use of CMP/BNE (Branch if Not Equal) instructions pair.

**MIPS**

Listing 1.89: Optimizing GCC 4.4.5 (IDA)

```assembly
.text:004006A0  main:
.text:004006A0  var_18  = -0x18
```
\texttt{scanf()} returns the result of its work in register $V0$. It is checked at address 0x004006E4 by comparing the values in $V0$ with $V1$ (1 has been stored in $V1$ earlier, at 0x004006DC). \texttt{BEQ} stands for “Branch Equal”. If the two values are equal (i.e., success), the execution jumps to address 0x0040070C.

Exercise

As we can see, the JNE/JNZ instruction can be easily replaced by the JE/JZ and vice versa (or BNE by BEQ and vice versa). But then the basic blocks must also be swapped. Try to do this in some of the examples.

1.12.5 Exercise

- \texttt{http://challenges.re/53}

1.13 Worth noting: global vs. local variables

Now that you know that global variables are filling with zeroes by OS at start (1.12.3 on page 77, [ISO/IEC 9899:TC3 (C99 standard), (2007)6.7.8p10]), but local variables are not (1.9.4 on page 36).

Sometimes, you have a global variable that you forgot to initialize and your program relies on the fact that it has zero at start. Then you edit a program and move the global variable into a function to make it local. It wouldn’t be zeroed at initialization anymore and this can result in nasty bugs.
1.14 Accessing passed arguments

Now we figured out that the caller function is passing arguments to the callee via the stack. But how does the callee access them?

Listing 1.90: simple example

```c
#include <stdio.h>

int f (int a, int b, int c)
{
    return a*b+c;
};

int main()
{
    printf ("%d\n", f(1, 2, 3));
    return 0;
};
```

1.14.1 x86

MSVC

Here is what we get after compilation (MSVC 2010 Express):

Listing 1.91: MSVC 2010 Express

```
_TEXT SEGMENT
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
_c$ = 16 ; size = 4
_f PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    imul eax, DWORD PTR _b$[ebp]
    add eax, DWORD PTR _c$[ebp]
    pop ebp
    ret 0
_f ENDP

_main PROC
    push ebp
    mov ebp, esp
    push 3 ; 3rd argument
    push 2 ; 2nd argument
    push 1 ; 1st argument
    call _f
    add esp, 12
    push eax
    push OFFSET $SG2463 ; '%d', 0aH, 00H
    call _printf
    add esp, 8
    ; return 0
    xor eax, eax
    pop ebp
    ret 0
_main ENDP
```

What we see is that the main() function pushes 3 numbers onto the stack and calls f(int,int,int). Argument access inside f() is organized with the help of macros like: _a$ = 8, in the same way as local variables, but with positive offsets (addressed with plus). So, we are addressing the outer side of the stack frame by adding the _a$ macro to the value in the EBP register.

Then the value of a is stored into EAX. After IMUL instruction execution, the value in EAX is a product of the value in EAX and the content of _b.

After that, ADD adds the value in _c to EAX.
The value in EAX does not need to be moved: it is already where it must be. On returning to caller, it takes the EAX value and uses it as an argument to printf().

**MSVC + OllyDbg**

Let’s illustrate this in OllyDbg. When we trace to the first instruction in f() that uses one of the arguments (first one), we see that EBP is pointing to the stack frame, which is marked with a red rectangle.

The first element of the stack frame is the saved value of EBP, the second one is RA, the third is the first function argument, then the second and third ones.

To access the first function argument, one needs to add exactly 8 (2 32-bit words) to EBP.

OllyDbg is aware about this, so it has added comments to the stack elements like “RETURN from” and “Arg1 = …”, etc.

N.B.: Function arguments are not members of the function’s stack frame, they are rather members of the stack frame of the caller function.

Hence, OllyDbg marked “Arg” elements as members of another stack frame.

**GCC**

Let’s compile the same in GCC 4.4.1 and see the results in IDA:

Listing 1.92: GCC 4.4.1

```c
public f
f   proc near
arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
arg_8 = dword ptr 10h
push ebp
mov ebp, esp
mov eax, [ebp+arg_0] ; 1st argument
imul eax, [ebp+arg_4] ; 2nd argument
add eax, [ebp+arg_8] ; 3rd argument
pop ebp
retn
f   endp

public main
```

Figure 1.24: OllyDbg: inside of f() function
main proc near

var_10 = dword ptr -10h
var_C  = dword ptr -0Ch
var_8  = dword ptr -8

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 10h
mov     [esp+10h+var_8], 3 ; 3rd argument
mov     [esp+10h+var_C], 2 ; 2nd argument
mov     [esp+10h+var_10], 1 ; 1st argument
call    f
mov     edx, offset aD ; "%d\n"
mov     [esp+10h+var_C], eax
call    _printf
mov     eax, 0
leave
ret

main endp

The result is almost the same with some minor differences discussed earlier.
The stack pointer is not set back after the two function calls (f and printf), because the penultimate LEAVE (.1.6 on page 1014) instruction takes care of this at the end.

1.14.2 x64

The story is a bit different in x86-64. Function arguments (first 4 or first 6 of them) are passed in registers i.e. the callee reads them from registers instead of reading them from the stack.

MSVC

Optimizing MSVC:

Listing 1.93: Optimizing MSVC 2012 x64

```
$SG2997 DB 'd', 0aH, 00H

main PROC
sub     rsp, 40
mov     edx, 2
lea     r8d, QWORD PTR [rdx+1] ; R8D=3
lea     ecx, QWORD PTR [rdx-1] ; ECX=1
call    f
lea     rcx, OFFSET FLAT:$SG2997 ; '%d'
mov     edx, eax
call    printf
xor     eax, eax
add     rsp, 40
ret     0
main ENDP

f PROC
; ECX - 1st argument
; EDX - 2nd argument
; R8D - 3rd argument
imul    ecx, edx
lea     eax, DWORD PTR [r8+rcx]
ret     0
f ENDP
```

As we can see, the compact function f() takes all its arguments from the registers.
The LEA instruction here is used for addition, apparently the compiler considered it faster than ADD.
LEA is also used in the main() function to prepare the first and third f() arguments. The compiler must have decided that this would work faster than the usual way of loading values into a register using MOV instruction.

Let's take a look at the non-optimizing MSVC output:

Listing 1.94: MSVC 2012 x64

```assembly
f proc near
; shadow space:
arg_0 = dword ptr 8
arg_8 = dword ptr 10h
arg_10 = dword ptr 18h

; ECX - 1st argument
; EDX - 2nd argument
; R8D - 3rd argument
mov [rsp+arg_10], r8d
mov [rsp+arg_8], edx
mov [rsp+arg_0], ecx
imul eax, [rsp+arg_0]
add eax, [rsp+arg_8]
retn
f endp

main proc near
sub rsp, 28h
mov r8d, 3 ; 3rd argument
mov edx, 2 ; 2nd argument
mov ecx, 1 ; 1st argument
call f
mov edx, eax
lea rcx, $SG2931 ; "%d\n"
call printf

; return 0
xor eax, eax
add rsp, 28h
retn
main endp
```

It looks somewhat puzzling because all 3 arguments from the registers are saved to the stack for some reason. This is called “shadow space”\(^1\): every Win64 may (but is not required to) save all 4 register values there. This is done for two reasons: 1) it is too lavish to allocate a whole register (or even 4 registers) for an input argument, so it will be accessed via stack; 2) the debugger is always aware where to find the function arguments at a break\(^2\).

So, some large functions can save their input arguments in the “shadow space” if they want to use them during execution, but some small functions (like ours) may not do this.

It is a caller responsibility to allocate “shadow space” in the stack.

**GCC**

Optimizing GCC generates more or less understandable code:

Listing 1.95: Optimizing GCC 4.4.6 x64

```assembly
f:

; EDI - 1st argument
; ESI - 2nd argument
; EDX - 3rd argument
imul esi, edi
lea eax, [rdx+rsi]
ret
```

\(^1\)MSDN
\(^2\)MSDN
There are no "shadow space" requirements in System V *NIX ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, System V Application Binary Interface. AMD64 Architecture Processor Supplement, (2013)] 83), but the callee may want to save its arguments somewhere in case of registers shortage.

**GCC: uint64_t instead of int**

Our example works with 32-bit int, that is why 32-bit register parts are used (prefixed by E-).

It can be altered slightly in order to use 64-bit values:

```c
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
    return a*b+c;
};

int main()
```

---

83Also available as https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf
{  
    printf("%lld\n", f(0x1122334455667788,  
             0x1111111122222222,  
             0x3333333444444444));  
    return 0;  
};

Listing 1.97: Optimizing GCC 4.4.6 x64

f   proc near
    imul   rsi, rdi
    lea    rax, [rdx+rsi]
    retn
f   endp

main proc near
    sub    rsp, 8
    mov    rdx, 3333333444444444h ; 3rd argument
    mov    rsi, 1111111122222222h ; 2nd argument
    mov    rdi, 1122334455667788h ; 1st argument
    call   f
    mov    edi, offset format ; "%lld\n"
    mov    rsi, rax
    xor    eax, eax ; number of vector registers passed
    call   __printf
    xor    eax, eax
    add    rsp, 8
    retn
main endp

The code is the same, but this time the full size registers (prefixed by R-) are used.

1.14.3 ARM
Non-optimizing Keil 6/2013 (ARM mode)

The main() function simply calls two other functions, with three values passed to the first one — (f()).

As was noted before, in ARM the first 4 values are usually passed in the first 4 registers (R0-R3).

The f() function, as it seems, uses the first 3 registers (R0-R2) as arguments.

The MLA (Multiply Accumulate) instruction multiplies its first two operands (R3 and R1), adds the third operand (R2) to the product and stores the result into the zeroth register (R0), via which, by standard, functions return values.

Multiplication and addition at once (Fused multiply–add) is a very useful operation. By the way, there was no such instruction in x86 before FMA-instructions appeared in SIMD 84.

The very first MOV R3, R0, instruction is, apparently, redundant (a single MLA instruction could be used here instead). The compiler has not optimized it, since this is non-optimizing compilation.

84 wikipedia
The BX instruction returns the control to the address stored in the LR register and, if necessary, switches the processor mode from Thumb to ARM or vice versa. This can be necessary since, as we can see, function f() is not aware from what kind of code it may be called, ARM or Thumb. Thus, if it gets called from Thumb code, BX is not only returns control to the calling function, but also switches the processor mode to Thumb. Or not switch, if the function has been called from ARM code [ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition, (2012)A2.3.2].

### Optimizing Keil 6/2013 (ARM mode)

```
.text:00000098     f
.text:00000098 91 20 20 E0   MLA   R0, R1, R0, R2
.text:0000009C 1E FF 2F E1   BX   LR
```

And here is the f() function compiled by the Keil compiler in full optimization mode (-O3).

The MOV instruction was optimized out (or reduced) and now MLA uses all input registers and also places the result right into R0, exactly where the calling function will read and use it.

### Optimizing Keil 6/2013 (Thumb mode)

```
.text:0000005E 48 43   MULS   R0, R1
.text:00000060 80 18   ADDS   R0, R0, R2
.text:00000062 70 47   BX   LR
```

The MLA instruction is not available in Thumb mode, so the compiler generates the code doing these two operations (multiplication and addition) separately.

First the MULS instruction multiplies R0 by R1, leaving the result in register R0. The second instruction (ADDS) adds the result and R2 leaving the result in register R0.

### ARM64

### Optimizing GCC (Linaro) 4.9

Everything here is simple. MADD is just an instruction doing fused multiply/add (similar to the MLA we already saw). All 3 arguments are passed in the 32-bit parts of X-registers. Indeed, the argument types are 32-bit int’s. The result is returned in W0.

#### Listing 1.98: Optimizing GCC (Linaro) 4.9

```
f:    madd   w0, w0, w1, w2
     ret

main:
    ; save FP and LR to stack frame:
    stp   x29, x30, [sp, -16]!
    mov   w2, 3
    mov   w1, 2
    add   x29, sp, 0
    mov   w0, 1
    bl    f
    mov   w1, w0
    adrp  x0, .LC7
    add   x0, x0, :lo12:.LC7
    bl    printf
    ; return 0
    mov   w0, 0
    ; restore FP and LR
    ldp   x29, x30, [sp], 16
    ret

.LC7:
    .string "%d\n"
```

Let’s also extend all data types to 64-bit uint64_t and test:
```c
#include <stdio.h>
#include <stdint.h>

uint64_t f(uint64_t a, uint64_t b, uint64_t c)
{
    return a*b+c;
};

int main()
{
    printf("%lld\n", f(0x1122334455667788,
                      0x1111111122222222,
                      0x3333333344444444));
    return 0;
};
```

The `f()` function is the same, only the whole 64-bit X-registers are now used. Long 64-bit values are loaded into the registers by parts, this is also described here: 1.39.3 on page 438.

**Non-optimizing GCC (Linaro) 4.9**

The non-optimizing compiler is more redundant:

```assembly
f:
    madd   x0, x0, x1, x2
    ret

main:
    mov    x1, 13396
    adrp   x0, .LC8
    stp    x29, x30, [sp, -16]!
    movk   x1, 0x27d0, lsl 16
    add    x0, x0, :lo12:.LC8
    movk   x1, 0x122, lsl 32
    add    x29, sp, 0
    movk   x1, 0x58be, lsl 48
    bl     printf
    mov    w0, 0
    ldp    x29, x30, [sp], 16
    ret

.LC8:
    .string "%lld\n"
```

The code saves its input arguments in the local stack, in case someone (or something) in this function needs using the W0...W2 registers. This prevents overwriting the original function arguments, which may be needed again in the future.

This is called Register Save Area. ((Procedure Call Standard for the ARM 64-bit Architecture (AArch64), (2013))85). The callee, however, is not obliged to save them. This is somewhat similar to “Shadow Space”: 1.14.2 on page 100.

85 Also available as http://go.yurichev.com/17287
Why did the optimizing GCC 4.9 drop this argument saving code? Because it did some additional optimizing work and concluded that the function arguments will not be needed in the future and also that the registers W0...W2 will not be used.

We also see a MUL/ADD instruction pair instead of single a MADD.

### 1.14.4 MIPS

Listing 1.99: Optimizing GCC 4.4.5

```
.text:00000000 f:
; $a0=a
; $a1=b
; $a2=c
.text:00000000 mult $a1, $a0
.text:00000004 mflo $v0
.text:00000008 jr $ra
.text:0000000c addu $v0, $a2, $v0 ; branch delay slot
; result is in $v0 upon return
.text:00000010 main:
.text:00000010 var_10 = -0x10
.text:00000010 var_4 = -4
.text:00000010 lui $gp, (_gnu_local_gp >> 16)
.text:00000014 addiu $sp, -0x20
.text:00000018 la $gp, (_gnu_local_gp & 0xFFFF)
.text:0000001c sw $ra, 0x20+var_4($sp)
.text:00000020 sw $gp, 0x20+var_10($sp)
; set c:
.text:00000024 li $a2, 3
; set a:
.text:00000028 li $a0, 1
.text:0000002c jal f
; set b:
.text:00000030 li $a1, 2 ; branch delay slot
; result in $v0 now
.text:00000034 lw $gp, 0x20+var_10($sp)
.text:00000038 lui $a0, ($LC0 >> 16)
.text:0000003c lw $t9, (printf & 0xFFFF)($gp)
.text:00000040 la $a0, ($LC0 & 0xFFFF)
.text:00000044 jalr $t9
; take result of f() function and pass it as a second argument to printf():
.text:00000048 move $a1, $v0 ; branch delay slot
.text:0000004c lw $ra, 0x20+var_4($sp)
.text:00000050 move $v0, $zero
.text:00000054 jr $ra
.text:00000058 addiu $sp, 0x20 ; branch delay slot
```

The first four function arguments are passed in four registers prefixed by A-.

There are two special registers in MIPS: HI and LO which are filled with the 64-bit result of the multiplication during the execution of the MULT instruction.

These registers are accessible only by using the MFLO and MFHI instructions. MFLO here takes the low-part of the multiplication result and stores it into $V0. So the high 32-bit part of the multiplication result is dropped (the HI register content is not used). Indeed: we work with 32-bit int data types here.

Finally, ADDU (“Add Unsigned”) adds the value of the third argument to the result.

There are two different addition instructions in MIPS: ADD and ADDU. The difference between them is not related to signedness, but to exceptions. ADD can raise an exception on overflow, which is sometimes useful and supported in Ada PL, for instance. ADDU does not raise exceptions on overflow.

Since C/C++ does not support this, in our example we see ADDU instead of ADD.

The 32-bit result is left in $V0.

There is a new instruction for us in main(): JAL (“Jump and Link”).

86http://go.yurichev.com/17326
The difference between JAL and JALR is that a relative offset is encoded in the first instruction, while JALR jumps to the absolute address stored in a register (“Jump and Link Register”).

Both f() and main() functions are located in the same object file, so the relative address of f() is known and fixed.

1.15 More about results returning

In x86, the result of function execution is usually returned \(^8^7\) in the EAX register. If it is byte type or a character (char), then the lowest part of register EAX (AL) is used. If a function returns a float number, the FPU register ST(0) is used instead. In ARM, the result is usually returned in the R0 register.

1.15.1 Attempt to use the result of a function returning void

So, what if the main() function return value was declared of type void and not int? The so-called startup-code is calling main() roughly as follows:

```
push envp
push argv
push argc
call main
push eax
call exit
```

In other words:

```
extit(main(argc,argv,envp));
```

If you declare main() as void, nothing is to be returned explicitly (using the return statement), then something random, that has been stored in the EAX register at the end of main() becomes the sole argument of the exit() function. Most likely, there will be a random value, left from your function execution, so the exit code of program is pseudorandom.

We can illustrate this fact. Please note that here the main() function has a void return type:

```c
#include <stdio.h>

void main()
{
    printf("Hello, world!\n");
}
```

Let’s compile it in Linux.

GCC 4.8.1 replaced printf() with puts() (we have seen this before: 1.5.3 on page 20), but that’s OK, since puts() returns the number of characters printed out, just like printf(). Please notice that EAX is not zeroed before main()’s end.

This implies that the value of EAX at the end of main() contains what puts() has left there.

Listing 1.100: GCC 4.8.1

```
.LC0:
    .string "Hello, world!"

main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 16
    mov     DWORD PTR [esp], OFFSET FLAT:.LC0
    call    puts
    leave
    ret
```

Let’s write a bash script that shows the exit status:

---

\(^8^7\)See also: MSDN: Return Values (C++): MSDN

106
Listing 1.101: tst.sh

```bash
#!/bin/sh
./hello_world
echo $?
```

And run it:

```bash
$ tst.sh
Hello, world!
14
```

14 is the number of characters printed. The number of characters printed slips from `printf()` through EAX/RAX into “exit code”.

Another example in the book: 3.32 on page 638.

By the way, when we decompile C++ in Hex-Rays, we can often encounter a function which terminated with destructor of some class:

```assembly
...  
call  ??1CString@@QAE@XZ ; CString:: CString(void)  
mov   ecx, [esp+30h+var_C]  
pop   edi  
pop   ebx  
mov   large fs:0, ecx  
add   esp, 28h  
retn  
```

By C++ standard, destructor doesn’t return anything, but when Hex-Rays don’t know about it, and thinks that both destructor and this function returns `int`, we can see something like that in output:

```assembly
...  
  return CString::~CString(&Str);  
}
```

### 1.15.2 What if we do not use the function result?

`printf()` returns the count of characters successfully output, but the result of this function is rarely used in practice.

It is also possible to call a function whose essence is in returning a value, and not use it:

```c
int f()
{
    // skip first 3 random values:
    rand();
    rand();
    rand();
    // and use 4th:
    return rand();
}
```

The result of the `rand()` function is left in EAX, in all four cases.

But in the first 3 cases, the value in EAX is just not used.

### 1.15.3 Returning a structure

Let’s go back to the fact that the return value is left in the EAX register.

That is why old C compilers cannot create functions capable of returning something that does not fit in one register (usually `int`), but if one needs it, one have to return information via pointers passed as function’s arguments.

So, usually, if a function needs to return several values, it returns only one, and all the rest—via pointers.
Now it has become possible to return, let's say, an entire structure, but that is still not very popular. If a function has to return a large structure, the caller must allocate it and pass a pointer to it via the first argument, transparently for the programmer. That is almost the same as to pass a pointer in the first argument manually, but the compiler hides it.

Small example:

```c
struct s {
    int a;
    int b;
    int c;
};
struct s get_some_values (int a) {
    struct s rt;
    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;
    return rt;
};
```

...what we got (MSVC 2010 /Ox):

```c
$T3853 = 8        ; size = 4
.a$ = 12         ; size = 4
?get_some_values@YA?AUs@H@Z PROC ; get_some_values
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, DWORD PTR $T3853[esp-4]
    lea    edx, DWORD PTR [ecx+1]
    mov    DWORD PTR [eax], edx
    lea    edx, DWORD PTR [ecx+2]
    add    ecx, 3
    mov    DWORD PTR [eax+4], edx
    mov    DWORD PTR [eax+8], ecx
    ret
?get_some_values@YA?AUs@H@Z ENDP ; get_some_values
```

The macro name for internal passing of pointer to a structure here is $T3853$.

This example can be rewritten using the C99 language extensions:

```c
struct s {
    int a;
    int b;
    int c;
};
struct s get_some_values (int a) {
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
```

Listing 1.102: GCC 4.8.1

```c
_ptr_to_struct = dword ptr 4
a = dword ptr 8
    mov    edx, [esp+a]
    mov    eax, [esp+_ptr_to_struct]
    lea    ecx, [edx+1]
    mov    [eax], ecx
    lea    ecx, [edx+2]
    add    edx, 3
    mov    [eax+4], ecx
```
As we see, the function is just filling the structure’s fields allocated by the caller function, as if a pointer to the structure has been passed. So there are no performance drawbacks.

### 1.16 Pointers

#### 1.16.1 Returning values

Pointers are often used to return values from functions (recall `scanf()` case ([1.12 on page 65](#))).

For example, when a function needs to return two values.

**Global variables example**

```c
#include <stdio.h>

void f1(int x, int y, int *sum, int *product) {
    *sum=x+y;
    *product=x*y;
};
int sum, product;
void main() {
    f1(123, 456, &sum, &product);
    printf("sum=\%d, product=\%d\n", sum, product);
};
```

This compiles to:

**Listing 1.103: Optimizing MSVC 2010 (/Ob0)**

```
COMM _product:DWORD
COMM _sum:DWORD
$SG2803 DB 'sum=\%d, product=\%d', 0aH, 00H

_x$ = 8 ; size = 4
_y$ = 12 ; size = 4
_sum$ = 16 ; size = 4
_product$ = 20 ; size = 4
_f1 PROC
    mov   ecx, DWORD PTR _y$[esp-4]
    mov   eax, DWORD PTR _x$[esp-4]
    lea   edx, DWORD PTR [eax+ecx]
    imul  eax, ecx
    mov   ecx, DWORD PTR _product$[esp-4]
    push  esi
    mov   esi, DWORD PTR _sum$[esp]
    mov   DWORD PTR [esi], edx
    mov   DWORD PTR [ecx], eax
    pop   esi
    ret 0
_f1 ENDP

_main PROC
    push OFFSET _product
    push OFFSET _sum
    push 456 ; 000001c8H
    push 123 ; 0000007bH
    call _f1
    mov   eax, DWORD PTR _product
    mov   ecx, DWORD PTR _sum
    push  eax
```

109
push ecx
push OFFSET $SG2803
call DWORD PTR __imp__printf
add esp, 28
xor eax, eax
ret 0
_main ENDP
Let's see this in OllyDbg:

Figure 1.25: OllyDbg: global variables addresses are passed to f1()

First, global variables' addresses are passed to f1(). We can click “Follow in dump” on the stack element, and we can see the place in the data segment allocated for the two variables.
These variables are zeroed, because non-initialized data (from BSS) is cleared before the execution begins, [see ISO/IEC 9899:TC3 (C C99 standard), (2007) 6.7.8p10].

They reside in the data segment, we can verify this by pressing Alt-M and reviewing the memory map:

![Figure 1.26: OllyDbg: memory map](image-url)
Let's trace (F7) to the start of f1():

Two values are visible in the stack: 456 (0x1C8) and 123 (0x7B), and also the addresses of the two global variables.

![Figure 1.27: OllyDbg: f1() starts](image)
Let's trace until the end of f1(). In the left bottom window we see how the results of the calculation appear in the global variables:

Figure 1.28: OllyDbg: f1() execution completed
Now the global variables' values are loaded into registers ready for passing to printf() (via the stack):

![Image of OllyDbg showing global variables' values passed into printf()](image)

**Figure 1.29: OllyDbg: global variables' values are passed into printf()**

### Local variables example

Let's rework our example slightly:

#### Listing 1.104: now the sum and product variables are local

```c
void main()
{
    int sum, product; // now variables are local in this function

    f1(123, 456, &sum, &product);
    printf("sum=%d, product=%d\n", sum, product);
}
```

f1() code will not change. Only the code of main() will do:

#### Listing 1.105: Optimizing MSVC 2010 (/Ob0)

```assembly
_product$ = -8 ; size = 4
sum$ = -4 ; size = 4
main PROC
; Line 10
    sub esp, 8
; Line 13
    lea eax, DWORD PTR _product$[esp+8]
    push eax
    lea ecx, DWORD PTR _sum$[esp+12]
    push ecx
    push 456 ; 000001c8H
    push 123 ; 0000007bH
    call _f1
; Line 14
    mov edx, DWORD PTR _product$[esp+24]
    mov eax, DWORD PTR _sum$[esp+24]
    push edx
    push eax
    push OFFSET $SG2803
    call DWORD PTR ___imp__printf
; Line 15
    xor eax, eax
    add esp, 36
    ret 0
```

115
Let’s look again with OllyDbg. The addresses of the local variables in the stack are 0x2EF854 and 0x2EF858. We see how these are pushed into the stack:

![OllyDbg screenshot](image)

Figure 1.30: OllyDbg: local variables’ addresses are pushed into the stack
f1() starts. So far there is only random garbage in the stack at 0x2EF854 and 0x2EF858:

Figure 1.31: OllyDbg: f1() starting
f1() completes:

Figure 1.32: OllyDbg: f1() completes execution

We now find 0xDB18 and 0x243 at addresses 0x2EF584 and 0x2EF588. These values are the f1() results.

**Conclusion**

f1() could return pointers to any place in memory, located anywhere.

This is in essence the usefulness of the pointers.

By the way, C++ references work exactly the same way. Read more about them: (3.21.3 on page 558).

### 1.16.2 Swap input values

This will do the job:

```c
#include <memory.h>
#include <stdio.h>

void swap_bytes(unsigned char* first, unsigned char* second) {
    unsigned char tmp1;
    unsigned char tmp2;

    tmp1=*first;
    tmp2=*second;

    *first=tmp2;
    *second=tmp1;
}

int main() {
    // copy string into heap, so we will be able to modify it
    char *s=strdup("string");

    // swap 2nd and 3rd characters
    swap_bytes (s+1, s+2);

    printf( "%s\n", s);
}
```
As we can see, bytes are loaded into lower 8-bit parts of ECX and EBX using MOVZX (so higher parts of these registers will be cleared) and then bytes are written back swapped.

Listing 1.106: Optimizing GCC 5.4

```c
swap_bytes:
    push ebx
    mov edx, DWORD PTR [esp+8]
    mov eax, DWORD PTR [esp+12]
    movzx ecx, BYTE PTR [edx]
    movzx ebx, BYTE PTR [eax]
    mov BYTE PTR [edx], bl
    mov BYTE PTR [eax], cl
    pop ebx
    ret
```

Addresses of both bytes are taken from arguments and through execution of the function are located in EDX and EAX.

So we use pointers: probably, there is no better way to solve this task without them.

### 1.17 GOTO operator

The GOTO operator is generally considered as anti-pattern, see [Edgar Dijkstra, Go To Statement Considered Harmful (1968)][88]. Nevertheless, it can be used reasonably, see [Donald E. Knuth, Structured Programming with go to Statements (1974)][89] 90.

Here is a very simple example:

```c
#include <stdio.h>

int main()
{
    printf("begin\n");
    goto exit;
    printf("skip me!\n");
exit:
    printf("end\n");
}
```

Here is what we have got in MSVC 2012:

Listing 1.107: MSVC 2012

```assembly
$SG2934 DB 'begin', 0aH, 00H
$SG2936 DB 'skip me!', 0aH, 00H
$SG2937 DB 'end', 0aH, 00H

_main PROC
    push ebp
    mov ebp, esp
    push OFFSET $SG2934 ; 'begin'
    call _printf
    add esp, 4
    jmp SHORT $exit$3
    push OFFSET $SG2936 ; 'skip me'
    call _printf
    add esp, 4
$exit$3:
    push OFFSET $SG2937 ; 'end'
    call _printf
    add esp, 4
    xor eax, eax
    pop ebp
    ret 0

_main ENDP
```

[Dennis Yurichev, C/C++ programming language notes][90] also has some examples.
The *goto* statement has been simply replaced by a *JMP* instruction, which has the same effect: unconditional jump to another place. The second *printf()* could be executed only with human intervention, by using a debugger or by patching the code.
This could also be useful as a simple patching exercise. Let’s open the resulting executable in Hiew:

![Hiew: goto.exe](image)

Figure 1.33: Hiew
Place the cursor to address JMP (0x410), press F3 (edit), press zero twice, so the opcode becomes EB 00:

**Figure 1.34: Hiew**

The second byte of the JMP opcode denotes the relative offset for the jump, 0 means the point right after the current instruction.

So now JMP not skipping the second `printf()` call.

Press F9 (save) and exit. Now if we run the executable we will see this:

```
Listing 1.108: Patched executable output
C:\...>goto.exe
begin
skip me!
end
```

The same result could be achieved by replacing the JMP instruction with 2 NOP instructions.

NOP has an opcode of 0x90 and length of 1 byte, so we need 2 instructions as JMP replacement (which is 2 bytes in size).

### 1.17.1 Dead code

The second `printf()` call is also called “dead code” in compiler terms.

This means that the code will never be executed. So when you compile this example with optimizations, the compiler removes “dead code”, leaving no trace of it:

```
Listing 1.109: Optimizing MSVC 2012
SG2981 DB 'begin', 0aH, 00H
SG2983 DB 'skip me!', 0aH, 00H
SG2984 DB 'end', 0aH, 00H

_main PROC
push OFFSET $SG2981 ; 'begin'
call _printf
push OFFSET $SG2984 ; 'end'
$exit$4:
call _printf
add esp, 8
xor eax, eax
ret 0
```
However, the compiler forgot to remove the “skip me!” string.

1.17.2 Exercise
Try to achieve the same result using your favorite compiler and debugger.

1.18 Conditional jumps
1.18.1 Simple example

```c
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
}

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
}

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
}
```

x86

x86 + MSVC

Here is how the f_signed() function looks like:

```
Listing 1.110: Non-optimizing MSVC 2010

_a$ = 8
_b$ = 12
_f_signed PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    cmp eax, DWORD PTR _b$[ebp]
    jle SHORT $LN3@f_signed
    push OFFSET $SG737 ; 'a>b'
    call _printf
    add esp, 4
$LN3@f_signed:
    mov ecx, DWORD PTR _a$[ebp]
    cmp ecx, DWORD PTR _b$[ebp]
    jne SHORT $LN2@f_signed
    push OFFSET $SG739 ; 'a==b'
    call _printf
```

123
The first instruction, JLE, stands for Jump if Less or Equal. In other words, if the second operand is larger or equal to the first one, the control flow will be passed to the address or label specified in the instruction. If this condition does not trigger because the second operand is smaller than the first one, the control flow would not be altered and the first printf() would be executed. The second check is JNE: Jump if Not Equal. The control flow will not change if the operands are equal.

The third check is JGE: jump if Greater or Equal—jump if the first operand is larger than the second or if they are equal. So, if all three conditional jumps are triggered, none of the printf() calls would be executed whatsoever. This is impossible without special intervention. Now let's take a look at the f_unsigned() function. The f_unsigned() function is the same as f_signed(), with the exception that the JBE and JAE instructions are used instead of JLE and JGE, as follows:

As already mentioned, the branch instructions are different: JBE—Jump if Below or Equal and JAE—Jump if Above or Equal. These instructions (JA/JAE/JB/JBE) differ from JG/JGE/JL/JLE in the fact that they work with unsigned numbers.

See also the section about signed number representations (2.2 on page 448). That is why if we see JG/JL in use instead of JA/JB or vice-versa, we can be almost sure that the variables are signed or unsigned, respectively. Here is also the main() function, where there is nothing much new to us:
push 2
push 1
call _f_signed
add esp, 8
push 2
push 1
call _f_unsigned
add esp, 8
xor eax, eax
pop ebp
ret 0

_main ENDP
We can see how flags are set by running this example in OllyDbg. Let’s begin with `f_unsigned()`, which works with unsigned numbers.

CMP is executed thrice here, but for the same arguments, so the flags are the same each time.

Result of the first comparison:

Figure 1.35: OllyDbg: `f_unsigned()`: first conditional jump

So, the flags are: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0.

They are named with one character for brevity in OllyDbg.

OllyDbg gives a hint that the (JBE) jump is to be triggered now. Indeed, if we take a look into Intel manuals (12.1.4 on page 999), we can read there that JBE is triggering if CF=1 or ZF=1. The condition is true here, so the jump is triggered.
The next conditional jump:

Figure 1.36: OllyDbg: \texttt{f\_unsigned()}: second conditional jump

OllyDbg gives a hint that JNZ is to be triggered now. Indeed, JNZ triggering if ZF=0 (zero flag).
The third conditional jump, JNB:

![CPU - main thread, module ex](image)

Figure 1.37: OllyDbg: f_unsigned(): third conditional jump

In Intel manuals (12.1.4 on page 999) we can see that JNB triggers if CF=0 (carry flag). That is not true in our case, so the third printf() will execute.
Now let's review the `f_signed()` function, which works with signed values, in OllyDbg. Flags are set in the same way: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. The first conditional jump JLE is to be triggered:

In Intel manuals (12.1.4 on page 999) we find that this instruction is triggered if ZF=1 or SF≠OF. SF≠OF in our case, so the jump triggers.
The second JNZ conditional jump triggering: if ZF=0 (zero flag):

Figure 1.39: OllyDbg: f_signed(): second conditional jump
The third conditional jump JGE will not trigger because it would only do so if SF=OF, and that is not true in our case:

Figure 1.40: OllyDbg: f_signed(): third conditional jump
We can try to patch the executable file in a way that the \texttt{f\_unsigned()} function would always print “a==b”, no matter the input values. Here is how it looks in Hiew:

![Hiew: f\_unsigned() function](image)

Essentially, we have to accomplish three tasks:

- force the first jump to always trigger;
- force the second jump to never trigger;
- force the third jump to always trigger.

Thus we can direct the code flow to always pass through the second \texttt{printf()}, and output “a==b”.

Three instructions (or bytes) has to be patched:

- The first jump becomes \texttt{JMP}, but the \texttt{jump offset} would remain the same.
- The second jump might be triggered sometimes, but in any case it will jump to the next instruction, because, we set the \texttt{jump offset} to 0.
  
  In these instructions the \texttt{jump offset} is added to the address for the next instruction. So if the offset is 0, the jump will transfer the control to the next instruction.
- The third jump we replace with \texttt{JMP} just as we do with the first one, so it will always trigger.
Here is the modified code:

![Image of the modified code]

Figure 1.42: Hiew: let's modify the f_unsigned() function

If we miss to change any of these jumps, then several printf() calls may execute, while we want to execute only one.

Non-optimizing GCC

Non-optimizing GCC 4.4.1 produces almost the same code, but with puts() (1.5.3 on page 20) instead of printf().

Optimizing GCC

An observant reader may ask, why execute CMP several times, if the flags has the same values after each execution?

Perhaps optimizing MSVC cannot do this, but optimizing GCC 4.8.1 can go deeper:

Listing 1.113: GCC 4.8.1 f_signed()

```assembly
f_signed:
    mov    eax, DWORD PTR [esp+8]
    cmp    DWORD PTR [esp+4], eax
    jg     .L6
    je     .L7
    jge    .L1
    mov    DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
    jmp    puts
.L6:
    mov    DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
    jmp    puts
```
We also see JMP puts here instead of CALL puts / RETN.
This kind of trick will have explained later: 1.21.1 on page 154.
This type of x86 code is somewhat rare. MSVC 2012 as it seems, can't generate such code. On the other hand, assembly language programmers are fully aware of the fact that Jcc instructions can be stacked. So if you see such stacking somewhere, it is highly probable that the code was hand-written.

The f_unsigned() function is not that æsthetically short:

```
Listing 1.114: GCC 4.8.1 f_unsigned()

f_unsigned:
    push   esi
    push   ebx
    sub    esp, 20
    mov    esi, DWORD PTR [esp+32]
    mov    ebx, DWORD PTR [esp+36]
    cmp    esi, ebx
    ja     .L13
    cmp    esi, ebx ; this instruction could be removed
    je     .L14
    jb     .L15
    add    esp, 20
    pop    ebx
    pop    esi
    ret

.L10:
    .L15:
    mov    DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
    add    esp, 20
    pop    ebx
    pop    esi
    jmp    puts

.L13:
    mov    DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
    call   puts
    cmp    esi, ebx
    jne    .L10

.L14:
    mov    DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
    add    esp, 20
    pop    ebx
    pop    esi
    jmp    puts
```

Nevertheless, there are two CMP instructions instead of three.
So optimization algorithms of GCC 4.8.1 are probably not perfect yet.

**ARM**

**32-bit ARM**

**Optimizing Keil 6/2013 (ARM mode)**
Many instructions in ARM mode could be executed only when specific flags are set. E.g. this is often used when comparing numbers.

For instance, the ADD instruction is in fact named ADDAL internally, where AL stands for Always, i.e., execute always. The predicates are encoded in 4 high bits of the 32-bit ARM instructions (condition field). The B instruction for unconditional jumping is in fact conditional and encoded just like any other conditional jump, but has AL in the condition field, and it implies execute Always, ignoring flags.

The ADRGT instruction works just like ADR but executes only in case the previous CMP instruction founds one of the numbers greater than the another, while comparing the two (Greater Than).

The next BLGT instruction behaves exactly as BL and is triggered only if the result of the comparison has been (Greater Than). ADRGT writes a pointer to the string a>b\n into R0 and BLGT calls printf(). Therefore, instructions suffixed with -GT are to execute only in case the value in R0 (which is a) is bigger than the value in R4 (which is b).

Moving forward we see the ADRREQ and BLEQ instructions. They behave just like ADR and BL, but are to be executed only if operands were equal to each other during the last comparison. Another CMP is located before them (because the printf() execution may have tampered the flags).

Then we see LDMGEFD, this instruction works just like LDFMFD\textsuperscript{91}, but is triggered only when one of the values is greater or equal than the other (Greater or Equal). The LDMGEFD SP!, {R4-R6,PC} instruction acts like a function epilogue, but it will be triggered only if a >= b, and only then the function execution will finish.

But if that condition is not satisfied, i.e., a < b, then the control flow will continue to the next “LDMFD SP!, {R4-R6,LR}” instruction, which is one more function epilogue. This instruction restores not only the R4-R6 registers state, but also LR instead of PC, thus, it does not return from the function. The last two instructions call printf() with the string «a\n» as a sole argument. We already examined an unconditional jump to the printf() function instead of function return in «printf() with several arguments» section (1.11.2 on page 53).

f unsigned is similar, only the ADRHI, BLHI, and LDMCSFD instructions are used there, these predicates (HI = Unsigned higher, CS = Carry Set (greater than or equal)) are analogous to those examined before, but for unsigned values.

There is not much new in the main() function for us:

\texttt{Listing \texttt{1.116: \texttt{main()}}}

That is how you can get rid of conditional jumps in ARM mode.

Why is this so good? Read here: \texttt{2.10.1 on page 462.}

\textsuperscript{91}LDFMFD
There is no such feature in x86, except the CMOVcc instruction, it is the same as MOV, but triggered only when specific flags are set, usually set by CMP.

**Optimizing Keil 6/2013 (Thumb mode)**

---

**Listing 1.117: Optimizing Keil 6/2013 (Thumb mode)**

```
.text:00000072 f_signed ; CODE XREF: main+6
.text:00000072 70 B5 PUSH {R4-R6,LR}
.text:00000074 0C 00 MOV R4, R1
.text:00000076 05 00 MOV R5, R0
.text:00000078 A0 42 CMP R0, R4
.text:0000007A 02 DD BLE loc_82
.text:0000007C A4 A0 ADR R0, aAB ; "a>b"
.text:0000007E 06 F0 B7 F8 BL __2printf
.text:00000082 loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42 CMP R5, R4
.text:00000084 02 D1 BNE loc_8C
.text:00000086 A4 A0 ADR R0, aAB_0 ; "a==b"
.text:00000088 06 F0 B2 F8 BL __2printf
.text:0000008C loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42 CMP R5, R4
.text:0000008E 02 DA BGE locret_96
.text:00000090 A3 A0 ADR R0, aAB_1 ; "a<b"
.text:00000092 06 F0 AD F8 BL __2printf
.text:00000096 locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD POP {R4-R6,PC}
.text:00000096 ; End of function f_signed
```

Only B instructions in Thumb mode may be supplemented by condition codes, so the Thumb code looks more ordinary.

BLE is a normal conditional jump Less than or Equal, BNE—Not Equal, BGE—Greater than or Equal.

f_unsigned is similar, only other instructions are used while dealing with unsigned values: BLS (Unsigned lower or same) and BCS (Carry Set (Greater than or equal)).

**ARM64: Optimizing GCC (Linaro) 4.9**

---

**Listing 1.118: f_signed()**

```
f_signed:
    ; W0=a, W1=b
    cmp     w0, w1
    bgt     .L19 ; Branch if Greater Than (a>b)
    beq     .L20 ; Branch if Equal (a==b)
    bge     .L15 ; Branch if Greater than or Equal (a>=b) (impossible here)
    ; a<b
    adrp    x0, .LC11 ; "a<b"
    add     x0, x0, :lo12:.LC11
    b       puts
    .L19:
    adrp    x0, .LC9 ; "a>b"
    add     x0, x0, :lo12:.LC9
    b       puts
    .L15:    ; impossible to get here
    ret
    .L20:
    adrp    x0, .LC10 ; "a==b"
    add     x0, x0, :lo12:.LC10
    b       puts
```

**Listing 1.119: f_unsigned()**

```
f_unsigned:

```

---
The comments were added by the author of this book. What is striking is that the compiler is not aware that some conditions are not possible at all, so there is dead code at some places, which can never be executed.

Exercise

Try to optimize these functions manually for size, removing redundant instructions, without adding new ones.

MIPS

One distinctive MIPS feature is the absence of flags. Apparently, it was done to simplify the analysis of data dependencies.

There are instructions similar to SETcc in x86: SLT (“Set on Less Than”: signed version) and SLTU (unsigned version). These instructions sets destination register value to 1 if the condition is true or to 0 if otherwise. The destination register is then checked using BEQ (“Branch on Equal”) or BNE (“Branch on Not Equal”) and a jump may occur. So, this instruction pair has to be used in MIPS for comparison and branch. Let’s first start with the signed version of our function:

Listing 1.120: Non-optimizing GCC 4.4.5 (IDA)
`$v0=b; $v1=a`

---

```assembly
.text:00000004  sw   $ra, 0x20+var_4($sp)
.text:00000008  sw   $fp, 0x20+var_8($sp)
.text:0000000C  move $fp, $sp
.text:00000010  la   $gp, __gnu_local_gp
.text:00000018  sw   $gp, 0x20+var_10($sp)

; store input values into local stack:
.text:0000001C  sw   $a0, 0x20+arg_0($fp)
.text:00000020  sw   $a1, 0x20+arg_4($fp)

; reload them.
.text:00000024  lw   $v1, 0x20+arg_0($fp)
.text:00000028  lw   $v0, 0x20+arg_4($fp)

; $v0=b; $v1=a

.text:0000002C  or   $at, $zero ; NOP

; this is pseudoinstruction. in fact, "slt $v0,$v0,$v1" is there.
; so $v0 will be set to 1 if $v0<$v1 (b<a) or to 0 if otherwise:
.text:00000030  slt  $v0, $v1

; jump to loc_5c, if condition is not true.
; this is pseudoinstruction. in fact, "beq $v0,$zero,loc_5c" is there:
.text:00000034  beqz $v0, loc_5C

; print "a>b" and finish
.text:00000038  or   $at, $zero : branch delay slot, NOP

.text:0000003C  lui  $v0, (unk_230 >> 16) # "a>b"
.text:00000040  addiu $a0, $v0, (unk_230 & 0xFFFF) # "a>b"
.text:00000044  lw   $v0, (puts & 0xFFFF)($gp)
.text:00000048  or   $at, $zero ; NOP

.text:0000004C  move $t9, $v0
.text:00000050  jalr $t9
.text:00000054  or   $at, $zero ; branch delay slot, NOP

.text:00000058  lw   $gp, 0x20+var_10($fp)

.text:0000005C  loc_5C:

.text:0000005C  lw   $v1, 0x20+arg_0($fp)
.text:00000060  lw   $v0, 0x20+arg_4($fp)
.text:00000064  or   $at, $zero ; NOP

; check if $a==b, jump to loc_90 if its not true:
.text:00000068  bne  $v1, $v0, loc_90
.text:0000006C  or   $at, $zero ; branch delay slot, NOP

; condition is true, so print "a==b" and finish:
.text:00000070  lui  $v0, (aAB >> 16) # "a==b"
.text:00000074  addiu $a0, $v0, (aAB & 0xFFFF) # "a==b"
.text:00000078  lw   $v0, (puts & 0xFFFF)($gp)
.text:0000007C  or   $at, $zero ; NOP

.text:00000080  move $t9, $v0
.text:00000084  jalr $t9
.text:00000088  or   $at, $zero ; branch delay slot, NOP

.text:0000008C  lw   $gp, 0x20+var_10($fp)

.text:00000090  loc_90:

.text:00000090  lw   $v1, 0x20+arg_0($fp)
.text:00000094  lw   $v0, 0x20+arg_4($fp)
.text:00000098  or   $at, $zero ; NOP

; check if $v1<$v0 (a<b), set $v0 to 1 if condition is true:
.text:0000009C  slt  $v0, $v1, $v0

; if condition is not true (i.e., $v0==0), jump to loc_c8:
.text:000000A0  beqz $v0, loc_C8

.text:000000A4  or   $at, $zero ; branch delay slot, NOP

; condition is true, print "a=b" and finish:
.text:000000A8  lui  $v0, (aAB 0 >> 16) # "a=b"
.text:000000AC  addiu $a0, $v0, (aAB 0 & 0xFFFF) # "a=b"
.text:000000B0  lw   $v0, (puts & 0xFFFF)($gp)
.text:000000B4  or   $at, $zero ; NOP

.text:000000B8  move $t9, $v0
.text:000000BC  jalr $t9

.text:000000C0  or   $at, $zero ; branch delay slot, NOP

.text:000000C4  lw   $gp, 0x20+var_10($fp)

.text:000000C8  loc_C8:

.move  $sp, $fp

; all 3 conditions were false, so just finish:
.text:000000C8  loc_C8:
.move  $sp, $fp

# CODE XREF: f_signed+68
```

---

This is a segment of assembly code. The code appears to be checking for specific conditions and printing results based on those conditions. The comments in the code explain the purpose of each step. For example, the `beqz` instruction checks if `v0` is zero and jumps to `loc_5c` if true. The code uses various assembly instructions to manipulate registers, load and store values, and make conditional jumps based on the results of comparisons.
We also see there which are in fact the unsigned version is just the same, but
SLT REG0, REG0, REG1 is reduced by IDA to its shorter form:
SLT REG0, REG1.

We also see there BEQZ pseudo instruction (“Branch if Equal to Zero”),
which are in fact BEQ REG, $ZERO, LABEL.

The unsigned version is just the same, but SLTU (unsigned version, hence “U” in name) is used instead of
SLT:

Listing 1.121: Non-optimizing GCC 4.4.5 (IDA)
1.18.2 Calculating absolute value

A simple function:

```c
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
}
```

Optimizing MSVC

This is how the code is usually generated:

Listing 1.122: Optimizing MSVC 2012 x64

```assembly
i$ = 8
my_abs PROC
    ECX = input
    test ecx, ecx
    ; check for sign of input value
    ; skip NEG instruction if sign is positive
    jns SHORT $LN2@my_abs
    ; negate value
    neg ecx
$LN2@my_abs:
    ; prepare result in EAX:
    mov eax, ecx
    ret 0
my_abs ENDP
```

GCC 4.9 does mostly the same.

Optimizing Keil 6/2013: Thumb mode

Listing 1.123: Optimizing Keil 6/2013: Thumb mode

```assembly
my_abs PROC
    CMP r0,#0
    ; is input value equal to zero or greater than zero?
    ; skip RSBS instruction then
    BGE |L0.6|
    ; subtract input value from 0:
    RSBS r0,r0,#0
|L0.6|
    BX lr
ENDP
```
ARM lacks a negate instruction, so the Keil compiler uses the “Reverse Subtract” instruction, which just subtracts with reversed operands.

**Optimizing Keil 6/2013: ARM mode**

It is possible to add condition codes to some instructions in ARM mode, so that is what the Keil compiler does:

```
my_abs PROC
  CMP  r0,#0
  ; execute "Reverse Subtract" instruction only if input value is less than 0:
  RSBLT r0,r0,#0
  BX   lr
ENDP
```

Now there are no conditional jumps and this is good: 2.10.1 on page 462.

**Non-optimizing GCC 4.9 (ARM64)**

ARM64 has instruction NEG for negating:

```
my_abs: sub  sp, sp, #16
  str   w0, [sp,12]
  ldr   w0, [sp,12]
  ; compare input value with contents of WZR register
  ; (which always holds zero)
  cmp   w0, wzr
  bge   .L2
  neg   w0, w0
  b     .L3
.L2:    ldr   w0, [sp,12]
.L3:    add   sp, sp, 16
  ret
```

**MIPS**

```
my_abs:
  ; jump if $a0<0:
  bltz  $a0, locret_10
  ; just return input value ($a0) in $v0:
  move  $v0, $a0
  jr    $ra
  or    $at, $zero ; branch delay slot, NOP
locret_10:
  ; negate input value and store it in $v0:
  jr    $ra
  ; this is pseudoinstruction, in fact, this is "subu $v0,$zero,$a0" ($v0=0-$a0)
  negu  $v0, $a0
```

Here we see a new instruction: BLTZ (“Branch if Less Than Zero”).

There is also the NEGU pseudo instruction, which just does subtraction from zero. The “U” suffix in both SUBU and NEGU implies that no exception to be raised in case of integer overflow.

**Branchless version?**

You could have also a branchless version of this code. This we will review later: 3.16 on page 517.
1.18.3 Ternary conditional operator

The ternary conditional operator in C/C++ has the following syntax:

expression ? expression : expression

Here is an example:

```c
const char* f(int a)
{
    return a==10 ? "it is ten" : "it is not ten";
}
```

x86

Old and non-optimizing compilers generate assembly code just as if an if/else statement was used:

Listing 1.127: Non-optimizing MSVC 2008

```assembly
SG746 DB 'it is ten', 00H
SG747 DB 'it is not ten', 00H

v65 = -4 ; this will be used as a temporary variable
_a$ = 8
_f PROC
    push ebp
    mov ebp, esp
    push ecx
    ; compare input value with 10
    cmp DWORD PTR _a$[ebp], 10
    ; jump to $LN3@f if not equal
    jne SHORT $LN3@f
    ; store pointer to the string into temporary variable:
    mov DWORD PTR tv65[ebp], OFFSET SG746 ; 'it is ten'
    ; jump to exit
    jmp SHORT $LN4@f
$LN3@f:
    ; store pointer to the string into temporary variable:
    mov DWORD PTR tv65[ebp], OFFSET SG747 ; 'it is not ten'
$LN4@f:
    ; this is exit. copy pointer to the string from temporary variable to EAX.
    mov eax, DWORD PTR tv65[ebp]
    mov esp, ebp
    pop ebp
    ret 0
_f ENDP
```

Listing 1.128: Optimizing MSVC 2008

```assembly
SG792 DB 'it is ten', 00H
SG793 DB 'it is not ten', 00H

_a$ = 8 ; size = 4
_f PROC
    ; compare input value with 10
    cmp DWORD PTR _a$[esp-4], 10
    ; jump to $LN4@f if equal
    je SHORT $LN4@f
    ; store pointer to the string into temporary variable:
    mov eax, OFFSET SG792 ; 'it is ten'
    ; jump to exit
    mov eax, OFFSET SG793 ; 'it is not ten'
$LN4@f:
    ret 0
_f ENDP
```

Newer compilers are more concise:

Listing 1.129: Optimizing MSVC 2012 x64

```assembly
SG1355 DB 'it is ten', 00H
SG1356 DB 'it is not ten', 00H
```

142
$a = 8$

f PROC
; load pointers to the both strings
  lea rdx, OFFSET FLAT:$SG1355 ; 'it is ten'
  lea rax, OFFSET FLAT:$SG1356 ; 'it is not ten'
; compare input value with 10
  cmp ecx, 10
; if equal, copy value from RDX ("it is ten")
; if not, do nothing. pointer to the string "it is not ten" is still in RAX as for now.
  cmove rax, rdx
  ret
f ENDP

Optimizing GCC 4.8 for x86 also uses the CMOVcc instruction, while the non-optimizing GCC 4.8 uses conditional jumps.

**ARM**

Optimizing Keil for ARM mode also uses the conditional instructions ADRcc:

Listing 1.130: Optimizing Keil 6/2013 (ARM mode)

f PROC
; compare input value with 10
  CMP r0,#0xa
; if comparison result is EQual, copy pointer to the "it is ten" string into R0
  ADREQr0,[L0.16]; "it is ten"
; if comparison result is Not Equal, copy pointer to the "it is not ten" string into R0
  ADRNER0,[L0.28]; "it is not ten"
  BX lr
ENDP

|L0.16|
DCB "it is ten",0

|L0.28|
DCB "it is not ten",0

Without manual intervention, the two instructions ADREQ and ADRNE cannot be executed in the same run.

Optimizing Keil for Thumb mode needs to use conditional jump instructions, since there are no load instructions that support conditional flags:

Listing 1.131: Optimizing Keil 6/2013 (Thumb mode)

f PROC
; compare input value with 10
  CMP r0,#0xa
; jump to [L0.8] if EQual
  BEQ [L0.8]
  ADR r0,[L0.12]; "it is not ten"
  BX lr
  [L0.8]
  ADR r0,[L0.28]; "it is ten"
  BX lr
ENDP

|L0.12|
DCB "it is not ten",0

|L0.28|
DCB "it is ten",0

**ARM64**

Optimizing GCC (Linaro) 4.9 for ARM64 also uses conditional jumps:

Listing 1.132: Optimizing GCC (Linaro) 4.9

f:
  cmp x0, 10
That is because ARM64 does not have a simple load instruction with conditional flags, like ADRcc in 32-bit ARM mode or CMOVcc in x86.

It has, however, “Conditional SELECT” instruction (CSEL)[ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)p390, C5.5], but GCC 4.9 does not seem to be smart enough to use it in such piece of code.

**MIPS**

Unfortunately, GCC 4.4.5 for MIPS is not very smart, either:

```
Listing 1.133: Optimizing GCC 4.4.5 (assembly output)

$LCO: .ascii "it is not ten\000"
$LCL: .ascii "it is ten\000"
f:
    li $2,10  # 0xa
    ; compare a0 and 10, jump if equal:
    beq $4,$2,$L2
    nop ; branch delay slot

    ; leave address of "it is not ten" string in $v0 and return:
    lui $2,%hi($LCO)
    j $31
    addiu $2,$2,%lo($LCO)

$L2:
    ; leave address of "it is ten" string in $v0 and return:
    lui $2,%hi($LCL)
    j $31
    addiu $2,$2,%lo($LCL)
```

Let's rewrite it in an if/else way

```c
const char* f (int a)
{
    if (a==10)
        return "it is ten";
    else
        return "it is not ten";
};
```

Interestingly, optimizing GCC 4.8 for x86 was also able to use CMOVcc in this case:

```
Listing 1.134: Optimizing GCC 4.8

.LC0: .string "it is ten"
.LC1: .string "it is not ten"
f:
.LFB0:
; compare input value with 10
    cmp DWORD PTR [esp+4], 10
```


Optimizing Keil in ARM mode generates code identical to listing 1.130.
But the optimizing MSVC 2012 is not that good (yet).

Conclusion
Why optimizing compilers try to get rid of conditional jumps? Read here about it: 2.10.1 on page 462.

1.18.4 Getting minimal and maximal values

32-bit

```c
int my_max(int a, int b)
{
    if (a>b)
        return a;
    else
        return b;
};

int my_min(int a, int b)
{
    if (a<b)
        return a;
    else
        return b;
};
```

Listing 1.135: Non-optimizing MSVC 2013

```assembly
a$ = 8
_b$ = 12
_my_min PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    ; compare A and B:
    cmp eax, DWORD PTR _b$[ebp]
    ; jump if A is greater or equal to B:
    jge SHORT $LN2@my_min
    ; reload A to EAX if otherwise and jump to exit
    mov eax, DWORD PTR _a$[ebp]
    jmp SHORT $LN3@my_min
    jmp SHORT $LN3@my_min ; this is redundant JMP
$LN2@my_min:
    ; return B
    mov eax, DWORD PTR _b$[ebp]
$LN3@my_min:
    pop ebp
    ret 0
_my_min ENDP

_a$ = 8
_b$ = 12
_my_max PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    ; compare A and B:
    cmp eax, DWORD PTR _b$[ebp]
    ; jump if A is less or equal to B:
    jle SHORT $LN2@my_max
```

145
These two functions differ only in the conditional jump instruction: JGE ("Jump if Greater or Equal") is used in the first one and JLE ("Jump if Less or Equal") in the second.

There is one unnecessary JMP instruction in each function, which MSVC presumably left by mistake.

**Branchless**

ARM for Thumb mode reminds us of x86 code:

**Listing 1.136: Optimizing Keil 6/2013 (Thumb mode)**

```assembly
my_max PROC
; R0=A
; R1=B
; compare A and B:
    CMP    r0,r1
; branch if A is greater then B:
    BGT    L0.6
; otherwise (A<=B) return R1 (B):
    MOVS   r0,r1
L0.6:
; return
    BX     lr
ENDP

my_min PROC
; R0=A
; R1=B
; compare A and B:
    CMP    r0,r1
; branch if A is less then B:
    BLT    L0.14
; otherwise (A>=B) return R1 (B):
    MOVS   r0,r1
L0.14:
; return
    BX     lr
ENDP
```

The functions differ in the branching instruction: BGT and BLT. It’s possible to use conditional suffixes in ARM mode, so the code is shorter.

MOVcc is to be executed only if the condition is met:

**Listing 1.137: Optimizing Keil 6/2013 (ARM mode)**

```assembly
my_max PROC
; R0=A
; R1=B
; compare A and B:
    CMP    r0,r1
; return B instead of A by placing B in R0
; this instruction will trigger only if A<=B (hence, LE - Less or Equal)
; if instruction is not triggered (in case of A>B), A is still in R0 register
    MOVL   r0,r1
    BX     lr
ENDP
```
my_min PROC
; R0=A
; R1=B
; compare A and B:
CMP r0, r1
; return B instead of A by placing B in R0
; this instruction will trigger only if A>=B (hence, GE - Greater or Equal)
; if instruction is not triggered (in case of A<B), A value is still in R0 register
MOVGE r0, r1
BX lr
ENDP

Optimizing GCC 4.8.1 and optimizing MSVC 2013 can use CMPVcc instruction, which is analogous to MOVcc in ARM:

Listing 1.138: Optimizing MSVC 2013

my_max:
    mov edx, DWORD PTR [esp+4]
    mov eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; compare A and B:
    cmp edx, eax
    if A>=B, load A value into EAX
    the instruction idle if otherwise (if A<B)
    cmovge eax, edx
    ret

my_min:
    mov edx, DWORD PTR [esp+4]
    mov eax, DWORD PTR [esp+8]
; EDX=A
; EAX=B
; compare A and B:
    cmp edx, eax
    if A<=B, load A value into EAX
    the instruction idle if otherwise (if A>B)
    cmovle eax, edx
    ret

64-bit

#include <stdint.h>

int64_t my_max(int64_t a, int64_t b)
{
    if (a>b)
        return a;
    else
        return b;
};

int64_t my_min(int64_t a, int64_t b)
{
    if (a<b)
        return a;
    else
        return b;
};

There is some unneeded value shuffling, but the code is comprehensible:

Listing 1.139: Non-optimizing GCC 4.9.1 ARM64

my_max:
    sub sp, sp, #16
    str x0, [sp,8]
    str x1, [sp]
Branchless

No need to load function arguments from the stack, as they are already in the registers:

Listing 1.140: Optimizing GCC 4.9.1 x64

```assembly
my_max:
; RDI=A
; RSI=B
; compare A and B:
   cmp   rdi, rsi
; prepare B in RAX for return:
   mov   rax, rsi
; if A>=B, put A (RDI) in RAX for return.
; this instruction is idle if otherwise (if A<B)
   cmovge rax, rdi
ret

my_min:
; RDI=A
; RSI=B
; compare A and B:
   cmp   rdi, rsi
; prepare B in RAX for return:
   mov   rax, rsi
; if A<=B, put A (RDI) in RAX for return.
; this instruction is idle if otherwise (if A>B)
   cmovle rax, rdi
ret
```

MSVC 2013 does almost the same.

ARM64 has the CSEL instruction, which works just as MOVcc in ARM or CMOVcc in x86, just the name is different: “Conditional SESelect”.

Listing 1.141: Optimizing GCC 4.9.1 ARM64

```assembly
my_max:
; X0=A
; X1=B
; compare A and B:
```
MIPS

Unfortunately, GCC 4.4.5 for MIPS is not that good:

Listing 1.142: Optimizing GCC 4.4.5 (IDA)

my_max:
; set $v1 to 1 if $a1<$a0, or clear otherwise (if $a1>$a0):
  slt  $v1, $a1, $a0
; jump, if $v1 is 0 (or $a1>$a0):
  beqz $v1, locret_10
; this is branch delay slot
; prepare $a1 in $v0 in case of branch triggered:
  move $v0, $a1
; no branch triggered, prepare $a0 in $v0:
  move $v0, $a0
locret_10:
  jr   $ra
  or   $at, $zero ; branch delay slot, NOP

; the min() function is same, but input operands in SLT instruction are swapped:
my_min:
  slt  $v1, $a0, $a1
  beqz $v1, locret_28
  move $v0, $a1
  move $v0, $a0
locret_28:
  jr   $ra
  or   $at, $zero ; branch delay slot, NOP

Do not forget about the branch delay slots: the first MOVE is executed before BEQZ, the second MOVE is executed only if the branch hasn’t been taken.

1.18.5 Conclusion

x86

Here’s the rough skeleton of a conditional jump:

Listing 1.143: x86

CMP register, register/value
Jcc true ; cc=condition code
false:
  ... some code to be executed if comparison result is false ...
JMP exit
ture:
  ... some code to be executed if comparison result is true ...
exit:
ARM

Listing 1.144: ARM

```
CMP register, register/value
Bcc true; cc=condition code
false:
... some code to be executed if comparison result is false ...
JMP exit
true:
... some code to be executed if comparison result is true ...
exit:
```

MIPS

Listing 1.145: Check for zero

```
BEQZ REG, label
...
```

Listing 1.146: Check for less than zero using pseudoinstruction

```
BLTZ REG, label
...
```

Listing 1.147: Check for equal values

```
BEQ REG1, REG2, label
...
```

Listing 1.148: Check for non-equal values

```
BNE REG1, REG2, label
...
```

Listing 1.149: Check for less than (signed)

```
SLT REG1, REG2, REG3
BEQ REG1, label
...
```

Listing 1.150: Check for less than (unsigned)

```
SLTU REG1, REG2, REG3
BEQ REG1, label
...
```

Branchless

If the body of a condition statement is very short, the conditional move instruction can be used: MOVcc in ARM (in ARM mode), CSEL in ARM64, CMOVcc in x86.

ARM

It’s possible to use conditional suffixes in ARM mode for some instructions:

Listing 1.151: ARM (ARM mode)

```
CMP register, register/value
instr1_cc ; some instruction will be executed if condition code is true
instr2_cc ; some other instruction will be executed if other condition code is true
... etc...
```

Of course, there is no limit for the number of instructions with conditional code suffixes, as long as the CPU flags are not modified by any of them.

Thumb mode has the IT instruction, allowing to add conditional suffixes to the next four instructions. Read more about it: 1.25.7 on page 259.
1.18.6 Exercise

(ARM64) Try rewriting the code in listing 1.132 by removing all conditional jump instructions and using the CSEL instruction.

1.19 Software cracking

The vast majority of software can be cracked like that — by searching the very place where protection is checked, a dongle (8.8 on page 818), license key, serial number, etc.

Often, it looks like:

```plaintext
... call check_protection jz all_OK
call message_box_protection_missing call exit all_OK:
; proceed ...
```

So if you see a patch (or “crack”), that cracks a software, and that patch replaces 0x74/0x75 (JZ/JNZ) byte(s) by 0xEB (JMP), this is it.

The process of software cracking comes down to a search of that JMP.

There are also a cases, when a software checks protection from time to time, this can be a dongle, or a license server can be queried through the Internet. Then you have to look for a function that checks protection. Then to patch it, to put there xor eax, eax / retn, or mov eax, 1 / retn.

It’s important to understand that after patching of function beginning, usually, a garbage follows these two instructions. The garbage consists of part of one instruction and the several next instructions.

This is a real case. The beginning of a function which we want to replace by return 1;

<table>
<thead>
<tr>
<th>Listing 1.153: Before</th>
</tr>
</thead>
<tbody>
<tr>
<td>8BFF</td>
</tr>
<tr>
<td>55</td>
</tr>
<tr>
<td>8BEC</td>
</tr>
<tr>
<td>81EC68000000</td>
</tr>
<tr>
<td>A110C00001</td>
</tr>
<tr>
<td>33C5</td>
</tr>
<tr>
<td>8945FC</td>
</tr>
<tr>
<td>53</td>
</tr>
<tr>
<td>8B5D08</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>mov edi,edi</td>
</tr>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp,esp</td>
</tr>
<tr>
<td>sub esp,000000068</td>
</tr>
<tr>
<td>mov eax,[00100C010]</td>
</tr>
<tr>
<td>xor eax,ebp</td>
</tr>
<tr>
<td>mov [ebp][-4],eax</td>
</tr>
<tr>
<td>push ebx</td>
</tr>
<tr>
<td>mov ebx,[ebp][8]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 1.154: After</th>
</tr>
</thead>
<tbody>
<tr>
<td>8B01000000</td>
</tr>
<tr>
<td>C3</td>
</tr>
<tr>
<td>EC</td>
</tr>
<tr>
<td>6B08000000A1</td>
</tr>
<tr>
<td>10C0</td>
</tr>
<tr>
<td>0001</td>
</tr>
<tr>
<td>33C5</td>
</tr>
<tr>
<td>8945FC</td>
</tr>
<tr>
<td>53</td>
</tr>
<tr>
<td>mov eax,1</td>
</tr>
<tr>
<td>retn</td>
</tr>
<tr>
<td>in al,dx</td>
</tr>
<tr>
<td>push 0A1000008</td>
</tr>
<tr>
<td>adc al,al</td>
</tr>
<tr>
<td>add [ecx],al</td>
</tr>
<tr>
<td>xor eax,ebp</td>
</tr>
<tr>
<td>mov [ebp][-4],eax</td>
</tr>
<tr>
<td>push ebx</td>
</tr>
</tbody>
</table>
Several incorrect instructions appears — IN, PUSH, ADC, ADD, after which, Hiew disassembler (which I just used) synchronized and continued to disassemble all the rest.

This is not important — all these instructions followed RETN will never be executed, unless a direct jump would occur from some place, and that wouldn’t be possible in general case.

Also, a global boolean variable can be present, having a flag, was the software registered or not.

```assembly
init_etc proc
... call check_protection_or_license_file
mov is_demo, eax
... retn
init_etc endp

save_file proc
... mov eax, is_demo
cmp eax, 1
jz all_OK1

; all_OK1
; continue saving file
...
save_proc endp

somewhere_else proc
mov eax, is_demo
cmp eax, 1
jz all_OK

; all_OK2
; continue
somewhere_else endp
```

A beginning of the check_protection_or_license_file() function could be patched, so that it will always return 1, or, if this is better by some reason, all JZ/JNZ instructions can be patched as well.

More about patching: 11.1.

### 1.20 Impossible shutdown practical joke (Windows 7)

I don’t quite remember how I found the ExitWindowsEx() function in Windows 98’s (it was late 1990s) user32.dll file. Probably, I just spotted its self-describing name. And then I tried to block it by patching its beginning by 0xC3 byte (RETN).

The result was funny: Windows 98 cannot be shutted down anymore. Had to press reset button.

These days I tried to do the same in Windows 7, that was created almost 10 years later and based on completely different Windows NT base. Still, ExitWindowsEx() function present in user32.dll file and serves the same purpose.
First, I turned off *Windows File Protection* by adding this to registry (Windows would silently restore modified system files otherwise):

```plaintext
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon]
"SFCDisable"=dword:ffffff9d
```

Then I renamed `c:\windows\system32\user32.dll` to `user32.dll.bak`. I found `ExitWindowsEx()` export entry using Hiew (IDA can help as well) and put 0xC3 byte here. I restarted by Windows 7 and now it can’t be shutted down. “Restart” and “Logoff” buttons don’t work anymore.

I don’t know if it’s funny today or not, but back then, in late 1990s, my friend took patched user32.dll file on a floppy diskette and copied it to all the computers (within his reach, that worked under Windows 98 (almost all)) at his university. No Windows can be shutted down after and his computer science teacher was lurid. (Hopefully he can forgive us if he is reading this right now.)

If you do this, backup everything. The best idea is to run Windows under a virtual machine.

### 1.21 switch()/case/default

#### 1.21.1 Small number of cases

```c
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    }
};

int main()
{
    f (2); // test
};
```

#### x86

**Non-optimizing MSVC**

Result (MSVC 2010):

```
Listing 1.155: MSVC 2010

tv64 = -4 ; size = 4
_a$ = 8 ; size = 4
_f PROC
    push ebp
    mov ebp, esp
    push ecx
    mov eax, DWORD PTR _a$[ebp]
    mov DWORD PTR tv64[ebp], eax
    cmp DWORD PTR tv64[ebp], 0
    je SHORT $LN4@f
    cmp DWORD PTR tv64[ebp], 1
    je SHORT $LN3@f
    cmp DWORD PTR tv64[ebp], 2
    je SHORT $LN2@f
    jmp SHORT $LN1@f
$LN4@f:
    push OFFSET $SG739 ; 'zero', 0AH, 00H
    call _printf
```

153
Our function with a few cases in switch() is in fact analogous to this construction:

```c
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
}
```

If we work with switch() with a few cases it is impossible to be sure if it was a real switch() in the source code, or just a pack of if() statements.

This implies that switch() is like syntactic sugar for a large number of nested if().

There is nothing especially new to us in the generated code, with the exception of the compiler moving input variable `a` to a temporary local variable `tv64`.

If we compile this in GCC 4.4.1, we’ll get almost the same result, even with maximal optimization turned on (-O3 option).

### Optimizing MSVC

Now let’s turn on optimization in MSVC (/Ox): cl 1.c /Fa1.asm /Ox

Listing 1.156: MSVC

<table>
<thead>
<tr>
<th>a$ = 8 ; size = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>_f PROC</td>
</tr>
<tr>
<td>mov eax, DWORD PTR _a$[esp-4]</td>
</tr>
<tr>
<td>sub eax, 0</td>
</tr>
<tr>
<td>je SHORT $LN4@f</td>
</tr>
<tr>
<td>sub eax, 1</td>
</tr>
<tr>
<td>je SHORT $LN3@f</td>
</tr>
<tr>
<td>sub eax, 1</td>
</tr>
<tr>
<td>je SHORT $LN2@f</td>
</tr>
<tr>
<td>mov DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something unknown', 0aH, 00H</td>
</tr>
<tr>
<td>jmp _printf</td>
</tr>
<tr>
<td>$LN2@f:</td>
</tr>
<tr>
<td>mov DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H</td>
</tr>
<tr>
<td>jmp _printf</td>
</tr>
<tr>
<td>$LN3@f:</td>
</tr>
</tbody>
</table>

92Local variables in stack are prefixed with tv—that’s how MSVC names internal variables for its needs
Here we can see some dirty hacks.

First: the value of $a$ is placed in EAX and 0 is subtracted from it. Sounds absurd, but it is done to check if the value in EAX is 0. If yes, the ZF flag is to be set (e.g. subtracting from 0 is 0) and the first conditional jump JE (Jump if Equal or synonym JZ — Jump if Zero) is to be triggered and control flow is to be passed to the $LN4@f$ label, where the 'zero' message is being printed. If the first jump doesn't get triggered, 1 is subtracted from the input value and if at some stage the result is 0, the corresponding jump is to be triggered.

And if no jump gets triggered at all, the control flow passes to printf() with string argument 'something unknown'.

Second: we see something unusual for us: a string pointer is placed into the $a$ variable, and then printf() is called not via CALL, but via JMP. There is a simple explanation for that: the caller pushes a value to the stack and calls our function via CALL. CALL itself pushes the return address (RA) to the stack and does an unconditional jump to our function address. Our function at any point of execution (since it do not contain any instruction that moves the stack pointer) has the following stack layout:

- ESP—points to RA
- ESP+4—points to the $a$ variable

On the other side, when we have to call printf() here we need exactly the same stack layout, except for the first printf() argument, which needs to point to the string. And that is what our code does.

It replaces the function's first argument with the address of the string and jumps to printf(), as if we didn't call our function $f()$, but directly printf(). printf() prints a string to stdout and then executes the RET instruction, which POPs RA from the stack and control flow is returned not to $f()$ but rather to $f()$'s caller, bypassing the end of the $f()$ function.

All this is possible because printf() is called right at the end of the $f()$ function in all cases. In some way, it is similar to the longjmp() function. And of course, it is all done for the sake of speed.

A similar case with the ARM compiler is described in “printf() with several arguments” section, here (1.11.2 on page 53).

---

93 wikipedia
Since this example is tricky, let’s trace it in OllyDbg.

OllyDbg can detect such switch() constructs, and it can add some useful comments. EAX is 2 at the beginning, that’s the function’s input value:

![OllyDbg screenshot]

Figure 1.43: OllyDbg: EAX now contain the first (and only) function argument
0 is subtracted from 2 in EAX. Of course, EAX still contains 2. But the ZF flag is now 0, indicating that the resulting value is non-zero:

![CPU - main thread, module few](image)

Figure 1.44: OllyDbg: SUB executed
DEC is executed and EAX now contains 1. But 1 is non-zero, so the ZF flag is still 0:

![OllyDbg: first DEC executed](image)

**Figure 1.45: OllyDbg: first DEC executed**
Next DEC is executed. EAX is finally 0 and the ZF flag gets set, because the result is zero:

OllyDbg shows that this jump is to be taken now.
A pointer to the string “two” is to be written into the stack now:

Figure 1.47: OllyDbg: pointer to the string is to be written at the place of the first argument

Please note: the current argument of the function is 2 and 2 is now in the stack at the address 0x001EF850.
MOV writes the pointer to the string at address 0x001EF850 (see the stack window). Then, jump happens. This is the first instruction of the printf() function in MSVC100.DLL (This example was compiled with /MD switch):

![CPU - main thread, module MSVC100](image)

Figure 1.48: OllyDbg: first instruction of printf() in MSVC100.DLL

Now printf() treats the string at 0x00FF3010 as its only argument and prints the string.
This is the last instruction of `printf()`:

![CPU Register](image)

Top of stack (001F484C) = `001F4857`

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
<th>DataFrame - Cr</th>
<th>String</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>001F45A0</td>
<td>74 77 6F 6F 00 00 00 00 78 6F 63 74 00 00 00 00</td>
<td>unknown</td>
<td>two* screenin* g unknownE</td>
<td></td>
</tr>
<tr>
<td>001F45B0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>0 Hit here</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.49: OllyDbg: last instruction of `printf()` in MSVCR100.DLL

The string “two” has just been printed to the console window.
Now let's press F7 or F8 (step over) and return...not to f(), but rather to main():

Yes, the jump has been direct, from the guts of printf() to main(). Because RA in the stack points not to some place in f(), but rather to main(). And CALL 0x00FF1000 has been the actual instruction which called f().

---

**Figure 1.50: OllyDbg: return to main()**

Again, by investigating this code we cannot say if it was a switch() in the original source code, or just a pack of if() statements.

Anyway, we see here predicated instructions again (like ADREQ (Equal)) which is triggered only in case R0 = 0, and then loads the address of the string «zero
» into R0. The next instruction BEQ redirects control flow to loc_170, if R0 = 0.

An astute reader may ask, will BEQ trigger correctly since ADREQ it has already filled the R0 register before with another value?

Yes, it will since BEQ checks the flags set by the CMP instruction, and ADREQ does not modify any flags at all.

The rest of the instructions are already familiar to us. There is only one call to printf(), at the end, and we have already examined this trick here (1.11.2 on page 53). At the end, there are three paths to printf().
The last instruction, CMP R0, #2, is needed to check if \( a = 2 \).

If it is not true, then ADRNE loads a pointer to the string «something unknown \ln» into R0, since \( a \) has already been checked to be equal to 0 or 1, and we can sure that the \( a \) variable is not equal to these numbers at this point. And if \( R0 = 2 \), a pointer to the string «two\ln» will be loaded by ADREQ into R0.

**ARM: Optimizing Keil 6/2013 (Thumb mode)**

```assembly
.text:00000004  f1:   
.text:00000004 10 B5 PUSH {R4,LR} 
.text:00000006 00 28 CMP   R0, #0 
.text:00000008 05 D0 BEQ   zero_case 
.text:0000000A 01 28 CMP   R0, #1 
.text:0000000C 05 D0 BEQ   one_case 
.text:0000000E 02 28 CMP   R0, #2 
.text:0000000E 05 00 BEQ   two_case 
.text:0000000E 91 A0 ADR   R0, aSomethingUnkno ; "something unknown\n" 
.text:0000000E 04 E0 B   default_case 

.text:0000000E  zero_case: ; CODE XREF: f1+4 
.text:0000000E 95 A0 ADR   R0, aZero ; "zero\n" 
.text:0000000E 02 E0 B   default_case 

.text:0000000E  one_case: ; CODE XREF: f1+8 
.text:0000000E 96 A0 ADR   R0, aOne ; "one\n" 
.text:0000000E 02 E0 B   default_case 

.text:0000000E  two_case: ; CODE XREF: f1+C 
.text:0000000E 97 A0 ADR   R0, aTwo ; "two\n" 
.text:0000000E 02 F0 7E F8 BL   2printf 
.text:0000000F 06 F0 7E F8 BL   2printf 
.text:0000000F 10 BD POP   {R4,PC} 
```

As was already mentioned, it is not possible to add conditional predicates to most instructions in Thumb mode, so the Thumb-code here is somewhat similar to the easily understandable x86 CISC-style code.

**ARM64: Non-optimizing GCC (Linaro) 4.9**

```
.LC12:  .string "zero" 
.LC13:  .string "one" 
.LC14:  .string "two" 
.LC15:  .string "something unknown" 

f12:    stp x29, x30, [sp, -32]! 
   add x29, sp, 0 
   str w0, [x29,28] 
   ldr w0, [x29,28] 
   cmp w0, 1 
   beq .L34 
   cmp w0, 2 
   beq .L35 
   cmp w0, wzr 
   bne .L38 ; jump to default label 
   adrp x0, .LC12 ; "zero" 
   add x0, x0, :lo12:.LC12 
   b! 
   puts 
   .L32: 
   adrp x0, .LC13 ; "one" 
   add x0, x0, :lo12:.LC13 
   b! 
   puts 
   .L32: 
   adrp x0, .LC14 ; "two" 
```

164
The type of the input value is int, hence register W0 is used to hold it instead of the whole X0 register.

The string pointers are passed to puts() using an ADRP/ADD instructions pair just like it was demonstrated in the "Hello, world!" example: 1.5.3 on page 23.

**ARM64: Optimizing GCC (Linaro) 4.9**

```asm
add x0, x0, :lo12:.LC14
bl puts
b .L32
.L38:
    adrp x0, .LC15 ; "something unknown"
add x0, x0, :lo12:.LC15
bl puts
nop
.L32:
    ldp x29, x30, [sp], 32
ret
```

Better optimized piece of code. CBZ (Compare and Branch on Zero) instruction does jump if W0 is zero. There is also a direct jump to puts() instead of calling it, like it was explained before: 1.21.1 on page 154.

**MIPS**

**Listing 1.157: Optimizing GCC 4.4.5 (IDA)**

```asm
f:
    lui $gp, (__gnu_local_gp >> 16)  # is it 1?
    li $v0, 1
    beq $a0, $v0, loc_60
    la $gp, (__gnu_local_gp & 0xFFFF) ; branch delay slot
    or $at, $zero ; branch delay slot, NOP
    ; jump, if not equal to 0:
    bnez $a0, loc_38
    or $at, $zero ; branch delay slot, NOP
    ; zero case:
    lui $a0, ($LC0 >> 16) # "zero"
    lw $t9, (puts & 0xFFFF)($gp)
    or $at, $zero ; load delay slot, NOP
    jr $t9 ; branch delay slot, NOP
    la $a0, ($LC0 & 0xFFFF) # "zero" ; branch delay slot
```

165
The function always ends with calling puts(), so here we see a jump to puts() (JR: “Jump Register”) instead of “jump and link”. We talked about this earlier: 1.21.1 on page 154.

We also often see NOP instructions after LW ones. This is “load delay slot”: another delayslot in MIPS.

An instruction next to LW may execute at the moment while LW loads value from memory. However, the next instruction must not use the result of LW.

Modern MIPS CPUs have a feature to wait if the next instruction uses result of LW, so this is somewhat outdated, but GCC still adds NOPs for older MIPS CPUs. In general, it can be ignored.

Conclusion

A switch() with few cases is indistinguishable from an if/else construction, for example: listing.1.21.1.

1.21.2 A lot of cases

If a switch() statement contains a lot of cases, it is not very convenient for the compiler to emit too large code with a lot JE/JNE instructions.

```c
#include <stdio.h>

void f (int a)
{
    switch (a)
    {
        case 0: printf ("zero\n"); break;
        case 1: printf ("one\n"); break;
        case 2: printf ("two\n"); break;
        case 3: printf ("three\n"); break;
        case 4: printf ("four\n"); break;
        default: printf ("something unknown\n"); break;
    }
}

int main()
{
    f (2); // test
}
```

x86

Non-optimizing MSVC

We get (MSVC 2010):
What we see here is a set of `printf()` calls with various arguments. All they have not only addresses in the memory of the process, but also internal symbolic labels assigned by the compiler. All these labels are also mentioned in the `$LN11@f` internal table.

At the function start, if $a$ is greater than 4, control flow is passed to label `$LN1@f`, where `printf()` with argument 'something unknown' is called.

But if the value of $a$ is less or equals to 4, then it gets multiplied by 4 and added with the `$LN11@f` table address. That is how an address inside the table is constructed, pointing exactly to the element we need. For example, let’s say $a$ is equal to 2. $2 \times 4 = 8$ (all table elements are addresses in a 32-bit process and that is why all elements are 4 bytes wide). The address of the `$LN11@f` table + 8 is the table element where the `$LN4@f` label is stored. JMP fetches the `$LN4@f` address from the table and jumps to it.
This table is sometimes called jumptable or branch table\textsuperscript{94}.

Then the corresponding printf() is called with argument ‘two’.

Literally, the jmp DWORD PTR $LN11@f[ecx*4] instruction implies jump to the DWORD that is stored at address $LN11@f + ecx * 4.

npad (\emph{.1.7 on page 1024}) is an assembly language macro that align the next label so that it will be stored at an address aligned on a 4 bytes (or 16 bytes) boundary. This is very suitable for the processor since it is able to fetch 32-bit values from memory through the memory bus, cache memory, etc., in a more effective way if it is aligned.

\textsuperscript{94}The whole method was once called computed GOTO in early versions of Fortran: \url{wikipedia}. Not quite relevant these days, but what a term!
Let’s try this example in OllyDbg. The input value of the function (2) is loaded into EAX:

Figure 1.51: OllyDbg: function’s input value is loaded in EAX
The input value is checked, is it bigger than 4? If not, the “default” jump is not taken:

Figure 1.52: OllyDbg: 2 is no bigger than 4: no jump is taken
Here we see a jumptable:

Figure 1.53: OllyDbg: calculating destination address using jumptable

Here we’ve clicked “Follow in Dump” → “Address constant”, so now we see the jumptable in the data window. These are 5 32-bit values\(^{95}\). ECX is now 2, so the third element (can be indexed as 2\(^{96}\)) of the table is to be used. It’s also possible to click “Follow in Dump” → “Memory address” and OllyDbg will show the element addressed by the JMP instruction. That’s 0x010B103A.

\(^{95}\)They are underlined by OllyDbg because these are also FIXUPS: 6.5.2 on page 751, we are going to come back to them later

\(^{96}\)About indexing, see also: 3.22.3 on page 593

171
After the jump we are at 0x010B103A: the code printing “two” will now be executed:

![Image: CPU - main thread, module lot](image)

**Figure 1.54: OllyDbg: now we at the case: label**

**Non-optimizing GCC**

Let’s see what GCC 4.4.1 generates:

**Listing 1.159: GCC 4.4.1**

```assembly
public f
def f proc near ; CODE XREF: main+10
var_18 = dword ptr -18h
arg_0 = dword ptr 8

push ebp
mov ebp, esp
sub esp, 18h
cmp [ebp+arg_0], 4
ja short loc_8048444
mov eax, [ebp+arg_0]
shr eax, 2
mov eax, ds:off_804855C[eax]
jmp eax

loc_80483FE: ; DATA XREF: .rodata:off_804855C
mov [esp+18h+var_18], offset aZero ; "zero"
call _puts
jmp short locret_8048450

loc_804840C: ; DATA XREF: .rodata:08048560
mov [esp+18h+var_18], offset aOne ; "one"
call _puts
jmp short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
mov [esp+18h+var_18], offset aTwo ; "two"
call _puts
jmp short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
```

172
It is almost the same, with a little nuance: argument arg_0 is multiplied by 4 by shifting it to left by 2 bits (it is almost the same as multiplication by 4) (1.24.2 on page 216). Then the address of the label is taken from the off_804855C array, stored in EAX, and then JMP EAX does the actual jump.

**ARM: Optimizing Keil 6/2013 (ARM mode)**

Listing 1.160: Optimizing Keil 6/2013 (ARM mode)

```
000000174     f2
000000174 05 00 50 E3    CMP R0, #5 ; switch 5 cases
000000178 00 F1 8F 30 ADDCC PC, PC, R0, LSL#2 ; switch jump
00000017C 0E 00 00 EA   B  default_case ; jumptable 00000017B default case

000000180
000000180 loc_180 ; CODE XREF: f2+4
000000180 03 00 00 EA   B  zero_case ; jumptable 00000017B case 0

000000184
000000184 loc_184 ; CODE XREF: f2+4
000000184 04 00 00 EA   B  one_case ; jumptable 00000017B case 1

000000188
000000188 loc_188 ; CODE XREF: f2+4
000000188 05 00 00 EA   B  two_case ; jumptable 00000017B case 2

00000018C
00000018C loc_18C ; CODE XREF: f2+4
00000018C 06 00 00 EA   B  three_case ; jumptable 00000017B case 3

000000190
000000190 loc_190 ; CODE XREF: f2+4
000000190 07 00 00 EA   B  four_case ; jumptable 00000017B case 4

000000194
000000194 zero_case ; CODE XREF: f2+4
000000194 ; f2:loc_180
000000194 EC 00 8F E2 ADR R0, aZero ; jumptable 00000017B case 0
000000198 06 00 00 EA   B  loc_188

00000019C
00000019C one_case ; CODE XREF: f2+4
00000019C ; f2:loc_184
00000019C EC 00 8F E2 ADR R0, aOne ; jumptable 00000017B case 1
0000001A0 04 00 00 EA   B  loc_188
```
This code makes use of the ARM mode feature in which all instructions have a fixed size of 4 bytes.

Let's keep in mind that the maximum value for \( a \) is 4 and any greater value will cause «something unknown» string to be printed.

The first CMP R0, #5 instruction compares the input value of \( a \) with 5.

The next ADDCC PC, PC, R0, LSL#2 instruction is being executed only if \( R0 < 5 \) (CC=Carry clear / Less than). Consequently, if ADDCC does not trigger (it is a \( R0 \geq 5 \) case), a jump to default_case label will occur.

But if \( R0 < 5 \) and ADDCC triggers, the following is to be happen:

The value in R0 is multiplied by 4. In fact, LSL#2 at the instruction's suffix stands for “shift left by 2 bits”. But as we will see later (1.24.2 on page 216) in section “Shifts”, shift left by 2 bits is equivalent to multiplying by 4.

Then we add \( R0 \times 4 \) to the current value in PC, thus jumping to one of the B (Branch) instructions located below.

At the moment of the execution of ADDCC, the value in PC is 8 bytes ahead (0x180) than the address at which the ADDCC instruction is located (0x178), or, in other words, 2 instructions ahead.

This is how the pipeline in ARM processors works: when ADDCC is executed, the processor at the moment is beginning to process the instruction after the next one, so that is why PC points there. This has to be memorized.

If \( a = 0 \), then is to be added to the value in PC, and the actual value of the PC will be written into PC (which is 8 bytes ahead) and a jump to the label loc_180 will happen, which is 8 bytes ahead of the point where the ADDCC instruction is.

If \( a = 1 \), then \( PC + 8 + a \times 4 = PC + 8 + 1 \times 4 = PC + 12 = 0x184 \) will be written to PC, which is the address of the loc_184 label.

With every 1 added to \( a \), the resulting PC is increased by 4.

4 is the instruction length in ARM mode and also, the length of each B instruction, of which there are 5 in row.

Each of these five B instructions passes control further, to what was programmed in the switch() function.

Pointer loading of the corresponding string occurs there, etc.

\(^{97}\)ADD—addition
One cannot be sure that all instructions in Thumb and Thumb-2 modes have the same size. It can even be said that in these modes the instructions have variable lengths, just like in x86.

So there is a special table added that contains information about how much cases are there (not including default-case), and an offset for each with a label to which control must be passed in the corresponding case.
A special function is present here in order to deal with the table and pass control, named \_ARM\_common\_switch8\_thumb. It starts with BX PC, whose function is to switch the processor to ARM-mode. Then you see the function for table processing.

It is too advanced to describe it here now, so let’s omit it.

It is interesting to note that the function uses the LR register as a pointer to the table.

Indeed, after calling of this function, LR contains the address after BL \_ARM\_common\_switch8\_thumb instruction, where the table starts.

It is also worth noting that the code is generated as a separate function in order to reuse it, so the compiler doesn’t generate the same code for every switch() statement.

IDA successfully perceived it as a service function and a table, and added comments to the labels like jumptable 000000FA case 0.

**MIPS**

Listing 1.162: Optimizing GCC 4.4.5 (IDA)

```mips
f:
    lui $gp, (_gnu_local_gp >> 16)
    sltiu $v0, $a0, 5
    bnez $v0, loc_24
    la $gp, (_gnu_local_gp & 0xFFFF) ; branch delay slot

loc_24:
    # CODE XREF: f+8
    la $v0, off_120
    sll $a0, 2
    addu $a0, $v0, $a0
    la $a0, ($LC5 & 0xFFFF) # "something unknown" ; branch delay slot

sub_44:
    # DATA XREF: .rodata:0000012C
    lui $a0, ($LC3 >> 16) # "three"
    lw $t9, (puts & 0xFFFF)($gp)
    jr $t9
    la $a0, ($LC3 & 0xFFFF) # "three" ; branch delay slot

sub_58:
    # DATA XREF: .rodata:00000130
    lui $a0, ($LC4 >> 16) # "four"
    lw $t9, (puts & 0xFFFF)($gp)
    jr $t9
    la $a0, ($LC4 & 0xFFFF) # "four" ; branch delay slot

sub_6C:
    # DATA XREF: .rodata:off_120
    lui $a0, ($LC0 >> 16) # "zero"
    lw $t9, (puts & 0xFFFF)($gp)
    jr $t9
```

176
The new instruction for us is SLTIU (“Set on Less Than Immediate Unsigned”).

This is the same as SLTU (“Set on Less Than Unsigned”), but “I” stands for “immediate”, i.e., a number has to be specified in the instruction itself.

BNEZ is “Branch if Not Equal to Zero”.

Code is very close to the other ISAs. SLL (“Shift Word Left Logical”) does multiplication by 4.

MIPS is a 32-bit CPU after all, so all addresses in the jumptable are 32-bit ones.

Conclusion

Rough skeleton of switch():
The jump to the address in the jump table may also be implemented using this instruction: 
\texttt{JMP jump\_table[REG*4]}. Or \texttt{JMP jump\_table[REG*8]} in x64.

A \textit{jump table} is just array of pointers, like the one described later: \textit{1.26.5 on page 283}.

\subsection*{1.21.3 When there are several case statements in one block}

Here is a very widespread construction: several case statements for a single block:

```c
#include <stdio.h>

void f(int a)
{
    switch (a)
    {
    case 1:
    case 2:
    case 7:
    case 10:
        printf ("1, 2, 7, 10\n");
        break;
    case 3:
    case 4:
    case 5:
    case 6:
        printf ("3, 4, 5\n");
        break;
    case 8:
    case 9:
    case 20:
    case 21:
        printf ("8, 9, 21\n");
        break;
    case 22:
        printf ("22\n");
        break;
    default:
        printf ("default\n");
        break;
    }
}

int main()
{
    f(4);
}
```

It’s too wasteful to generate a block for each possible case, so what is usually done is to generate each block plus some kind of dispatcher.

\textbf{MSVC}

Listing 1.164: Optimizing MSVC 2010

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$SG2798 DB '1, 2, 7, 10', 0aH, 00H</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$SG2800 DB '3, 4, 5', 0aH, 00H</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$SG2802 DB '8, 9, 21', 0aH, 00H</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$SG2804 DB '22', 0aH, 00H</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$SG2806 DB 'default', 0aH, 00H</td>
<td></td>
</tr>
</tbody>
</table>
We see two tables here: the first table ($LN10@f$) is an index table, and the second one ($LN11@f$) is an array of pointers to blocks.

First, the input value is used as an index in the index table (line 13).

Here is a short legend for the values in the table: 0 is the first case block (for values 1, 2, 7, 10), 1 is the second one (for values 3, 4, 5), 2 is the third one (for values 8, 9, 21), 3 is the fourth one (for value 22), 4 is for the default block.

There we get an index for the second table of code pointers and we jump to it (line 14).

What is also worth noting is that there is no case for input value 0.

That’s why we see the DEC instruction at line 10, and the table starts at $a = 1$, because there is no need to allocate a table element for $a = 0$. 

---

_We see two tables here: the first table ($LN10@f$) is an index table, and the second one ($LN11@f$) is an array of pointers to blocks._

_First, the input value is used as an index in the index table (line 13)._

_Here is a short legend for the values in the table: 0 is the first case block (for values 1, 2, 7, 10), 1 is the second one (for values 3, 4, 5), 2 is the third one (for values 8, 9, 21), 3 is the fourth one (for value 22), 4 is for the default block._

_There we get an index for the second table of code pointers and we jump to it (line 14)._

_What is also worth noting is that there is no case for input value 0._

_That’s why we see the DEC instruction at line 10, and the table starts at $a = 1$, because there is no need to allocate a table element for $a = 0$. _
This is a very widespread pattern.
So why is this economical? Why isn’t it possible to make it as before (1.21.2 on page 172), just with one table consisting of block pointers? The reason is that the elements in index table are 8-bit, hence it’s all more compact.

**GCC**

GCC does the job in the way we already discussed (1.21.2 on page 172), using just one table of pointers.

**ARM64: Optimizing GCC 4.9.1**

There is no code to be triggered if the input value is 0, so GCC tries to make the jump table more compact and so it starts at 1 as an input value.

GCC 4.9.1 for ARM64 uses an even cleverer trick. It’s able to encode all offsets as 8-bit bytes.

Let’s recall that all ARM64 instructions have a size of 4 bytes.

GCC is uses the fact that all offsets in my tiny example are in close proximity to each other. So the jump table consisting of single bytes.

```
Listing 1.165: Optimizing GCC 4.9.1 ARM64

f14: 1
; input value in W0
    sub    w0, w0, #1
    cmp    w0, 21
; branch if less or equal (unsigned):
    bls    .L9
.L2:
; print "default":
    adrp   x0, .LC4
    add    x0, x0, :lo12:.LC4
    b      puts
.L9:
; load jump table address to X1:
    adrp   x1, .L4
    add    x1, x1, :lo12:.L4
; W0=input_value-1
; load byte from the table:
    ldrb   w0, [x1,w0,uxtw]
; load address of the Lrtx label:
    adr    x1, .Lrtx4
; multiply table element by 4 (by shifting 2 bits left) and add (or subtract) to the address of Lrtx:
    add    x0, x1, w0, sxtb #2
; jump to the calculated address:
    br     x0
; this label is pointing in code (text) segment:
.Lrtx4:
    .section   .rodata
; everything after ".section" statement is allocated in the read-only data (rodata) segment:
.L4:
    .byte   (.L3 - .Lrtx4) / 4 ; case 1
    .byte   (.L3 - .Lrtx4) / 4 ; case 2
    .byte   (.L5 - .Lrtx4) / 4 ; case 3
    .byte   (.L5 - .Lrtx4) / 4 ; case 4
    .byte   (.L5 - .Lrtx4) / 4 ; case 5
    .byte   (.L5 - .Lrtx4) / 4 ; case 6
    .byte   (.L3 - .Lrtx4) / 4 ; case 7
    .byte   (.L6 - .Lrtx4) / 4 ; case 8
    .byte   (.L6 - .Lrtx4) / 4 ; case 9
    .byte   (.L3 - .Lrtx4) / 4 ; case 10
    .byte   (.L2 - .Lrtx4) / 4 ; case 11
    .byte   (.L2 - .Lrtx4) / 4 ; case 12
    .byte   (.L2 - .Lrtx4) / 4 ; case 13
    .byte   (.L2 - .Lrtx4) / 4 ; case 14
    .byte   (.L2 - .Lrtx4) / 4 ; case 15
    .byte   (.L2 - .Lrtx4) / 4 ; case 16
    .byte   (.L2 - .Lrtx4) / 4 ; case 17
```
Let's compile this example to object file and open it in IDA. Here is the jump table:

### Listing 1.166: jumptable in IDA

```assembly
.rodata:0000000000000064 AREA .rodata, DATA, READONLY
.rodata:0000000000000064 $d DCB 9 ; case 1
.rodata:0000000000000064 $d DCB 9 ; case 2
.rodata:0000000000000064 $d DCB 6 ; case 3
.rodata:0000000000000064 $d DCB 6 ; case 4
.rodata:0000000000000064 $d DCB 6 ; case 5
.rodata:0000000000000064 $d DCB 6 ; case 6
.rodata:0000000000000064 $d DCB 6 ; case 7
.rodata:0000000000000064 $d DCB 6 ; case 8
.rodata:0000000000000064 $d DCB 6 ; case 9
.rodata:0000000000000064 $d DCB 6 ; case 10
.rodata:0000000000000064 $d DCB 6 ; case 11
.rodata:0000000000000064 $d DCB 6 ; case 12
.rodata:0000000000000064 $d DCB 6 ; case 13
.rodata:0000000000000064 $d DCB 6 ; case 14
.rodata:0000000000000064 $d DCB 6 ; case 15
.rodata:0000000000000064 $d DCB 6 ; case 16
.rodata:0000000000000064 $d DCB 6 ; case 17
.rodata:0000000000000064 $d DCB 6 ; case 18
.rodata:0000000000000064 $d DCB 6 ; case 19
.rodata:0000000000000064 $d DCB 6 ; case 20
.rodata:0000000000000064 $d DCB 6 ; case 21
.rodata:0000000000000064 $d DCB 6 ; case 22
.rodata:0000000000000064 $d DCB 6 ; .rodata ends
```

So in case of 1, 9 is to be multiplied by 4 and added to the address of Lrtx4 label.
In case of 22, 0 is to be multiplied by 4, resulting in 0.
Right after the Lrtx4 label is the L7 label, where you can find the code that prints “22”.

There is no jump table in the code segment, it’s allocated in a separate .rodata section (there is no special necessity to place it in the code section).

There are also negative bytes (0xF7), they are used for jumping back to the code that prints the “default” string (at .L2).

### 1.21.4 Fall-through

Another popular usage of switch() operator is so-called “fallthrough”. Here is simple example:

```c
bool is_whitespace(char c) {
    switch (c) {
    case ' ': // fallthrough
    case '\t': // fallthrough
    case '\r': // fallthrough
    case '\n':
        return true;
    default: // not whitespace
        return false;
    }
}
```

Slightly harder, from Linux kernel:

```c
char nco1, nco2;

void f(int if_freq_khz) {
    switch (if_freq_khz) {
    default:
        printf("IF=%d KHz is not supported, 3250 assumed\n", if_freq_khz);
        /* fallthrough */
    case 3250: /* 3.25Mhz */
        nco1 = 0x34;
        nco2 = 0x00;
        break;
    case 3500: /* 3.50Mhz */
        nco1 = 0x38;
        nco2 = 0x00;
        break;
    case 4000: /* 4.00Mhz */
        nco1 = 0x40;
        nco2 = 0x00;
        break;
    case 5000: /* 5.00Mhz */
        nco1 = 0x50;
        nco2 = 0x00;
        break;
    case 5380: /* 5.38Mhz */
        nco1 = 0x56;
        nco2 = 0x14;
        break;
    }
}
```

Listing 1.167: Optimizing GCC 5.4.0 x86

```
.LC0:
.string "IF=%d KHz is not supported, 3250 assumed\n"

f:
    sub    esp, 12
    mov    eax, DWORD PTR [esp+16]
    cmp    eax, 4000
    je     .L3
```

---

98 Copypasted from https://github.com/azonalon/prgraas/blob/master/prog1lib/lecture_examples/is_whitespace.c
99 Copypasted from https://github.com/torvalds/linux/blob/master/drivers/media/dvb-frontends/lgdt3306a.c
We can get to .L5 label if there is number 3250 at function’s input. But we can get to this label from the other side: we see that there are no jumps between printf() call and .L5 label.

Now we can understand why switch() statement is sometimes a source of bugs: one forgotten break will transform your switch() statement into fallthrough one, and several blocks will be executed instead of single one.

1.21.5 Exercises

Exercise#1

It’s possible to rework the C example in 1.21.2 on page 166 in such way that the compiler can produce even smaller code, but will work just the same. Try to achieve it.

1.22 Loops

1.22.1 Simple example

x86

There is a special LOOP instruction in x86 instruction set for checking the value in register ECX and if it is not 0, to decrement ECX and pass control flow to the label in the LOOP operand. Probably this instruction is not very convenient, and there are no any modern compilers which emit it automatically. So, if you see this instruction somewhere in code, it is most likely that this is a manually written piece of assembly code.

In C/C++ loops are usually constructed using for(), while() or do/while() statements. Let's start with for().
This statement defines loop initialization (set loop counter to initial value), loop condition (is the counter bigger than a limit?), what is performed at each iteration (increment/decrement) and of course loop body.

```c
for (initialization; condition; at each iteration)
{
    loop_body;
}
```

The generated code is consisting of four parts as well.

Let's start with a simple example:

```c
#include <stdio.h>

void printing_function(int i)
{
    printf("f(%d)\n", i);
};

int main()
{
    int i;
    for (i=2; i<10; i++)
    {
        printing_function(i);
    }
    return 0;
};
```

The result (MSVC 2010):

```assembly
_i$ = -4
_main PROC
    push ebp
    mov ebp, esp
    push ecx
    mov DWORD PTR _i$[ebp], 2 ; loop initialization
    jmp SHORT $LN3@main
$LN2@main:
    mov eax, DWORD PTR _i$[ebp]; here is what we do after each iteration:
    add eax, 1 ; add 1 to (i) value
    mov DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp DWORD PTR _i$[ebp], 10 ; this condition is checked before each iteration
    jge SHORT $LN1@main
    mov ecx, DWORD PTR _i$[ebp]; loop body: call printing_function(i)
    push ecx
    call _printing_function
    add esp, 4
    jmp SHORT $LN2@main ; jump to loop begin
$LN1@main:
    ; loop end
    xor eax, eax
    pop ebp
    ret 0
_main ENDP
```

As we see, nothing special.

GCC 4.4.1 emits almost the same code, with one subtle difference:

```assembly
main proc near
    var_20 = dword ptr -20h
    var_4 = dword ptr -4
    push ebp
    mov ebp, esp
```

184
and esp, 0FFFFFFF0h
sub esp, 20h
mov [esp+20h+var_4], 2 ; (i) initializing
jmp short loc_8048476

loc_8048465:
mov eax, [esp+20h+var_4]
mov [esp+20h+var_20], eax
call printing_function
add [esp+20h+var_4], 1 ; (i) increment

loc_8048476:
 cmp [esp+20h+var_4], 9
 jle short loc_8048465 ; if i<=9, continue loop
mov eax, 0
leave
ret
main endp

Now let's see what we get with optimization turned on (/Ox):

Listing 1.170: Optimizing MSVC

main PROC
 push esi
 mov esi, 2
$LL3@main:
push esi
call _printing_function
inc esi
add esp, 4
cmp esi, 10 ; 0000000ah
 jl SHORT $LL3@main
xor eax, eax
pop esi
ret 0
main ENDP

What happens here is that space for the i variable is not allocated in the local stack anymore, but uses an individual register for it, ESI. This is possible in such small functions where there aren't many local variables.

One very important thing is that the f() function must not change the value in ESI. Our compiler is sure here. And if the compiler decides to use the ESI register in f() too, its value would have to be saved at the function's prologue and restored at the function’s epilogue, almost like in our listing: please note PUSH ESI/POP ESI at the function start and end.

Let's try GCC 4.4.1 with maximal optimization turned on (-O3 option):

Listing 1.171: Optimizing GCC 4.4.1

main proc near
var_10 = dword ptr -10h
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 10h
mov [esp+10h+var_10], 2
call printing_function
mov [esp+10h+var_10], 3
call printing_function
mov [esp+10h+var_10], 4
call printing_function
mov [esp+10h+var_10], 5
call printing_function
mov [esp+10h+var_10], 6
call printing_function
mov [esp+10h+var_10], 7
call printing_function

mov [esp+10h+var_10], 8
call printing_function
mov [esp+10h+var_10], 9
call printing_function
xor eax, eax
leave
retn

main endp

Huh, GCC just unwound our loop.

Loop unwinding has an advantage in the cases when there aren’t much iterations and we could cut some execution time by removing all loop support instructions. On the other side, the resulting code is obviously larger.

Big unrolled loops are not recommended in modern times, because bigger functions may require bigger cache footprint\[100].

OK, let’s increase the maximum value of the \( i \) variable to 100 and try again. GCC does:

Listing 1.172: GCC

public main
main proc near
var_20 = dword ptr -20h
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
push ebx
mov ebx, 2 ; i=2
sub esp, 1Ch
; aligning label loc_80484D0 (loop body begin) by 16-byte border:
 nop
loc_80484D0:
; pass (i) as first argument to printing_function():
 mov [esp+20h+var_20], ebx
 add ebx, 1 ; i++
 call printing_function
 cmp ebx, 64h ; i==100?
 jnz short loc_80484D0 ; if not, continue
 add esp, 1Ch
 xor eax, eax ; return 0
 pop ebx
 mov esp, ebp
 pop ebp
 retn
main endp

It is quite similar to what MSVC 2010 with optimization (/Ox) produce, with the exception that the EBX register is allocated for the \( i \) variable.

GCC is sure this register will not be modified inside of the f() function, and if it will, it will be saved at the function prologue and restored at epilogue, just like here in the main() function.

\[100\] A very good article about it: [Ulrich Drepper, What Every Programmer Should Know About Memory, (2007)]\[101\]. Another recommendations about loop unrolling from Intel are here: [Intel® 64 and IA-32 Architectures Optimization Reference Manual, (2014)3.4.1.7].
Let's compile our example in MSVC 2010 with `/Ox` and `/Ob0` options and load it into OllyDbg.

It seems that OllyDbg is able to detect simple loops and show them in square brackets, for convenience:

By tracing (F8 — step over) we see ESI incrementing. Here, for instance, ESI = i = 6:

9 is the last loop value. That's why JL is not triggering after the increment, and the function will finish:
As we might see, it is not very convenient to trace manually in the debugger. That’s a reason we will try tracer.

We open compiled example in IDA, find the address of the instruction PUSH ESI (passing the sole argument to f()), which is 0x401026 for this case and we run the tracer:

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX just sets a breakpoint at the address and tracer will then print the state of the registers.

In the tracer.log, this is what we see:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
```

Figure 1.57: OllyDbg: $ESI = 10$, loop end

x86: tracer
We see how the value of ESI register changes from 2 to 9.

Even more than that, the tracer can collect register values for all addresses within the function. This is called trace there. Every instruction gets traced, all interesting register values are recorded.

Then, an IDA.idc-script is generated, that adds comments. So, in the IDA we’ve learned that the main() function address is 0x00401020 and we run:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF stands for set breakpoint on function.

As a result, we get the loops_2.exe.idc and loops_2.exe_clear.idc scripts.
We load loops_2.exe.idc into IDA and see:

We see that ESI can be from 2 to 9 at the start of the loop body, but from 3 to 0xA (10) after the increment. We can also see that main() is finishing with 0 in EAX. tracer also generates loops_2.exe.txt, that contains information about how many times each instruction has been executed and register values:

<table>
<thead>
<tr>
<th>Address</th>
<th>Execution Count</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x401020</td>
<td>1</td>
<td>[PUSH ESI] ESI=1</td>
</tr>
<tr>
<td>0x401021</td>
<td>1</td>
<td>[MOV ESI, 2]</td>
</tr>
<tr>
<td>0x401026</td>
<td>8</td>
<td>[PUSH ESI] ESI=2..9</td>
</tr>
<tr>
<td>0x401027</td>
<td>8</td>
<td>[CALL 8D1000h] tracing nested maximum level (1) reached, \ skipping this CALL 8D1000h=0x8d1000</td>
</tr>
<tr>
<td>0x40102c</td>
<td>8</td>
<td>[INC ESI] ESI=2..9</td>
</tr>
<tr>
<td>0x40102d</td>
<td>8</td>
<td>[ADD ESP, 4] ESP=0x30fcbc</td>
</tr>
<tr>
<td>0x401030</td>
<td>8</td>
<td>[CMP ESI, 0Ah] ESI=3..0xa</td>
</tr>
<tr>
<td>0x401033</td>
<td>8</td>
<td>[JL short loc_401026] SF=false,true OF=false</td>
</tr>
<tr>
<td>0x401035</td>
<td>1</td>
<td>[XOR EAX, EAX]</td>
</tr>
<tr>
<td>0x401037</td>
<td>1</td>
<td>[POP ESI]</td>
</tr>
<tr>
<td>0x401038</td>
<td>1</td>
<td>[RETN] EAX=0</td>
</tr>
</tbody>
</table>

We can use grep here.

**ARM**

Non-optimizing Keil 6/2013 (ARM mode)

```
main
  STMFD SP!, {R4,LR}
  MOV R4, #2
  B loc_368

loc_35c ; CODE XREF: main+1c
  MOV R0, R4
  BL printing_function
  ADD R4, R4, #1

loc_368 ; CODE XREF: main+8
  CMP R4, #0xa
```
Iteration counter $i$ is to be stored in the R4 register. The MOV R4, #2 instruction just initializes $i$. The MOV R0, R4 and BL printing_function instructions compose the body of the loop, the first instruction preparing the argument for $f()$ function and the second calling the function. The ADD R4, R4, #1 instruction just adds 1 to the $i$ variable at each iteration. CMP R4, #0xA compares $i$ with 0xA (10). The next instruction BLT (Branch Less Than) jumps if $i$ is less than 10. Otherwise, 0 is to be written into R0 (since our function returns 0) and function execution finishes.

**Optimizing Keil 6/2013 (Thumb mode)**

```
_iter
PUSH    {R4,LR}
MOV     R4, #2
loc_132 ; CODE XREF: _iter+E
MOV     R0, R4
BL      printing_function
ADDS    R4, R4, #1
CMP     R4, #0xA
BLT     loc_132
MOV     R0, #0
POP     {R4,PC}
```

Practically the same.

**Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)**

```
_iter
PUSH    {R4,R7,LR}
MOVW    R4, #0x1124 ; "%d
"
MOV     R1, #2
MOVT.W  R4, #0
ADD     R7, SP, #4
ADD     R4, PC
MOV     R0, R4
BLX     _printf
MOV     R0, R4
MOV     R1, #3
BLX     _printf
MOV     R0, R4
MOV     R1, #4
BLX     _printf
MOV     R0, R4
MOV     R1, #5
BLX     _printf
MOV     R0, R4
MOV     R1, #6
BLX     _printf
MOV     R0, R4
MOV     R1, #7
BLX     _printf
MOV     R0, R4
MOV     R1, #8
BLX     _printf
MOV     R0, R4
MOV     R1, #9
BLX     _printf
MOV     R0, #0
POP     {R4,R7,PC}
```

In fact, this was in my $f()$ function:
void printing_function(int i) {
    printf ("%d\n", i);
};

So, LLVM not just unrolled the loop, but also inlined my very simple function f(), and inserted its body 8 times instead of calling it.

This is possible when the function is so simple (like mine) and when it is not called too much (like here).

**ARM64: Optimizing GCC 4.9.1**

Listing 1.174: Optimizing GCC 4.9.1

printing_function:
; prepare second argument of printf():
mov w1, w0
; load address of the "f(%d)\n" string
adrp x0, .LC0
add x0, x0, :lo12:.LC0
; just branch here instead of branch with link and return:
b  printf

main:
; save FP and LR in the local stack:
stp x29, x30, [sp, -32]!
; set up stack frame:
add x29, sp, 0
; save contents of X19 register in the local stack:
str x19, [sp,16]
; we will use W19 register as counter.
; set initial value of 2 to it:
mov w19, 2
.L3:
; prepare first argument of printing_function():
mov w0, w19
; increment counter register.
add w19, w19, 1
; W0 here still holds value of counter value before increment.
bl printing_function
; is it end?
cmp w19, 10
bne .L3
; no, jump to the loop body begin:
bne .L3
; return 0
mov w0, 0
; restore contents of X19 register:
ldr x19, [sp,16]
; restore FP and LR values:
ldp x29, x30, [sp], 32
ret
.LC0:
.string "f(%d)\n"

**ARM64: Non-optimizing GCC 4.9.1**

Listing 1.175: Non-optimizing GCC 4.9.1 -fno-inline

.LC0:
.string "f(%d)\n"

ing function:
; save FP and LR in the local stack:
stp x29, x30, [sp, -32]!
; set up stack frame:
add x29, sp, 0
; save contents of W0 register:
str w0, [x29,28]
; load address of the "f(%d)\n" string
adr p x0, .LC0
add x0, x0, :lo12:.LC0
; reload input value from the local stack to W1 register:
ldr w1, [x29,28]
; call printf()
bl printf
; restore FP and LR values:
ldp x29, x30, [sp], 32
ret
main:
; save FP and LR in the local stack:
stp x29, x30, [sp, -32]!
; set up stack frame:
add x29, sp, 0
; initialize counter
mov w0, 2
; store it to the place allocated for it in the local stack:
str w0, [x29,28]
; skip loop body and jump to the loop condition check instructions:
b L3
.L4:
; load counter value to W0.
; it will be the first argument of printing_function():
ldr w0, [x29,28]
; call printing_function():
bl printing_function
; increment counter value:
ldr w0, [x29,28]
add w0, w0, 1
str w0, [x29,28]
.L3:
; loop condition check.
; load counter value:
ldr w0, [x29,28]
; is it 9?
cmp w0, 9
; less or equal? then jump to loop body begin:
; do nothing otherwise.
b le L4
; return 0
mov w0, 0
; restore FP and LR values:
ldp x29, x30, [sp], 32
ret

MIPS

Listing 1.176: Non-optimizing GCC 4.4.5 (IDA)

main:
; IDA is not aware of variable names in local stack
; We gave them names manually:
i = -0x10
saved_FP = -8
saved_RA = -4
; function prologue:
addiu $sp, -0x28
sw $ra, 0x28+saved_RA($sp)
sw $fp, 0x28+saved_FP($sp)
move $fp, $sp
; initialize counter at 2 and store this value in local stack
li $v0, 2
sw $v0, 0x28+i($fp)
; pseudoinstruction. "BEQ $ZERO, $ZERO, loc_9C" there in fact:
b loc_9C
or $at, $zero ; branch delay slot, NOP

The instruction that’s new to us is B. It is actually the pseudo instruction (BEQ).

One more thing

In the generated code we can see: after initializing i, the body of the loop is not to be executed, as the condition for i is checked first, and only after that loop body can be executed. And that is correct.

Because, if the loop condition is not met at the beginning, the body of the loop must not be executed. This is possible in the following case:

```c
for (i=0; i<total_entries_to_process; i++)
    loop_body;
```

If total_entries_to_process is 0, the body of the loop must not be executed at all.

This is why the condition checked before the execution.

However, an optimizing compiler may swap the condition check and loop body, if it sure that the situation described here is not possible (like in the case of our very simple example and using compilers like Keil, Xcode (LLVM), MSVC in optimization mode).

1.22.2 Memory blocks copying routine

Real-world memory copy routines may copy 4 or 8 bytes at each iteration, use SIMD\footnote{Single Instruction, Multiple Data}, vectorization, etc. But for the sake of simplicity, this example is the simplest possible.

```c
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
}
```
**Straight-forward implementation**

Listing 1.177: GCC 4.9 x64 optimized for size (-Os)

```c
my_memcpyp:
    ; RDI = destination address
    ; RSI = source address
    ; RDX = size of block

    ; initialize counter (i) at 0
    xor    eax, eax
.L2:
    ; all bytes copied? exit then:
    cmp     rax, rdx
    je     .L5
    load byte at RSI+i:
    mov     cl, BYTE PTR [rsi+rax]
    store byte at RDI+i:
    mov     BYTE PTR [rdi+rax], cl
    inc     rax ; i++
    jmp     .L2
.L5:
    ret
```

Listing 1.178: GCC 4.9 ARM64 optimized for size (-Os)

```c
my_memcpy:
    ; X0 = destination address
    ; X1 = source address
    ; X2 = size of block

    ; initialize counter (i) at 0
    mov     x3, 0
.L2:
    ; all bytes copied? exit then:
    cmp     x3, x2
    beq    .L5
    load byte at X1+i:
    ldrb    w4, [x1,x3]
    store byte at X0+i:
    strb    w4, [x0,x3]
    add     x3, x3, 1 ; i++
    b      .L2
.L5:
    ret
```

Listing 1.179: Optimizing Keil 6/2013 (Thumb mode)

```c
my_memcpy PROC
    ; R0 = destination address
    ; R1 = source address
    ; R2 = size of block

    ; initialize counter (i) at 0
    MOVVS  r3,#0
    ; condition checked at the end of function, so jump there:
    B      |L0.12|
|L0.6|
    ; load byte at R1+i:
    LDRB    r4,[r1,r3]
    ; store byte at R0+i:
    STRB    r4,[r0,r3]
    ; i++
    ADDS    r3,r3,#1
|L0.12|
    ; i<size?
    CMP     r3,r2
    ; jump to the loop begin if its so:
    BCC     |L0.6|
    POP     {r4,pc}
```

195
ARM in ARM mode

Keil in ARM mode takes full advantage of conditional suffixes:

That's why there is only one branch instruction instead of 2.

MIPS

That's why there is only one branch instruction instead of 2.
Here we have two new instructions: LBU ("Load Byte Unsigned") and SB ("Store Byte").

Just like in ARM, all MIPS registers are 32-bit wide, there are no byte-wide parts like in x86. So when dealing with single bytes, we have to allocate whole 32-bit registers for them.

LBU loads a byte and clears all other bits ("Unsigned").

On the other hand, LB ("Load Byte") instruction sign-extends the loaded byte to a 32-bit value. SB just writes a byte from lowest 8 bits of register to memory.

Vectorization

Optimizing GCC can do much more on this example: 1.36.1 on page 411.

1.22.3 Condition check

It’s important to keep in mind that in for() construct, condition is checked not at the end, but at the beginning, before execution of loop body. But often, it’s more convenient for compiler to check it at the end, after body. Sometimes, additional check can be appended at the beginning.

For example:

```c
#include <stdio.h>

void f(int start, int finish)
{
    for (; start<finish; start++)
        printf("%d\n", start);
}
```

Optimizing GCC 5.4.0 x64:

```c
f:
 ; check condition (1):
    cmp   edi, esi
    jge   .L9
    push  rbp
    push  rbx
    mov   ebp, esi
    mov   ebx, edi
    sub   rsp, 8
    .L5:
    mov   edx, ebx
    xor   eax, eax
    mov   esi, OFFSET FLAT:.LC0 ; "%d\n"
    mov   edi, 1
    add   ebx, 1
    call  __printf_chk
 ; check condition (2):
    cmp   ebp, ebx
    jne   .L5
    add   rsp, 8
    pop   rbx
    pop   rbp
 .L9:
    rep ret
```

We see two checks.

Hex-Rays (at least version 2.2.0) decompiles this as:

```c
void __cdecl f(unsigned int start, unsigned int finish)
{
    unsigned int v2; // ebx@2
    __int64 v3; // rdx@3

    if ( (signed int)start < (signed int)finish )
    {
        v2 = start;
```
In this case, do/while() can be replaced by for() without any doubt, and the first check can be removed.

### 1.22.4 Conclusion

Rough skeleton of loop from 2 to 9 inclusive:

```c
Listing 1.182: x86

mov [counter], 2 ; initialization
jmp check
body:
; loop body
; do something here
; use counter variable in local stack
add [counter], 1 ; increment
check:
    cmp [counter], 9
jle body
```

The increment operation may be represented as 3 instructions in non-optimized code:

```c
Listing 1.183: x86

MOV [counter], 2 ; initialization
JMP check
body:
; loop body
; do something here
; use counter variable in local stack
MOV REG, [counter] ; increment
INC REG
MOV [counter], REG
check:
    CMP [counter], 9
    JLE body
```

If the body of the loop is short, a whole register can be dedicated to the counter variable:

```c
Listing 1.184: x86

MOV EBX, 2 ; initialization
JMP check
body:
; loop body
; do something here
; use counter in EBX, but do not modify it!
INC EBX ; increment
check:
    CMP EBX, 9
    JLE body
```

Some parts of the loop may be generated by compiler in different order:

```c
Listing 1.185: x86

MOV [counter], 2 ; initialization
JMP label_check
label_increment:
    ADD [counter], 1 ; increment
label_check:
    CMP [counter], 10
```
JGE exit 
; loop body 
; do something here 
; use counter variable in local stack 
JMP label_increment 
exit: 

Usually the condition is checked before loop body, but the compiler may rearrange it in a way that the condition is checked after loop body.

This is done when the compiler is sure that the condition is always true on the first iteration, so the body of the loop is to be executed at least once:

**Listing 1.186: x86**

```assembly
MOV REG, 2 ; initialization
body:
; loop body
; do something here
; use counter in REG, but do not modify it!
INC REG ; increment
CMP REG, 10
JL body
```

Using the LOOP instruction. This is rare, compilers are not using it. When you see it, it’s a sign that this piece of code is hand-written:

**Listing 1.187: x86**

```assembly
; count from 10 to 1
MOV ECX, 10
body:
; loop body
; do something here
; use counter in ECX, but do not modify it!
LOOP body
```

**ARM.**

The R4 register is dedicated to counter variable in this example:

**Listing 1.188: ARM**

```assembly
MOV R4, 2 ; initialization
B check
body:
; loop body
; do something here
; use counter in R4, but do not modify it!
ADD R4,R4, #1 ; increment
check:
CMP R4, #10
BLT body
```

### 1.22.5 Exercises

- [http://challenges.re/54](http://challenges.re/54)
- [http://challenges.re/55](http://challenges.re/55)
- [http://challenges.re/56](http://challenges.re/56)
- [http://challenges.re/57](http://challenges.re/57)
1.23 More about strings

1.23.1 strlen()

Let's talk about loops one more time. Often, the strlen() function is implemented using a while() statement. Here is how it is done in the MSVC standard libraries:

```c
int my_strlen (const char * str)
{
    const char * eos = str;
    while( *eos++ )
    {
        return( eos - str - 1 );
    }
}

int main()
{
    // test
    return my_strlen("hello");
}
```

x86

Non-optimizing MSVC

Let's compile:

```
_eos$ = -4 ; size = 4
_str$ = 8 ; size = 4
_strlen PROC
    push ebp
    mov ebp, esp
    push ecx
    mov eax, DWORD PTR _str$[ebp] ; place pointer to string from "str"
    mov DWORD PTR _eos$[ebp], eax ; place it to local variable "eos"
$LN2@strlen_
    mov ecx, DWORD PTR _eos$[ebp] ; ECX=eos

    ; take 8-bit byte from address in ECX and place it as 32-bit value to EDX with sign extension
    movsx edx, BYTE PTR [ecx]
    mov eax, DWORD PTR _eos$[ebp] ; EAX=eos
    add eax, 1 ; increment EAX
    mov DWORD PTR _eos$[ebp], eax ; place EAX back to "eos"
    test edx, edx ; EDX is zero?
    je SHORT $LN1@strlen_ ; yes, then finish loop
    jmp SHORT $LN2@strlen_ ; continue loop
$LN1@strlen_

    ; here we calculate the difference between two pointers
    mov eax, DWORD PTR _eos$[ebp]
    sub eax, DWORD PTR _str$[ebp]
    sub eax, 1 ; subtract 1 and return result
    pop ebp
    ret 0
_strlen_ ENDP
```

We get two new instructions here: MOV SX and TEST.

The first one—MOV SX—takes a byte from an address in memory and stores the value in a 32-bit register. MOV SX stands for MOV with Sign-Extend. MOV SX sets the rest of the bits, from the 8th to the 31th, to 1 if the source byte is negative or to 0 if is positive.

103 Counting the characters in a string in the C language
And here is why.

By default, the char type is signed in MSVC and GCC. If we have two values of which one is char and the other is int, (int is signed too), and if the first value contain -2 (coded as 0xFE) and we just copy this byte into the int container, it makes 0x000000FE, and this from the point of signed int view is 254, but not -2. In signed int, -2 is coded as 0xFFFFFFFF. So if we have to transfer 0xFE from a variable of char type to int, we have to identify its sign and extend it. That is what MOV SX does.

You can also read about it in “Signed number representations” section (2.2 on page 448).

It’s hard to say if the compiler needs to store a char variable in EDX, it could just take a 8-bit register part (for example DL). Apparently, the compiler’s register allocator works like that.

Then we see TEST EDX, EDX. You can read more about the TEST instruction in the section about bit fields (1.28 on page 303). Here this instruction just checks if the value in EDX equals to 0.

**Non-optimizing GCC**

Let’s try GCC 4.4.1:

```assembly
public strlen
strlen proc near

eos = dword ptr -4
g0 = dword ptr 8

push ebp
mov ebp, esp
sub esp, 10h
mov eax, [ebp+g0]
mov [ebp+eos], eax

loc_80483F0:

mov eax, [ebp+eos]
movzx eax, byte ptr [eax]
test al, al
setnz al
add [ebp+eos], 1

test al, al
jnz short loc_80483F0

mov edx, [ebp+eos]
mov eax, [ebp+arg_0]
mov ecx, edx
sub ecx, eax
mov eax, ecx
sub eax, 1
leave
ret

strlen endp
```

The result is almost the same as in MSVC, but here we see MOVZX instead of MOVSX. MOVSX stands for MOV with Zero-Extend. This instruction copies a 8-bit or 16-bit value into a 32-bit register and sets the rest of the bits to 0. In fact, this instruction is convenient only because it enable us to replace this instruction pair:

xor eax, eax / mov al, [...].

On the other hand, it is obvious that the compiler could produce this code:

mov al, byte ptr [eax] / test al, al—it is almost the same, however, the highest bits of the EAX register will contain random noise. But let’s think it is compiler’s drawback—it cannot produce more understandable code. Strictly speaking, the compiler is not obliged to emit understandable (to humans) code at all.

The next new instruction for us is SETNZ. Here, if AL doesn’t contain zero, test AL, AL sets the ZF flag to 0, but SETNZ, if ZF==0 (NZ stands for not zero) sets AL to 1. Speaking in natural language, if AL is not zero, let’s jump to loc_80483F0. The compiler emits some redundant code, but let’s not forget that the optimizations are turned off.
Now let’s compile all this in MSVC 2012, with optimizations turned on (/Ox):

Listing 1.189: Optimizing MSVC 2012 /Ob0

```
_str$ = 8 ; size = 4
_strlen PROC
  mov edx, DWORD PTR _str$[esp-4] ; EDX -> pointer to the string
  mov eax, edx ; move to EAX
$LL2@strlen:
  mov cl, BYTE PTR [eax] ; CL = *EAX
  inc eax ; EAX++
  test cl, cl ; CL==0?
  jne SHORT $LL2@strlen ; no, continue loop
  sub eax, edx ; calculate pointers difference
  dec eax ; decrement EAX
ret 0
_strlen ENDP
```

Now it is all simpler. Needless to say, the compiler could use registers with such efficiency only in small functions with a few local variables.

INC/DEC—are **increment/decrement** instructions, in other words: add or subtract 1 to/from a variable.
Optimizing MSVC + OllyDbg

We can try this (optimized) example in OllyDbg. Here is the first iteration:

![CPU - main thread, module ex1](image)

**Figure 1.59: OllyDbg: first iteration start**

We see that OllyDbg found a loop and, for convenience, wrapped its instructions in brackets. By clicking the right button on EAX, we can choose “Follow in Dump” and the memory window scrolls to the right place. Here we can see the string “hello!” in memory. There is at least one zero byte after it and then random garbage.

If OllyDbg sees a register with a valid address in it, that points to some string, it is shown as a string.
Let's press F8 (step over) a few times, to get to the start of the body of the loop:

![CPU - main thread, module ex1](image)

**Figure 1.60: OllyDbg: second iteration start**

We see that EAX contains the address of the second character in the string.
We have to press F8 enough number of times in order to escape from the loop:

![CPU - main thread, module ex1]

We see that EAX now contains the address of zero byte that's right after the string plus 1 (because INC EAX was executed regardless of whether we exit from the loop or not). Meanwhile, EDX hasn't changed, so it still pointing to the start of the string.

The difference between these two addresses is being calculated now.

Figure 1.61: OllyDbg: pointers difference to be calculated now
The SUB instruction just got executed:

```assembly
public strlen
strlen proc near
arg_0 = dword ptr 8
push ebp
mov ebp, esp
mov ecx, [ebp+arg_0]
mov eax, ecx
loc_8048418:
movzx edx, byte ptr [eax]
add eax, 1
test dl, dl
jnz short loc_8048418
not ecx
add eax, ecx
pop ebp
retn
strlen endp
```

Here GCC is almost the same as MSVC, except for the presence of MOVZX. However, here MOVZX could be replaced with

```
mov dl, byte ptr [eax].
```

Perhaps it is simpler for GCC’s code generator to remember the whole 32-bit EDX register is allocated for a `char` variable and it then can be sure that the highest bits has no any noise at any point.
After that we also see a new instruction—NOT. This instruction inverts all bits in the operand. You can say that it is a synonym to the XOR ECX, 0xffffffff instruction. NOT and the following ADD calculate the pointer difference and subtract 1, just in a different way. At the start ECX, where the pointer to str is stored, gets inverted and 1 is subtracted from it.

See also: “Signed number representations” (2.2 on page 448).

In other words, at the end of the function just after loop body, these operations are executed:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

... and this is effectively equivalent to:

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Why did GCC decide it would be better? Hard to guess. But perhaps the both variants are equivalent in efficiency.

ARM

32-bit ARM

Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Listing 1.190: Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Non-optimizing LLVM generates too much code, however, here we can see how the function works with local variables in the stack. There are only two local variables in our function: eos and str. In this listing, generated by IDA, we have manually renamed var_8 and var_4 to eos and str.

The first instructions just saves the input values into both str and eos.

The body of the loop starts at label loc_2CB8.
The first three instruction in the loop body (LDR, ADD, STR) load the value of eos into R0. Then the value is incremented and saved back into eos, which is located in the stack.

The next instruction, LDRSB R0, [R0] (“Load Register Signed Byte”), loads a byte from memory at the address stored in R0 and sign-extends it to 32-bit 104. This is similar to the MOVSX instruction in x86.

The compiler treats this byte as signed since the char type is signed according to the C standard. It was already written about it (1.23.1 on page 200) in this section, in relation to x86.

It has to be noted that it is impossible to use 8- or 16-bit part of a 32-bit register in ARM separately of the whole register, as it is in x86.

Apparently, it is because x86 has a huge history of backwards compatibility with its ancestors up to the 16-bit 8086 and even 8-bit 8080, but ARM was developed from scratch as a 32-bit RISC-processor.

Consequently, in order to process separate bytes in ARM, one has to use 32-bit registers anyway.

So, LDRSB loads bytes from the string into R0, one by one. The following CMP and BEQ instructions check if the loaded byte is 0. If it’s not 0, control passes to the start of the body of the loop. And if it’s 0, the loop ends.

At the end of the function, the difference between eos and str is calculated, 1 is subtracted from it, and resulting value is returned via R0.

N.B. Registers were not saved in this function.

That’s because in the ARM calling convention registers R0-R3 are “scratch registers”, intended for arguments passing, and we’re not required to restore their value when the function exits, since the calling function will not use them anymore. Consequently, they may be used for anything we want.

No other registers are used here, so that is why we have nothing to save on the stack.

Thus, control may be returned back to calling function by a simple jump (BX), to the address in the LR register.

**Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)**

<table>
<thead>
<tr>
<th>Listing 1.191: Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_strlen</code></td>
</tr>
<tr>
<td><code>loc_2DF6</code></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

As optimizing LLVM concludes, eos and str do not need space on the stack, and can always be stored in registers.

Before the start of the loop body, str is always in R0, and eos—in R1.

The LDRB.W R2, [R1],#1 instruction loads a byte from the memory at the address stored in R1, to R2, sign-extending it to a 32-bit value, but not just that. #1 at the instruction’s end is implies “Post-indexed addressing”, which means that 1 is to be added to R1 after the byte is loaded. Read more about it: 1.39.2 on page 437.

Then you can see CMP and BNE105 in the body of the loop, these instructions continue looping until 0 is found in the string.

MVNS106 (inverts all bits, like NOT in x86) and ADD instructions compute eos − str − 1. In fact, these two instructions compute R0 = str + eos, which is effectively equivalent to what was in the source code, and why it is so, was already explained here (1.23.1 on page 206).

Apparently, LLVM, just like GCC, concludes that this code can be shorter (or faster).

---

104The Keil compiler treats the char type as signed, just like MSVC and GCC.
105(PowerPC, ARM) Branch if Not Equal
106MoVe Not
Almost the same as what we saw before, with the exception that the \texttt{str-\texttt{eos}-1} expression can be computed not at the function's end, but right in the body of the loop. The \texttt{-EQ} suffix, as we may recall, implies that the instruction executes only if the operands in the \texttt{CMP} that has been executed before were equal to each other. Thus, if \texttt{R0} contains 0, both \texttt{SUBEQ} instructions executes and result is left in the \texttt{R0} register.

\textbf{ARM64}

**Optimizing GCC (Linaro) 4.9**

```
my_strlen:
    mov  x1, x0
    ; X1 is now temporary pointer (eos), acting like cursor
.L58:
    ; load byte from X1 to W2, increment X1 (post-index)
    ldrb  w2, [x1],1
    ; Compare and Branch if NonZero: compare W2 with 0, jump to .L58 if it is not
    cbnz  w2, .L58
    ; calculate difference between initial pointer in X0 and current address in X1
    sub   x0, x1, x0
    ; decrement lowest 32-bit of result
    sub   w0, w0, #1
    ret
```

The algorithm is the same as in 1.23.1 on page 202: find a zero byte, calculate the difference between the pointers and decrement the result by 1. Some comments were added by the author of this book.

The only thing worth noting is that our example is somewhat wrong: \texttt{my_strlen()} returns 32-bit \texttt{int}, while it has to return \texttt{size\_t} or another 64-bit type.

The reason is that, theoretically, \texttt{strlen()} can be called for a huge blocks in memory that exceeds 4GB, so it must able to return a 64-bit value on 64-bit platforms.

Because of my mistake, the last \texttt{SUB} instruction operates on a 32-bit part of register, while the penultimate \texttt{SUB} instruction works on full the 64-bit register (it calculates the difference between the pointers).

It’s my mistake, it is better to leave it as is, as an example of how the code could look like in such case.

**Non-optimizing GCC (Linaro) 4.9**

```
my_strlen:
    ; function prologue
    sub   sp, sp, #32
    ; first argument (str) will be stored in [sp,8]
    str   x0, [sp,8]
    ldr   x0, [sp,8]
    ; copy "str" to "eos" variable
    str   x0, [sp,24]
    nop
.L62:
    ; eos++
```
It’s more verbose. The variables are often tossed here to and from memory (local stack). The same mistake here: the decrement operation happens on a 32-bit register part.

**MIPS**

```
my_strlen:
    ; "eos" variable will always reside in $v1:
    move $v1, $a0

loc_4:
    ; load byte at address in "eos" into $a1:
    lb $a1, 0($v1)
    or $at, $zero ; load delay slot, NOP
    ; if loaded byte is not zero, jump to loc_4:
    bnez $a1, loc_4
    ; increment "eos" anyway:
    addiu $v1, 1 ; branch delay slot
    ; loop finished. invert "str" variable:
    nor $v0, $zero, $a0
    ; $v0=-str-1
    jr $ra
    ; return value = $v1 + $v0 = eos + ( -str-1 ) = eos - str - 1
    addu $v0, $v1, $v0 ; branch delay slot
```

MIPS lacks a NOT instruction, but has NOR which is OR + NOT operation.

This operation is widely used in digital electronics\(^\text{107}\). For example, the Apollo Guidance Computer used in the Apollo program, was built by only using 5600 NOR gates: [Jens Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*, (2011)]. But NOR element isn’ very popular in computer programming.

So, the NOT operation is implemented here as NOR DST, $ZERO, SRC.

From fundamentals 2.2 on page 448 we know that bitwise inverting a signed number is the same as changing its sign and subtracting 1 from the result.

So what NOT does here is to take the value of str and transform it into $-str-1$. The addition operation that follows prepares result.

### 1.23.2 Boundaries of strings

It’s interesting to note, how parameters are passed into win32 `GetOpenFileName()` function. In order to call it, one must set list of allowed file extensions:

\(^{107}\)NOR is called “universal gate”
What happens here is that list of strings are passed into `GetOpenFileName()`. It is not a problem to parse it: whenever you encounter single zero byte, this is an item. Whenever you encounter two zero bytes, this is end of the list. If you will pass this string into `printf()`, it will treat first item as a single string.

So this is string, or...? It’s better say this is buffer containing several zero-terminated C-strings, which can be stored and processed as a whole.

Another example is `strtok()` function. It takes a string and write zero bytes in the middle of it. It thus transforms input string into some kind of buffer, which has several zero-terminated C-strings.

### 1.24 Replacing arithmetic instructions to other ones

In the pursuit of optimization, one instruction may be replaced by another, or even with a group of instructions. For example, `ADD` and `SUB` can replace each other: line 18 in listing 3.122.

For example, the `LEA` instruction is often used for simple arithmetic calculations: [1.6 on page 1014](#).

#### 1.24.1 Multiplication

**Multiplication using addition**

Here is a simple example:

```c
unsigned int f(unsigned int a)
{
    return a*8;
}
```

Multiplication by 8 is replaced by 3 addition instructions, which do the same. Apparently, MSVC’s optimizer decided that this code can be faster.

Listing 1.194: Optimizing MSVC 2010

```asm
_TEXT SEGMENT
_a$ = 8 ; size = 4
_f PROC
    mov    eax, DWORD PTR _a$[esp-4]
    add    eax, eax
    add    eax, eax
    add    eax, eax
    ret
_f ENDP
TEXT ENDS
END
```

**Multiplication using shifting**

Multiplication and division instructions by a numbers that’s a power of 2 are often replaced by shift instructions.

```c
unsigned int f(unsigned int a)
{
    return a*4;
}
```
Listing 1.195: Non-optimizing MSVC 2010

```assembly
_a$ = 8 ; size = 4
_f PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    shl eax, 2
    pop ebp
    ret 0
_f ENDP
```

Multiplication by 4 is just shifting the number to the left by 2 bits and inserting 2 zero bits at the right (as the last two bits). It is just like multiplying 3 by 100—we just have to add two zeros at the right.

That’s how the shift left instruction works:

![Shift Left Instruction Diagram]

The added bits at right are always zeros.

Multiplication by 4 in ARM:

```assembly
Listing 1.196: Non-optimizing Keil 6/2013 (ARM mode)
f PROC
    LSL r0, r0, #2
    BX lr
_f ENDP
```

Multiplication by 4 in MIPS:

```assembly
Listing 1.197: Optimizing GCC 4.4.5 (IDA)
jr $ra
sll $v0, $a0, 2 ; branch delay slot
```

SLL is “Shift Left Logical”.

**Multiplication using shifting, subtracting, and adding**

It’s still possible to get rid of the multiplication operation when you multiply by numbers like 7 or 17 again by using shifting. The mathematics used here is relatively easy.

**32-bit**

```c
#include <stdint.h>

int f1(int a)
{
    return a*7;
};

int f2(int a)
{
    return a*28;
};

int f3(int a)
{
    return a*17;
};
```

212
x86

Listing 1.198: Optimizing MSVC 2012

```assembly
; a*7
_a$ = 8
_f1 PROC
    ecx, DWORD PTR _a$[esp-4]
; ECX=a
    eax, DWORD PTR [ecx*8]
; EAX=EAX*8
    sub eax, ecx
    EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
    ret 0
_f1 ENDP

; a*28
_a$ = 8
_f2 PROC
    ecx, DWORD PTR _a$[esp-4]
; ECX=a
    eax, DWORD PTR [ecx*8]
; EAX=EAX*8
    sub eax, ecx
    EAX=EAX-ECX=ECX*8-ECX=ECX*7=a*7
    shl eax, 2
    EAX=EAX<<2=(a*7)*4=a*28
    ret 0
_f2 ENDP

; a*17
_a$ = 8
_f3 PROC
    eax, DWORD PTR _a$[esp-4]
; EAX=a
    shl eax, 4
; EAX=EAX<<4=EAX*16=a*16
    add eax, DWORD PTR _a$[esp-4]
; EAX=EAX+a*16=a*17
    ret 0
_f3 ENDP
```

ARM

Keil for ARM mode takes advantage of the second operand's shift modifiers:

Listing 1.199: Optimizing Keil 6/2013 (ARM mode)

```assembly
; a*7
||f1|| PROC
RSB r0, r0, r0, LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
    BX lr
ENDP

; a*28
||f2|| PROC
RSB r0, r0, r0, LSL #3
; R0=R0<<3-R0=R0*8-R0=a*8-a=a*7
    LSL r0, r0, #2
; R0=R0<<2=R0*4=a*7*4=a*28
    BX lr
ENDP

; a*17
||f3|| PROC
ADD r0, r0, r0, LSL #4
; R0=R0+R0<<4=R0+R0*16=R0*17=a*17
```
But there are no such modifiers in Thumb mode. It also can’t optimize \( f_2() \):

**Listing 1.200: Optimizing Keil 6/2013 (Thumb mode)**

```assembly
; a*7
||f1|| PROC
    LSLS r1,r0,#3
    ; R1=R0<<3=a<<3=a*8
    SUBS r0,r1,r0
    ; R0=R1-R0=a*8-a=a*7
    BX lr
ENDP

; a*28
||f2|| PROC
    MOV r1,#0x1c ; 28
    ; R1=28
    MULS r0,r1,r0
    ; R0=R1*R0=28*a
    BX lr
ENDP

; a*17
||f3|| PROC
    LSLS r1,r0,#4
    ; R1=R0<<4=R0*16=a*16
    ADDS r0,r0,r1
    ; R0=R0+R1=a+a*16=a*17
    BX lr
ENDP
```

**MIPS**

**Listing 1.201: Optimizing GCC 4.4.5 (IDA)**

```
f1:  sll  $v0, $a0, 3
; $v0 = $a0<<3 = $a0*8
    jr $ra    subu $v0, $a0 ; branch delay slot
; $v0 = $v0-$a0 = $a0*8-$a0 = $a0*7

f2:  sll  $v0, $a0, 5
; $v0 = $a0<<5 = $a0*32
    sll $a0, 2
; $a0 = $a0<<2 = $a0*4
    jr $ra
    subu $v0, $a0 ; branch delay slot
; $v0 = $a0*32-$a0*4 = $a0*28

f3:  sll  $v0, $a0, 4
; $v0 = $a0<<4 = $a0*16
    jr $ra
    addu $v0, $a0 ; branch delay slot
; $v0 = $a0*16+$a0 = $a0*17
```

**64-bit**

```c
#include <stdint.h>

int64_t f1(int64_t a)
```

214
Listing 1.202: Optimizing MSVC 2012

```c
int64_t f1(int64_t a)
{
    return a*7;
}

int64_t f2(int64_t a)
{
    return a*28;
}

int64_t f3(int64_t a)
{
    return a*17;
}

x64

ARM64

GCC 4.9 for ARM64 is also terse, thanks to the shift modifiers:

Listing 1.203: Optimizing GCC (Linaro) 4.9 ARM64

```c
; a*7
f1:
    lsl x1, x0, 3
    ; X1=X0<<3=X0*8=a*8
    sub x0, x1, x0
    ; X0=X1-X0=a*8-a=a*7
    ret

; a*28
f2:
    lsl x1, x0, 5
    ; X1=X0<<5=X0*32=a*32
    sub x0, x1, x0, lsl 2
    ; X0=X1-X0=a*32-a<<2=a*32-a*4=a*28
    ret
```
Booth’s multiplication algorithm

There was a time when computers were big and that expensive, that some of them lacked hardware support of multiplication operation in CPU, like Data General Nova. And when one need multiplication operation, it can be provided at software level, for example, using Booth’s multiplication algorithm. This is a multiplication algorithm which uses only addition operation and shifts.

What modern optimizing compilers do, isn’t the same, but the goal (multiplication) and resources (faster operations) are the same.

1.24.2 Division

Division using shifts

Example of division by 4:

```c
unsigned int f(unsigned int a)
{
    return a/4;
}
```

We get (MSVC 2010):

```
Listing 1.204: MSVC 2010

_a$ = 8 ; size = 4
_f PROC
    mov    eax, DWORD PTR _a$[esp-4]
    shr    eax, 2
    ret
_f ENDP
```

The SHR (Shift Right) instruction in this example is shifting a number by 2 bits to the right. The two freed bits at left (e.g., two most significant bits) are set to zero. The two least significant bits are dropped. In fact, these two dropped bits are the division operation remainder.

The SHR instruction works just like SHL, but in the other direction.

```
7 6 5 4 3 2 1 0
```

It is easy to understand if you imagine the number 23 in the decimal numeral system. 23 can be easily divided by 10 just by dropping last digit (3—division remainder). 2 is left after the operation as a quotient.

So the remainder is dropped, but that’s OK, we work on integer values anyway, these are not a real numbers!

Division by 4 in ARM:

```
Listing 1.205: Non-optimizing Keil 6/2013 (ARM mode)

f PROC
    LSR    r0,r0,#2
    BX     lr
ENDP
```

Division by 4 in MIPS:
The SRL instruction is “Shift Right Logical”.

### 1.24.3 Exercise

- [http://challenges.re/59](http://challenges.re/59)

### 1.25 Floating-point unit

The FPU is a device within the main CPU, specially designed to deal with floating point numbers. It was called “coprocessor” in the past and it stays somewhat aside of the main CPU.

#### 1.25.1 IEEE 754

A number in the IEEE 754 format consists of a sign, a significand (also called fraction) and an exponent.

#### 1.25.2 x86

It is worth looking into stack machines or learning the basics of the Forth language, before studying the FPU in x86.

It is interesting to know that in the past (before the 80486 CPU) the coprocessor was a separate chip and it was not always pre-installed on the motherboard. It was possible to buy it separately and install it.

Starting with the 80486 DX CPU, the FPU is integrated in the CPU.

The FWAIT instruction reminds us of that fact—it switches the CPU to a waiting state, so it can wait until the FPU has finished with its work.

Another rudiment is the fact that the FPU instruction opcodes start with the so called “escape”-opcodes (D8..DF), i.e., opcodes passed to a separate coprocessor.

The FPU has a stack capable to holding 8 80-bit registers, and each register can hold a number in the IEEE 754 format.

They are ST(0)…ST(7). For brevity, IDA and OllyDbg show ST(0) as ST, which is represented in some textbooks and manuals as “Stack Top”.

#### 1.25.3 ARM, MIPS, x86/x64 SIMD

In ARM and MIPS the FPU is not a stack, but a set of registers, which can be accessed randomly, like GPR.

The same ideology is used in the SIMD extensions of x86/x64 CPUs.

#### 1.25.4 C/C++

The standard C/C++ languages offer at least two floating number types, float (single-precision, 32 bits) and double (double-precision, 64 bits).

In [Donald E. Knuth, The Art of Computer Programming, Volume 2, 3rd ed., (1997)246] we can find the single-precision means that the floating point value can be placed into a single [32-bit] machine word, double-precision means it can be stored in two words (64 bits).

GCC also supports the long double type (extended precision, 80 bit), which MSVC doesn’t.

The float type requires the same number of bits as the int type in 32-bit environments, but the number representation is completely different.

---

108For example, John Carmack used fixed-point arithmetic values in his Doom video game, stored in 32-bit GPR registers (16 bit for integral part and another 16 bit for fractional part), so Doom could work on 32-bit computers without FPU, i.e., 80386 and 80486 SX.

109the single precision floating point number format is also addressed in the Handling float data type as a structure (1.30.6 on page 370) section
1.25.5 Simple example

Let's consider this simple example:

```c
#include <stdio.h>

double f (double a, double b) {
    return a/3.14 + b*4.1;
};

int main() {
    printf("%f\n", f(1.2, 3.4));
};
```

x86

MSVC

Compile it in MSVC 2010:

```asm
CONST SEGMENT
_real@4010666666666666 DQ 040106666666666666
CONST ENDS
CONST SEGMENT
_real@40091eb851eb851f DQ 040091eb851eb851fr
CONST ENDS
_TEXT SEGMENT
  a$ = 8 ; size = 8
  b$ = 16 ; size = 8
_f PROC
  push ebp
  mov ebp, esp
  fld QWORD PTR _a$[ebp]
  ; current stack state: ST(0) = _a

  fdiv QWORD PTR __real@40091eb851eb851f
  ; current stack state: ST(0) = result of _a divided by 3.14

  fld QWORD PTR _b$[ebp]
  ; current stack state: ST(0) = _b;
  ; ST(1) = result of _a divided by 3.14

  fmul QWORD PTR __real@4010666666666666
  ; current stack state:
  ; ST(0) = result of _b * 4.1;
  ; ST(1) = result of _a divided by 3.14

  faddp ST(1), ST(0)
  ; current stack state: ST(0) = result of addition

  pop ebp
  ret 0
_f ENDP
```

FLD takes 8 bytes from stack and loads the number into the ST(0) register, automatically converting it into the internal 80-bit format (extended precision).

FDIV divides the value in ST(0) by the number stored at address __real@40091eb851eb851f — the value 3.14 is encoded there. The assembly syntax doesn't support
floating point numbers, so what we see here is the hexadecimal representation of 3.14 in 64-bit IEEE 754 format.

After the execution of FDIV ST(0) holds the quotient.

By the way, there is also the FDIVP instruction, which divides ST(1) by ST(0), popping both these values from stack and then pushing the result. If you know the Forth language, you can quickly understand that this is a stack machine.

The subsequent FLD instruction pushes the value of $b$ into the stack.

After that, the quotient is placed in ST(1), and ST(0) has the value of $b$.

The next FMUL instruction does multiplication: $b$ from ST(0) is multiplied by value at __real@4010666666666666 (the number 4.1 is there) and leaves the result in the ST(0) register.

The last FADDP instruction adds the two values at top of stack, storing the result in ST(1) and then popping the value of ST(0), thereby leaving the result at the top of the stack, in ST(0).

The function must return its result in the ST(0) register, so there are no any other instructions except the function epilogue after FADDP.
2 pairs of 32-bit words are marked by red in the stack. Each pair is a double-number in IEEE 754 format and is passed from `main()`.

We see how the first `FLD` loads a value (1.2) from the stack and puts it into ST(0):

Because of unavoidable conversion errors from 64-bit IEEE 754 floating point to 80-bit (used internally in the FPU), here we see 1.1999…, which is close to 1.2.

EIP now points to the next instruction (FDIV), which loads a double-number (a constant) from memory. For convenience, OllyDbg shows its value: 3.14
Let's trace further. FDIV has been executed, now ST(0) contains 0.382\ldots (quotient):

![CPU - main thread, module simple](image)

**Figure 1.64:** OllyDbg: FDIV has been executed
Third step: the next FLD has been executed, loading 3.4 into ST(0) (here we see the approximate value 3.39999…):

At the same time, quotient is pushed into ST(1). Right now, EIP points to the next instruction: FMUL. It loads the constant 4.1 from memory, which OllyDbg shows.

Figure 1.65: OllyDbg: the second FLD has been executed
Next: FMUL has been executed, so now the product is in ST(0):

Figure 1.66: OllyDbg: the FMUL has been executed
Next: the FADDP has been executed, now the result of the addition is in ST(0), and ST(1) is cleared:

Figure 1.67: OllyDbg: FADDP has been executed

The result is left in ST(0), because the function returns its value in ST(0).

main() takes this value from the register later.

We also see something unusual: the 13.93...value is now located in ST(7). Why?
As we have read some time before in this book, the FPU registers are a stack: 1.25.2 on page 217. But this is a simplification.

Imagine if it was implemented in hardware as it's described, then all 7 register’s contents must be moved (or copied) to adjacent registers during pushing and popping, and that’s a lot of work.

In reality, the FPU has just 8 registers and a pointer (called TOP) which contains a register number, which is the current “top of stack”.

When a value is pushed to the stack, TOP is pointed to the next available register, and then a value is written to that register.

The procedure is reversed if a value is popped, however, the register which has been freed is not cleared (it could possibly be cleared, but this is more work which can degrade performance). So that’s what we see here.

It can be said that FADDP saved the sum in the stack, and then popped one element.

More precisely, the registers of the FPU are a circular buffer.

GCC

GCC 4.4.1 (with -O3 option) emits the same code, just slightly different:
Listing 1.208: Optimizing GCC 4.4.1

```c
public f
  proc near
  arg_0 = qword ptr 8
  arg_8 = qword ptr 10h
  push ebp
  fld ds:dbl_8048608 ; 3.14

; stack state now: ST(0) = 3.14
  mov ebp, esp
  fdivr [ebp+arg_0]

; stack state now: ST(0) = result of division
  fld ds:dbl_8048610 ; 4.1

; stack state now: ST(0) = 4.1, ST(1) = result of division
  fmul [ebp+arg_8]

; stack state now: ST(0) = result of multiplication, ST(1) = result of division
  pop ebp
  faddp st(1), st

; stack state now: ST(0) = result of addition
  retn
f
endp
```

The difference is that, first of all, 3.14 is pushed to the stack (into ST(0)), and then the value in arg_0 is divided by the value in ST(0).

FDIVR stands for Reverse Divide—to divide with divisor and dividend swapped with each other. There is no likewise instruction for multiplication since it is a commutative operation, so we just have FMUL without its -R counterpart.

FADDP adds the two values but also pops one value from the stack. After that operation, ST(0) holds the sum.

**ARM: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)**

Until ARM got standardized floating point support, several processor manufacturers added their own instructions extensions. Then, VFP (Vector Floating Point) was standardized.

One important difference from x86 is that in ARM, there is no stack, you work just with registers.

Listing 1.209: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```c
f
  VLDR   D16, =3.14
  VMOV   D17, R0, R1 ; load "a"
  VMOV   D18, R2, R3 ; load "b"
  VDIV.F64 D16, D17, D16 ; a/3.14
  VLDR   D17, =4.1
  VMUL.F64 D17, D18, D17 ; b*4.1
  VADD.F64 D16, D17, D16 ; +
  VMOV   R0, R1, D16
  BX      LR
  
  dbl_2C98 DCFD 3.14 ; DATA XREF: f
  dbl_2CA0 DCFD 4.1 ; DATA XREF: f+10
```

So, we see here new some registers used, with D prefix.

These are 64-bit registers, there are 32 of them, and they can be used both for floating-point numbers (double) but also for SIMD (it is called NEON here in ARM).
There are also 32 32-bit S-registers, intended to be used for single precision floating pointer numbers (float).

It is easy to memorize: D-registers are for double precision numbers, while S-registers—for single precision numbers. More about it: 2.3 on page 1026.

Both constants (3.14 and 4.1) are stored in memory in IEEE 754 format.

VLDR and VMOV, as it can be easily deduced, are analogous to the LDR and MOV instructions, but they work with D-registers.

It has to be noted that these instructions, just like the D-registers, are intended not only for floating point numbers, but can be also used for SIMD (NEON) operations and this will also be shown soon.

The arguments are passed to the function in a common way, via the R-registers, however each number that has double precision has a size of 64 bits, so two R-registers are needed to pass each one.

VMOV D17, R0, R1 at the start, composes two 32-bit values from R0 and R1 into one 64-bit value and saves it to D17.
VMOV R0, R1, D16 is the inverse operation: what has been in D16 is split in two registers, R0 and R1, because a double-precision number that needs 64 bits for storage, is returned in R0 and R1.

VDIV, VMUL and VADD, are instruction for processing floating point numbers that compute quotient, product and sum, respectively.

The code for Thumb-2 is same.

**ARM: Optimizing Keil 6/2013 (Thumb mode)**

```
f                                  
PUSH {R3-R7,LR}                     
MOV    R7, R2                       
MOV    R4, R3                       
MOV    R5, R0                       
MOV    R6, R1                       
LDR    R2, =0x6666666666           ; 4.1
LDR    R3, =0x4010666666           
MOV    R0, R7                       
MOV    R1, R4                       
BL     __aeabi_dmul                
MOV    R7, R0                       
MOV    R4, R1                       
LDR    R2, =0x51EB851F             ; 3.14
LDR    R3, =0x40091EB8             
MOV    R0, R5                       
MOV    R1, R6                       
BL     __aeabi_ddiv                
MOV    R2, R7                       
MOV    R3, R4                       
BL     __aeabi_dadd                
POP    {R3-R7,PC}                  

; 4.1 in IEEE 754 form:
dword_364  DCD 0x6666666666       ; DATA XREF: f+A

dword_368  DCD 0x4010666666       ; DATA XREF: f+C

; 3.14 in IEEE 754 form:
dword_36C  DCD 0x51EB851F         ; DATA XREF: f+1A

dword_370  DCD 0x40091EB8         ; DATA XREF: f+1C
```

Keil generated code for a processor without FPU or NEON support.

The double-precision floating-point numbers are passed via generic R-registers, and instead of FPU-instructions, service library functions are called (like __aeabi_dmul, __aeabi_ddiv, __aeabi_dadd) which emulate multiplication, division and addition for floating-point numbers.

Of course, that is slower than FPU-coprocessor, but it’s still better than nothing.

By the way, similar FPU-emulating libraries were very popular in the x86 world when coprocessors were rare and expensive, and were installed only on expensive computers.
The FPU-coprocessor emulation is called \textit{soft float} or \textit{armel (emulation)} in the ARM world, while using the coprocessor’s FPU-instructions is called \textit{hard float} or \textit{armhf}.

**ARM64: Optimizing GCC (Linaro) 4.9**

Very compact code:

Listing 1.210: Optimizing GCC (Linaro) 4.9

```assembly
f:
  ; D0 = a, D1 = b
  ldr d2, .LC25 ; 3.14
  ; D2 = 3.14
  fdiv d0, d0, d2
  ; D0 = D0/D2 = a/3.14
  ldr d2, .LC26 ; 4.1
  ; D2 = 4.1
  fmadd d0, d1, d2, d0
  ; D0 = D1*D2+D0 = b*4.1+a/3.14
  ret

; constants in IEEE 754 format:
.LC25:
  .word 1374389535 ; 3.14
  .word 1074339512
.LC26:
  .word 1717986918 ; 4.1
  .word 1074816614
```

**ARM64: Non-optimizing GCC (Linaro) 4.9**

Listing 1.211: Non-optimizing GCC (Linaro) 4.9

```assembly
f:
  sub sp, sp, #16
  str d0, [sp,8] ; save "a" in Register Save Area
  str d1, [sp] ; save "b" in Register Save Area
  ldr x1, [sp,8]
  ; X1 = a
  ldr x0, .LC25
  ; X0 = 3.14
  fmov d0, x1
  fmov d1, x0
  ; D0 = a, D1 = 3.14
  fdiv d0, d0, d1
  ; D0 = D0/D1 = a/3.14
  fmov x1, d0
  ; X1 = a/3.14
  ldr x2, [sp]
  ; X2 = b
  ldr x0, .LC26
  ; X0 = 4.1
  fmov d0, x2
  ; D0 = b
  fmov d1, x0
  ; D1 = 4.1
  fmul d0, d0, d1
  ; D0 = D0*D1 = b*4.1
  fmov x0, d0
  ; X0 = D0 = b*4.1
  fmov d0, x1
  ; D0 = a/3.14
  fmov d1, x0
  ; D1 = X0 = b*4.1
  fadd d0, d0, d1
  ; D0 = D0+D1 = a/3.14 + b*4.1
```
Non-optimizing GCC is more verbose.

There is a lot of unnecessary value shuffling, including some clearly redundant code (the last two FMOV instructions). Probably, GCC 4.9 is not yet good in generating ARM64 code.

What is worth noting is that ARM64 has 64-bit registers, and the D-registers are 64-bit ones as well. So the compiler is free to save values of type double in GPRs instead of the local stack. This isn’t possible on 32-bit CPUs.

And again, as an exercise, you can try to optimize this function manually, without introducing new instructions like FMADD.

### 1.25.6 Passing floating point numbers via arguments

```c
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));
    return 0;
}
```

**x86**

Let’s see what we get in (MSVC 2010):

```assembly
CONST    SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r ; 1.54
CONST    ENDS

_main  PROC
    push    ebp
    mov     ebp, esp
    sub     esp, 8 ; allocate space for the first variable
    fld     QWORD PTR __real@3ff8a3d70a3d70a4
    fstp    QWORD PTR [esp]
    sub     esp, 8 ; allocate space for the second variable
    fld     QWORD PTR __real@40400147ae147ae1
    fstp    QWORD PTR [esp]
    call    _pow
    add     esp, 8 ; return back place of one variable.

; in local stack here 8 bytes still reserved for us.
; result now in ST(0)
    fstp    QWORD PTR [esp] ; move result from ST(0) to local stack for printf()
    push    OFFSET $SG2651
    call    _printf
    add     esp, 12
    xor     eax, eax
    pop     ebp
    ret     0

_main  ENDP
```

228
FLD and FSTP move variables between the data segment and the FPU stack. \texttt{pow()}\textsuperscript{110} takes both values from the stack and returns its result in the ST(0) register. \texttt{printf()} takes 8 bytes from the local stack and interprets them as double type variable.

By the way, a pair of \texttt{MOV} instructions could be used here for moving values from the memory into the stack, because the values in memory are stored in IEEE 754 format, and \texttt{pow()} also takes them in this format, so no conversion is necessary. That’s how it’s done in the next example, for ARM: \texttt{1.25.6}.

\begin{verbatim}
ARM + Non-optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

\_main

var\_C  = -0xC

PUSH  \{R7,LR\}
MOV   R7, SP
SUB   SP, SP, \#4
VLDR  D16, =32.01
VMOV  R0, R1, D16
VLDR  D16, =1.54
VMOV  R2, R3, D16
BLX   \_pow
VMOV  D16, R0, R1
MOV   R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\n"
ADD   R0, PC
VMOV  R1, R2, D16
BLX   \_printf
MOV   R1, 0
STR   R0, [SP,\#0xC+var\_C]
MOV   R0, R1
ADD   SP, SP, \#4
POP   \{R7,PC\}

dbl\_2F90  DCFD  32.01 ; DATA XREF: _main+6
dbl\_2F98  DCFD  1.54 ; DATA XREF: _main+E
\end{verbatim}

As it was mentioned before, 64-bit floating pointer numbers are passed in R-registers pairs.

This code is a bit redundant (certainly because optimization is turned off), since it is possible to load values into the R-registers directly without touching the D-registers.

So, as we see, the \_pow function receives its first argument in R0 and R1, and its second one in R2 and R3. The function leaves its result in R0 and R1. The result of \_pow is moved into D16, then in the R1 and R2 pair, from where \texttt{printf()} takes the resulting number.

\begin{verbatim}
ARM + Non-optimizing Keil 6/2013 (ARM mode)

\_main

STMFD  SP!, \{R4-R6,LR\}
LDR   R2, =0xA3D70A4 ; y
LDR   R3, =0x3FF8A3D7
LDR   R0, =0xAE147AE1 ; x
LDR   R1, =0x40400147
BL    \_pow
MOV   R4, R0
MOVS  R2, R4
MOV   R3, R1
ADR   R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n"
BL    _2printf
MOV   R0, \#0
LDMFD SP!, \{R4-R6,PC\}

y       DCD  0xA3D70A4 ; DATA XREF: _main+4
dword_520 DCD  0x3FF8A3D7 ; DATA XREF: _main+8
x       DCD  0xAE147AE1 ; DATA XREF: _main+C
dword_528 DCD  0x40400147 ; DATA XREF: _main+10
a32_011_54Lf DCB "32.01 ^ 1.54 = %lf",0xA,0
\end{verbatim}

\textsuperscript{110} a standard C function, raises a number to the given power (exponentiation)
D-registers are not used here, just R-register pairs.

### ARM64 + Optimizing GCC (Linaro) 4.9

Listing 1.213: Optimizing GCC (Linaro) 4.9

```
f:
  stp  x29, x30, [sp, -16]!
  add  x29, sp, 0
  ldr  d1, .LC1; load 1.54 into D1
  ldr  d0, .LC0; load 32.01 into D0
  bl  pow
; result of pow() in D0
  adrp  x0, .LC2
  add  x0, x0, :lo12:.LC2
  bl  printf
  mov  w0, 0
  ldp  x29, x30, [sp], 16
ret

.LC0:
; 32.01 in IEEE 754 format
  .word  -1374389535
  .word   1077936455

.LC1:
; 1.54 in IEEE 754 format
  .word   171798692
  .word  1073259479

.LC2:
  .string "32.01 ^ 1.54 = %lf\n"
```

The constants are loaded into D0 and D1: pow() takes them from there. The result will be in D0 after the execution of pow(). It is to be passed to printf() without any modification and moving, because printf() takes arguments of integral types and pointers from X-registers, and floating point arguments from D-registers.

### 1.25.7 Comparison example

Let's try this:

```c
#include <stdio.h>

double d_max (double a, double b) {
  if (a>b)
    return a;
  return b;
};

int main() {
  printf ("%f\n", d_max (1.2, 3.4));
  printf ("%f\n", d_max (5.6, -4));
};
```

Despite the simplicity of the function, it will be harder to understand how it works.

### x86

**Non-optimizing MSVC**

MSVC 2010 generates the following:

Listing 1.214: Non-optimizing MSVC 2010
PUBLIC    d_max
_TEXT    SEGMENT
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_d_max  PROC
  push    ebp
  mov     ebp, esp
  fld     QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b
; compare _b (ST(0)) and _a, and pop register
  fcomp   QWORD PTR _a$[ebp]

; stack is empty here
  fnstsw  ax
  test    ah, 5
  jpe     SHORT $LN1@d_max

; we are here only if a>b
  fld     QWORD PTR _a$[ebp]
  jmp     SHORT $LN2@d_max

$LN1@d_max:
  fld     QWORD PTR _b$[ebp]
$LN2@d_max:
  pop     ebp
  ret 0
_d_max  ENDP

So, FLD loads _b into ST(0).

FCOMP compares the value in ST(0) with what is in _a and sets C3/C2/C0 bits in FPU status word register, accordingly. This is a 16-bit register that reflects the current state of the FPU.

After the bits are set, the FCOMP instruction also pops one variable from the stack. This is what distinguishes it from FCOM, which is just compares values, leaving the stack in the same state.

Unfortunately, CPUs before Intel P6 don’t have any conditional jumps instructions which check the C3/C2/C0 bits. Perhaps, it is a matter of history (recall: FPU was a separate chip in past). Modern CPU starting at Intel P6 have FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions —which do the same, but modify the ZF/IF/CF CPU flags.

The FNSTSW instruction copies FPU the status word register to AX. C3/C2/C0 bits are placed at positions 14/10/8, they are at the same positions in the AX register and all they are placed in the high part of AX —AH.

• If \( b > a \) in our example, then C3/C2/C0 bits are to be set as following: 0, 0, 0.
• If \( a > b \), then the bits are: 0, 0, 1.
• If \( a = b \), then the bits are: 1, 0, 0.
• If the result is unordered (in case of error), then the set bits are: 1, 1, 1.

This is how C3/C2/C0 bits are located in the AX register:

\[
\begin{array}{ccc}
14 & 10 & 9 & 8 \\
C3 & C2 & C1 & C0
\end{array}
\]

This is how C3/C2/C0 bits are located in the AH register:

\[
\begin{array}{ccc}
6 & 2 & 1 & 0 \\
C3 & C2 & C1 & C0
\end{array}
\]

After the execution of test ah,\(^{1,12}\) only C0 and C2 bits (on 0 and 2 position) are considered, all other bits are just ignored.

Now let’s talk about the parity flag, another notable historical rudiment.

---

\(^{111}\) Intel P6 is Pentium Pro, Pentium II, etc.
\(^{112}\) 5=101b
This flag is set to 1 if the number of ones in the result of the last calculation is even, and to 0 if it is odd. Let’s look into Wikipedia$^{113}$:

One common reason to test the parity flag actually has nothing to do with parity. The FPU has four condition flags (C0 to C3), but they cannot be tested directly, and must instead be first copied to the flags register. When this happens, C0 is placed in the carry flag, C2 in the parity flag and C3 in the zero flag. The C2 flag is set when e.g. incomparable floating point values (NaN or unsupported format) are compared with the FUCOM instructions.

As noted in Wikipedia, the parity flag used sometimes in FPU code, let’s see how.

The PF flag is to be set to 1 if both C0 and C2 are set to 0 or both are 1, in which case the subsequent JP (jump if PF==1) is triggering. If we recall the values of C3/C2/C0 for various cases, we can see that the conditional jump JP is triggering in two cases: if \( b > a \) or \( a = b \) (C3 bit is not considered here, since it has been cleared by the test \( ah, 5 \) instruction).

It is all simple after that. If the conditional jump has been triggered, FLD loads the value of \(_b\) in ST(0), and if it hasn’t been triggered, the value of \(_a\) is loaded there.

**And what about checking C2?**

The C2 flag is set in case of error (NaN, etc.), but our code doesn’t check it.

If the programmer cares about FPU errors, he/she must add additional checks.

---

$^{113}$https://en.wikipedia.org/wiki/Parity_flag
First OllyDbg example: a=1.2 and b=3.4

Let’s load the example into OllyDbg:

Current arguments of the function: $a = 1.2$ and $b = 3.4$ (We can see them in the stack: two pairs of 32-bit values). $b (3.4)$ is already loaded in ST($0$). Now FCOMP is being executed. OllyDbg shows the second FCOMP argument, which is in stack right now.
FCOMP has been executed:

Figure 1.69: OllyDbg: FCOMP has been executed

We see the state of the FPU’s condition flags: all zeros. The popped value is reflected as ST(7), it was written earlier about reason for this: 1.25.5 on page 224.
Figure 1.70: OllyDbg: FNSTSW has been executed

We see that the AX register contain zeros: indeed, all condition flags are zero. (OllyDbg disassembles the FNSTSW instruction as FSTSW—they are synonyms).
TEST has been executed:

The PF flag is set to 1.

Indeed: the number of bits set in 0 is 0 and 0 is an even number. OllyDbg disassembles JP as JPE\textsuperscript{114}—they are synonyms. And it is about to trigger now.

\textsuperscript{114}Jump Parity Even (x86 instruction)
JPE triggered, FLD loads the value of $b \times (3.4)$ in ST(0):

![CPU - main thread, module d_max](image)

Figure 1.72: OllyDbg: the second FLD has been executed

The function finishes its work.
Second OllyDbg example: \( a=5.6 \) and \( b=-4 \)

Let’s load example into OllyDbg:

![Figure 1.73: OllyDbg: first FLD executed](image)

Current function arguments: \( a = 5.6 \) and \( b = -4 \). \( b \) \((-4)\) is already loaded in \( ST(0) \). FCOMP about to execute now. OllyDbg shows the second FCOMP argument, which is in stack right now.
Figure 1.74: OllyDbg: FCOMP executed

We see the state of the FPU’s condition flags: all zeros except C0.
We see that the AX register contains 0x100: the C0 flag is at the 8th bit.
The PF flag is cleared. Indeed:
the count of bits set in 0x100 is 1 and 1 is an odd number. JPE is being skipped now.
JPE hasn't been triggered, so FLD loads the value of a (5.6) in ST(0):

The function finishes its work.

**Optimizing MSVC 2010**

Listing 1.215: Optimizing MSVC 2010

```assembly
a$ = 8 ; size = 8
b$ = 16 ; size = 8
d_max PROC
    fld QWORD PTR _b$[esp-4]
    fld QWORD PTR _a$[esp-4]
    ; current stack state: ST(0) = _a, ST(1) = _b
    fcom ST(0) ; compare _a and ST(1) = (_b)
    fnstsw ax
    test ah, 65 ; 00000001H
    jne SHORT $LN5@d_max
    ; copy ST(0) to ST(1) and pop register,
    ; leave (_a) on top
    fstp ST(1)
    ; current stack state: ST(0) = _a
    ret 0

$LN5@d_max:
    ; copy ST(0) to ST(0) and pop register,
    ; leave (_b) on top
```

Figure 1.77: OllyDbg: second FLD executed

The function finishes its work.
FCOM differs from FCOMP in the sense that it just compares the values and doesn’t change the FPU stack. Unlike the previous example, here the operands are in reverse order, which is why the result of the comparison in C3/C2/C0 is different:

- If \( a > b \) in our example, then C3/C2/C0 bits are to be set as: 0, 0, 0.
- If \( b > a \), then the bits are: 0, 0, 1.
- If \( a = b \), then the bits are: 1, 0, 0.

The `test ah, 65` instruction leaves just two bits —C3 and C0. Both will be zero if \( a > b \): in that case the JNE jump will not be triggered. Then `FSTP ST(1)` follows —this instruction copies the value from ST(0) to the operand and pops one value from the FPU stack. In other words, the instruction copies ST(0) (where the value of \_a is now) into ST(1). After that, two copies of \_a are at the top of the stack. Then, one value is popped. After that, ST(0) contains \_a and the function is finishes.

The conditional jump JNE is triggering in two cases: if \( b > a \) or \( a = b \). ST(0) is copied into ST(0), it is just like an idle (`NOP`) operation, then one value is popped from the stack and the top of the stack (ST(0)) is contain what has been in ST(1) before (that is \_b). Then the function finishes. The reason this instruction is used here probably is because the FPU has no other instruction to pop a value from the stack and discard it.
First OllyDbg example: \( a=1.2 \) and \( b=3.4 \)

Both FLD are executed:

![OllyDbg screenshot showing FLD and FCOM instructions](image)

**Figure 1.78: OllyDbg: both FLD are executed**

FCOM being executed: OllyDbg shows the contents of ST(0) and ST(1) for convenience.
FCOM has been executed:

![OllyDbg screenshot](image)

C0 is set, all other condition flags are cleared.

---

**Figure 1.79:** OllyDbg: FCOM has been executed

C0 is set, all other condition flags are cleared.
FNSTSW has been executed, AX=0x3100:

Figure 1.80: OllyDbg: FNSTSW is executed
Figure 1.81: OllyDbg: TEST is executed

ZF=0, conditional jump is about to trigger now.
FSTP ST (or FSTP ST(0)) has been executed —1.2 has been popped from the stack, and 3.4 was left on top:

![CPU - main thread, module d_max](image)

**Figure 1.82: OllyDbg: FSTP is executed**

We see that the FSTP ST instruction works just like popping one value from the FPU stack.
Second OllyDbg example: a=5.6 and b=-4

Both FLD are executed:

![OllyDbg screenshot with instruction addresses and values]

Figure 1.83: OllyDbg: both FLD are executed

FCOM is about to execute.

249
FCOM has been executed:

Figure 1.84: OllyDbg: FCOM is finished

All conditional flags are cleared.
Figure 1.85: OllyDbg: FNSTSW has been executed
TEST has been executed:

Figure 1.86: OllyDbg: TEST has been executed

ZF=1, jump will not happen now.
FSTP ST(1) has been executed: a value of 5.6 is now at the top of the FPU stack.

Figure 1.87: OllyDbg: FSTP has been executed

We now see that the FSTP ST(1) instruction works as follows: it leaves what has been at the top of the stack, but clears ST(1).

**GCC 4.4.1**

Listing 1.216: GCC 4.4.1

d_max proc near
b = qword ptr -10h
a = qword ptr -8
a_first_half = dword ptr 8
a_second_half = dword ptr 0Ch
b_first_half = dword ptr 10h
b_second_half = dword ptr 14h

    push ebp
    mov ebp, esp
    sub esp, 10h

; put a and b to local stack:

    mov eax, [ebp+a_first_half]
    mov dword ptr [ebp+a], eax
    mov eax, [ebp+a_second_half]
    mov dword ptr [ebp+a+4], eax
    mov eax, [ebp+b_first_half]
    mov dword ptr [ebp+b], eax

    pop ebp
    ret

d_max endp
mov    eax, [ebp+b_second_half]
mov    dword ptr [ebp+b+4], eax

; load a and b to FPU stack:
    fld    [ebp+a]
    fld    [ebp+b]

; current stack state: ST(0) - b; ST(1) - a
    fxch   st(1); this instruction swaps ST(1) and ST(0)
; current stack state: ST(0) - a; ST(1) - b
    fucompp ; compare a and b and pop two values from stack, i.e., a and b
    fnstsw  ax ; store FPU status to AX
    sahf    ; load SF, ZF, AF, PF, and CF flags state from AH
    setnbe  al ; store 1 to AL, if CF=0 and ZF=0
    test   al, al ; AL==0 ?
    jz     short loc_8048453 ; yes
    fld    [ebp+a]
    jmp    short locret_8048456

loc_8048453:
    fld    [ebp+b]
locret_8048456:
    leave
    retn
d_max   endp

FUCOMPP is almost like FCOM, but pops both values from the stack and handles “not-a-numbers” differently.

A bit about not-a-numbers.

The FPU is able to deal with special values which are not-a-numbers or NaNs. These are infinity, result of division by 0, etc. Not-a-numbers can be “quiet” and “signaling”. It is possible to continue to work with "quiet" NaNs, but if one tries to do any operation with “signaling” NaNs, an exception is to be raised.

FCOM raises an exception if any operand is NaN. FUCOM raises an exception only if any operand is a signaling NaN (SNaN).

The next instruction is SAHF (Store AH into Flags) —this is a rare instruction in code not related to the FPU. 8 bits from AH are moved into the lower 8 bits of the CPU flags in the following order:

```
| SF | ZF | AF | PF | CF |
```

Let’s recall that FNSTSW moves the bits that interest us (C3/C2/C0) into AH and they are in positions 6, 2, 0 of the AH register:

```
| C3 | C2 | C1 | C0 |
```

In other words, the fnstsw ax / sahf instruction pair moves C3/C2/C0 into ZF, PF and CF.

Now let’s also recall the values of C3/C2/C0 in different conditions:

- If \( a > b \) in our example, then C3/C2/C0 are to be set to: 0, 0, 0.
- If \( a < b \), then the bits are to be set to: 0, 0, 1.
- If \( a = b \), then: 1, 0, 0.

In other words, these states of the CPU flags are possible after three FUCOMPP/FNSTSW/SAHF instructions:

- If \( a > b \), the CPU flags are to be set as: ZF=0, PF=0, CF=0.
- If \( a < b \), then the flags are to be set as: ZF=0, PF=0, CF=1.
- And if \( a = b \), then: ZF=1, PF=0, CF=0.
Depending on the CPU flags and conditions, SETNBE stores 1 or 0 to AL. It is almost the counterpart of JNBE, with the exception that SETcc stores 1 or 0 in AL, but Jcc does actually jump or not. SETNBE stores 1 only if CF=0 and ZF=0. If it is not true, 0 is to be stored into AL.

Only in one case both CF and ZF are 0: if \(a > b\).

Then 1 is to be stored to AL, the subsequent JZ is not to be triggered and the function will return \(a\). In all other cases, \(b\) is to be returned.

**Optimizing GCC 4.4.1**

<table>
<thead>
<tr>
<th>public d_max</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_max proc near</td>
</tr>
<tr>
<td>arg_0 = qword ptr 8</td>
</tr>
<tr>
<td>arg_8 = qword ptr 10h</td>
</tr>
<tr>
<td>push ebp</td>
</tr>
<tr>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>fld [ebp+arg_0]; _a</td>
</tr>
<tr>
<td>fld [ebp+arg_8]; _b</td>
</tr>
<tr>
<td>; stack state now: ST(0) = _b, ST(1) = _a</td>
</tr>
<tr>
<td>fxch st(1)</td>
</tr>
<tr>
<td>; stack state now: ST(0) = _a, ST(1) = _b</td>
</tr>
<tr>
<td>fucom st(1); compare _a and _b</td>
</tr>
<tr>
<td>fnstsw ax</td>
</tr>
<tr>
<td>sahf</td>
</tr>
<tr>
<td>ja short loc_8048448</td>
</tr>
<tr>
<td>; store ST(0) to ST(0) (idle operation),</td>
</tr>
<tr>
<td>; pop value at top of stack,</td>
</tr>
<tr>
<td>; leave _b at top</td>
</tr>
<tr>
<td>fstp st</td>
</tr>
<tr>
<td>jmp short loc_804844A</td>
</tr>
</tbody>
</table>

**loc_8048448:**

| ; store _a to ST(1), pop value at top of stack, leave _a at top |
| fstp st(1) |
| loc_804844A: |
| ; pop ebp |
| retn |
| d_max endp |

It is almost the same except that s used after SAHF. Actually, conditional jump instructions that check “larger”, “lesser” or “equal” for unsigned number comparison (these are JA, JAE, JB, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) check only flags CF and ZF.

Let’s recall where bits C3/C2/C0 are located in the AH register after the execution of FSTSW/FNSTSW:

| 6 2 1 0 |
| C3 C2 C1 C0 |

Let’s also recall, how the bits from AH are stored into the CPU flags after the execution of SAHF:

| 7 6 4 2 0 |
| SF ZF AF PF CF |

After the comparison, the C3 and C0 bits are moved into ZF and CF, so the conditional jumps are able work after. s triggering if both CF are ZF zero.

Thereby, the conditional jumps instructions listed here can be used after a FNSTSW/SAHF instruction pair.

\(^{115}\text{cc is condition code}\)
Apparently, the FPU C3/C2/C0 status bits were placed there intentionally, to easily map them to base CPU flags without additional permutations?

**GCC 4.8.1 with -O3 optimization turned on**

Some new FPU instructions were added in the P6 Intel family\(^{116}\). These are FUCOMI (compare operands and set flags of the main CPU) and FCMOVcc (works like CMOVcc, but on FPU registers).

Apparently, the maintainers of GCC decided to drop support of pre-P6 Intel CPUs (early Pentiums, 80486, etc.).

And also, the FPU is no longer separate unit in P6 Intel family, so now it is possible to modify/check flags of the main CPU from the FPU.

So what we get is:

---

**Listing 1.218: Optimizing GCC 4.8.1**

```assembly
fld QWORD PTR [esp+4] ; load "a"
fld QWORD PTR [esp+12] ; load "b"
; ST0=b, ST1=a
fxch st(1)
; ST0=a, ST1=b
; compare "a" and "b"
fucomi st, st(1)
; copy ST1 ("b" here) to ST0 if a<=b
; leave "a" in ST0 otherwise
fcmovbe st, st(1)
; discard value in ST1
fstp st(1)
ret
```

---

Hard to guess why FXCH (swap operands) is here.

It’s possible to get rid of it easily by swapping the first two FLD instructions or by replacing FCMOVBE (below or equal) by FCMOVA (above). Probably it’s a compiler inaccuracy.

So FUCOMI compares ST(0) (a) and ST(1) (b) and then sets some flags in the main CPU. FCMOVBE checks the flags and copies ST(1) (b here at the moment) to ST(0) (a here) if ST0(a) <= ST1(b). Otherwise (a > b), it leaves a in ST(0).

The last FSTP leaves ST(0) on top of the stack, discarding the contents of ST(1).

Let’s trace this function in GDB:

---

**Listing 1.219: Optimizing GCC 4.8.1 and GDB**

```
dennis@ubuntuvm:~/polygon$ gcc -O3 d_max.c -o d_max -fno-inline
dennis@ubuntuvm:~/polygon$ gdb d_max
GNU gdb (GDB) 7.6.1-ubuntu
...  
Reading symbols from /home/dennis/polygon/d_max...(no debugging symbols found)...done.
(gdb) b d_max
Breakpoint 1 at 0x80484a0
(gdb) run
Starting program: /home/dennis/polygon/d_max
11
Breakpoint 1, 0x080484a0 in d_max ()
(gdb) ni
15
Dump of assembler code for function d_max:
0x080484a0 <+0>: fldl 0x4(%esp)
0x080484a4 <+4>: fldl 0xc(%esp)
0x080484a8 <+8>: fxch %st(1)
0x080484aa <+10>: fucomi %st(1),%st
0x080484ac <+12>: fcmovbe %st(1),%st
0x080484ae <+14>: fstp %st(1)
0x080484b0 <+16>: ret
```

---

\(^{116}\)Starting at Pentium Pro, Pentium-II, etc.
(gdb) ni
0x080484a8 in d_max()
(gdb) info float
R7: Valid 0x3fff9999999999999800 +1.199999999999999956
=>R6: Valid 0x4000d999999999999800 +3.399999999999999911
R5: Empty 0x00000000000000000000
R4: Empty 0x00000000000000000000
R3: Empty 0x00000000000000000000
R2: Empty 0x00000000000000000000
R1: Empty 0x00000000000000000000
R0: Empty 0x00000000000000000000

Status Word: 0x3000
TOP: 6
Control Word: 0x037f IM DM ZM OM UM PM
PC: Extended Precision (64-bits)
RC: Round to nearest
Tag Word: 0x0fff
Instruction Pointer: 0x73:0x080484a8
Operand Pointer: 0x7b:0xbffff118
Opcode: 0x0000
(gdb) ni
0x080484ac in d_max()
(gdb) info float
R7: Valid 0x4000d999999999999800 +3.399999999999999911
=>R6: Valid 0x3fff9999999999999800 +1.199999999999999956
R5: Empty 0x00000000000000000000
R4: Empty 0x00000000000000000000
R3: Empty 0x00000000000000000000
R2: Empty 0x00000000000000000000
R1: Empty 0x00000000000000000000
R0: Empty 0x00000000000000000000

Status Word: 0x3000
TOP: 6
Control Word: 0x037f IM DM ZM OM UM PM
PC: Extended Precision (64-bits)
RC: Round to nearest
Tag Word: 0x0fff
Instruction Pointer: 0x73:0x080484a4
Operand Pointer: 0x7b:0xbffff118
Opcode: 0x0000
(gdb) disas $eip
Dump of assembler code for function d_max:
0x080484a0 <+0>: fild 0x4(%esp)
0x080484a4 <+4>: fild 0xc(%esp)
0x080484a8 <+8>: fxch %st(1)
=> 0x080484ac <+10>: fucomi %st(1),%st
0x080484ac <+12>: fcmovbe %st(1),%st
0x080484ae <+14>: fstp %st(1)
0x080484b0 <+16>: ret
End of assembler dump.
(gdb) ni
0x080484ac in d_max()
(gdb) info registers
eax 0x1 1
ecx 0xbffff1c4 -1073745468
edx 0x8048340 134513472
ebx 0xb7fbf000 -1208225792
esp 0xbffff10c 0xbffff10c
ebp 0xbffff128 0xbffff128
esi 0x0 0
edi 0x0 0
eip 0x80484ac 0x80484ac <d_max+12>
eflags 0x203 [ CF IF ]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51
(gdb) ni
0x080484ae in d_max ()
(gdb) info float
R7: Valid 0x4000d9999999999999999 +3.399999999999999911
Rightarrow: Valid 0x4000d9999999999999999 +3.399999999999999911
R5: Empty 0x00000000000000000000000000000000
R4: Empty 0x00000000000000000000000000000000
R3: Empty 0x00000000000000000000000000000000
R2: Empty 0x00000000000000000000000000000000
R1: Empty 0x00000000000000000000000000000000
R0: Empty 0x00000000000000000000000000000000
Status Word: 0x3000
TOP: 6
Control Word: 0x037f IM DM ZM OM UM PM
PC: Extended Precision (64-bits)
RC: Round to nearest
Tag Word: 0x0fff
Instruction Pointer: 0x73:0x080484ac
Operand Pointer: 0x7b:0xbffff118
Opcode: 0x0000
(gdb) disas $eip
Dump of assembler code for function d_max:
0x080484a0 <+0>: fldl 0x4(%esp)
0x080484a4 <+4>: fldl 0xc(%esp)
0x080484a8 <+8>: fxch %st(1)
0x080484aa <+10>: fucomi %st(1),%st
0x080484ac <+12>: fcmovbe %st(1),%st
=> 0x080484ae <+14>: fstp %st(1)
0x080484b0 <+16>: ret
End of assembler dump.
(gdb) ni
0x080484b0 in d_max ()
(gdb) info float
R7: Valid 0x4000d9999999999999999 +3.399999999999999911
R6: Empty 0x00000000000000000000000000000000
R5: Empty 0x00000000000000000000000000000000
R4: Empty 0x00000000000000000000000000000000
R3: Empty 0x00000000000000000000000000000000
R2: Empty 0x00000000000000000000000000000000
R1: Empty 0x00000000000000000000000000000000
R0: Empty 0x00000000000000000000000000000000
Status Word: 0x3000
TOP: 7
Control Word: 0x037f IM DM ZM OM UM PM
PC: Extended Precision (64-bits)
RC: Round to nearest
Tag Word: 0x3fff
Instruction Pointer: 0x73:0x080484ac
Operand Pointer: 0x7b:0xbffff118
Opcode: 0x0000
(gdb) quit
A debugging session is active.
Quit anyway? (y or n) y
dennis@ubuntuvm:~/polygon$

Using "ni", let's execute the first two FLD instructions.
Let's examine the FPU registers (line 33).
As it was mentioned before, the FPU registers set is a circular buffer rather than a stack (1.25.5 on page 224). And GDB doesn't show STx registers, but internal the FPU registers (Rx). The arrow (at line 35) points to the current top of the stack.
You can also see the TOP register contents in Status Word (line 36-37)—it is 6 now, so the stack top is now
pointing to internal register 6.

The values of \(a\) and \(b\) are swapped after FXCH is executed (line 54).

FUCOMI is executed (line 83). Let’s see the flags: CF is set (line 95).

FCMOVBE has copied the value of \(b\) (see line 104).

FSTP leaves one value at the top of stack (line 139). The value of TOP is now 7, so the FPU stack top is pointing to internal register 7.

**ARM**

**Optimizing Xcode 4.6.3 (LLVM) (ARM mode)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV</td>
<td>D16, R2, R3</td>
</tr>
<tr>
<td>VMOV</td>
<td>D17, R0, R1</td>
</tr>
<tr>
<td>VCMPE.F64</td>
<td>D17, D16</td>
</tr>
<tr>
<td>VMRS</td>
<td>APSR_nzcv, FPSCR</td>
</tr>
<tr>
<td>VMOVGT.F64</td>
<td>D16, D17</td>
</tr>
<tr>
<td>VMOV</td>
<td>R0, R1, D16</td>
</tr>
<tr>
<td>BX</td>
<td>LR</td>
</tr>
</tbody>
</table>

A very simple case. The input values are placed into the D17 and D16 registers and then compared using the VCMPE instruction.

Just like in the x86 coprocessor, the ARM coprocessor has its own status and flags register \((\text{FPSCR})^{117}\), since there is a necessity to store coprocessor-specific flags. And just like in x86, there are no conditional jump instruction in ARM, that can check bits in the status register of the coprocessor. So there is VMRS, which copies 4 bits \((N, Z, C, V)\) from the coprocessor status word into bits of the general status register \((\text{APSR})^{118}\).

VMOVGT is the analog of the MOVGT, instruction for D-registers, it executes if one operand is greater than the other while comparing \((GT—\text{Greater Than})\).

If it gets executed, the value of \(a\) is to be written into D16 (that is currently stored in D17). Otherwise the value of \(b\) stays in the D16 register.

The penultimate instruction VMOV prepares the value in the D16 register for returning it via the R0 and R1 register pair.

**Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV</td>
<td>D16, R2, R3</td>
</tr>
<tr>
<td>VMOV</td>
<td>D17, R0, R1</td>
</tr>
<tr>
<td>VCMPE.F64</td>
<td>D17, D16</td>
</tr>
<tr>
<td>VMRS</td>
<td>APSR_nzcv, FPSCR</td>
</tr>
<tr>
<td>IT GT</td>
<td></td>
</tr>
<tr>
<td>VMOVGT.F64</td>
<td>D16, D17</td>
</tr>
<tr>
<td>VMOV</td>
<td>R0, R1, D16</td>
</tr>
<tr>
<td>BX</td>
<td>LR</td>
</tr>
</tbody>
</table>

Almost the same as in the previous example, however slightly different. As we already know, many instructions in ARM mode can be supplemented by condition predicate. But there is no such thing in Thumb mode. There is no space in the 16-bit instructions for 4 more bits in which conditions can be encoded.

However, Thumb-2 was extended to make it possible to specify predicates to old Thumb instructions. Here, in the IDA-generated listing, we see the VMOVGT instruction, as in previous example.

In fact, the usual VMOV is encoded there, but IDA adds the -GT suffix to it, since there is a IT GT instruction placed right before it.

---

117(ARM) Floating-Point Status and Control Register  
118(ARM) Application Program Status Register
The IT instruction defines a so-called if-then block.

After the instruction it is possible to place up to 4 instructions, each of them has a predicate suffix. In our example, IT GT implies that the next instruction is to be executed, if the GT (Greater Than) condition is true.

Here is a more complex code fragment, by the way, from Angry Birds (for iOS):

Listing 1.222: Angry Birds Classic

```assembly
ITE NE
VMOVNE R2, R3, D16
VMOVEQ R2, R3, D17
BLX _objc_msgSend; not suffixed
```

ITE stands for if-then-else
and it encodes suffixes for the next two instructions.
The first instruction executes if the condition encoded in ITE (NE, not equal) is true at, and the second—if the condition is not true. (The inverse condition of NE is EQ (equal)).
The instruction followed after the second VMOV (or VMOVEQ) is a normal one, not suffixed (BLX).

One more that’s slightly harder, which is also from Angry Birds:

Listing 1.223: Angry Birds Classic

```assembly
ITTTT EQ
MOVEQ R0, R4
ADDEQ SP, SP, #0x20
POPEQ {R8,R10}
POPEQ {R4-R7,PC}
BLX ___stack_chk_fail; not suffixed
```

Four “T” symbols in the instruction mnemonic mean that the four subsequent instructions are to be executed if the condition is true.

That’s why IDA adds the -EQ suffix to each one of them.
And if there was, for example, ITEEE EQ (if-then-else-else-else), then the suffixes would have been set as follows:

- EQ
- NE
- NE
- NE

Another fragment from Angry Birds:

Listing 1.224: Angry Birds Classic

```assembly
CMP.W R0, #0xFFFFFFFF
ITTE LE
SUBLE.W R10, R0, #1
NEGLE R0, R0
MOVGT R10, R0
MOVSG R6, #0; not suffixed
CBZ R0, loc_1E7E32; not suffixed
```

ITTE (if-then-then-else)
implies that the 1st and 2nd instructions are to be executed if the LE (Less or Equal) condition is true, and the 3rd—if the inverse condition (GT—Greater Than) is true.

Compilers usually don’t generate all possible combinations.
For example, in the mentioned Angry Birds game (classic version for iOS) only these variants of the IT instruction are used: IT, ITE, ITT, ITTE, ITTT, ITTTT. How to learn this? In IDA, it is possible to produce
listing files, so it was created with an option to show 4 bytes for each opcode. Then, knowing the high part of the 16-bit opcode (IT is 0xBF), we do the following using grep:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

By the way, if you program in ARM assembly language manually for Thumb-2 mode, and you add conditional suffixes, the assembler will add the IT instructions automatically with the required flags where it is necessary.

**Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)**

Listing 1.225: Non-optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
b = -0x20
a = -0x18
val_to_return = -0x10
saved_R7 = -4

STR R7, [SP,#saved_R7]!
MOV R7, SP
SUB SP, SP, #0x1C
BIC SP, SP, #7
VMOV D16, R2, R3
VMOV D17, R0, R1
VSTR D17, [SP,#0x20+a]
VSTR D16, [SP,#0x20+b]
VLDR D17, [SP,#0x20+a]
VLDR D16, [SP,#0x20+b]
VCMPE.F64 D16, D17
VMRS APSR_nzcv, FPSCR
BLE loc_2E08
VLDR D16, [SP,#0x20+a]
VSTR D16, [SP,#0x20+val_to_return]
B loc_2E10

loc_2E08
VLDR D16, [SP,#0x20+b]
VSTR D16, [SP,#0x20+val_to_return]

loc_2E10
VLDR D16, [SP,#0x20+val_to_return]
VMOV R0, R1, D16
MOV SP, R7
LDR R7, [SP+0x20+b],#4
BX LR
```

Almost the same as we already saw, but there is too much redundant code because the \( a \) and \( b \) variables are stored in the local stack, as well as the return value.

**Optimizing Keil 6/2013 (Thumb mode)**

Listing 1.226: Optimizing Keil 6/2013 (Thumb mode)

```
PUSH {R3-R7,LR}
MOVS R4, R2
MOVS R5, R3
MOVS R6, R0
MOVS R7, R1
BL __aeabi_cdrcmple
BCS loc_1C0
MOVS R0, R6
MOVS R1, R7
POP {R3-R7,PC}

loc_1C0
MOVS R0, R4
MOVS R1, R5
POP {R3-R7,PC}
```
Keil doesn’t generate FPU-instructions since it cannot rely on them being supported on the target CPU, and it cannot be done by straightforward bitwise comparing. So it calls an external library function to do the comparison: 

```
__aeabi_cdrcmple
```

N.B. The result of the comparison is to be left in the flags by this function, so the following BCS (Carry set—Greater than or equal) instruction can work without any additional code.

**ARM64**

Optimizing GCC (Linaro) 4.9

```
d_max:
; D0 - a, D1 - b
    fcmpe   d0, d1
    fcSEL   d0, d0, d1, gt
; now result in D0
    ret
```

The ARM64 ISA has FPU-instructions which set APSR the CPU flags instead of FPSCR for convenience. The FPU is not a separate device here anymore (at least, logically). Here we see FCMPE. It compares the two values passed in D0 and D1 (which are the first and second arguments of the function) and sets APSR flags (N, Z, C, V).

FCSEL (Floating Conditional Select) copies the value of D0 or D1 into D0 depending on the condition (GT—Greater Than), and again, it uses flags in APSR register instead of FPSCR.

This is much more convenient, compared to the instruction set in older CPUs.

If the condition is true (GT), then the value of D0 is copied into D0 (i.e., nothing happens). If the condition is not true, the value of D1 is copied into D0.

Non-optimizing GCC (Linaro) 4.9

```
d_max:
; save input arguments in "Register Save Area"
    sub    sp, sp, #16
    str    d0, [sp,8]
    str    d1, [sp]
; reload values
    ldr    x1, [sp,8]
    ldr    x0, [sp]
    fmov   d0, x1
    fmov   d1, x0
; D0 - a, D1 - b
    fcmpe   d0, d1
    ble    .L76
; a>b; load D0 (a) into X0
    ldr    x0, [sp,8]
    b      .L74
.L76:
; a<=b; load D1 (b) into X0
    ldr    x0, [sp]
.L74:
; result in X0
    fmov   d0, x0
; result in D0
    add    sp, sp, 16
    ret
```

Non-optimizing GCC is more verbose.

First, the function saves its input argument values in the local stack (Register Save Area). Then the code reloads these values into registers X0/X1 and finally copies them to D0/D1 to be compared using FCMPE. A lot of redundant code, but that is how non-optimizing compilers work. FCMPE compares the values and sets the APSR flags. At this moment, the compiler is not thinking yet about the more convenient FCSEL instruction, so it proceed using old methods: using the BLE instruction (Branch if Less than or Equal). In the first case ($a > b$), the value of $a$ gets loaded into X0. In the other case ($a <= b$), the value of $b$ gets
loaded into X0. Finally, the value from X0 gets copied into D0, because the return value needs to be in this register.

**Exercise**

As an exercise, you can try optimizing this piece of code manually by removing redundant instructions and not introducing new ones (including FCSEL).

**Optimizing GCC (Linaro) 4.9—float**

Let’s also rewrite this example to use *float* instead of *double*.

```c
float f_max (float a, float b) {
  if (a>b)
    return a;
  return b;
}
```

It is the same code, but the S-registers are used instead of D- ones. It’s because numbers of type *float* are passed in 32-bit S-registers (which are in fact the lower parts of the 64-bit D-registers).

**MIPS**

The co-processor of the MIPS processor has a condition bit which can be set in the FPU and checked in the CPU.

Earlier MIPS-es have only one condition bit (called FCC0), later ones have 8 (called FCC7-FCC0). This bit (or bits) are located in the register called FCCR.

Listing 1.227: Optimizing GCC 4.4.5 (IDA)

```mips
f_max: 
; S0 - a, S1 - b 
    fcmpe  $s0, $s1 
    fcsel  $s0, $s0, $s1, gt 
; now result in S0 
    ret
```

C.LT.D compares two values. LT is the condition “Less Than”. D implies values of type *double*. Depending on the result of the comparison, the FCC0 condition bit is either set or cleared.

BC1T checks the FCC0 bit and jumps if the bit is set. T means that the jump is to be taken if the bit is set ("True"). There is also the instruction BC1F which jumps if the bit is cleared ("False"). Depending on the jump, one of function arguments is placed into $F0.
1.25.8 Some constants

It's easy to find representations of some constants in Wikipedia for IEEE 754 encoded numbers. It's interesting to know that 0.0 in IEEE 754 is represented as 32 zero bits (for single precision) or 64 zero bits (for double). So in order to set a floating point variable to 0.0 in register or memory, one can use MOV or XOR reg, reg instruction. This is suitable for structures where many variables present of various data types. With usual memset() function one can set all integer variables to 0, all boolean variables to false, all pointers to NULL, and all floating point variables (of any precision) to 0.0.

1.25.9 Copying

One may think inertially that FLD/FST instructions must be used to load and store (and hence, copy) IEEE 754 values. Nevertheless, same can be achieved easier by usual MOV instruction, which, of course, copies values bitwisely.

1.25.10 Stack, calculators and reverse Polish notation

Now we understand why some old programmable calculators use reverse Polish notation. For example, for addition of 12 and 34 one has to enter 12, then 34, then press “plus” sign. It’s because old calculators were just stack machine implementations, and this was much simpler than to handle complex parenthesized expressions.

Such a calculator still present in many Unix distributions: dc.

1.25.11 80 bits?

Internal numbers representation in FPU — 80-bit. Strange number, because the number not in $2^n$ form. There is a hypothesis that this is probably due to historical reasons—the standard IBM punched card can encode 12 rows of 80 bits. 80 · 25 text mode resolution was also popular in past. Wikipedia has another explanation: https://en.wikipedia.org/wiki/Extended_precision.

If you know better, please a drop email to the author: <first_name @ last_name . com> or <first_name . last_name @ gmail . com>.

1.25.12 x64

On how floating point numbers are processed in x86-64, read more here: 1.38 on page 425.

1.25.13 Exercises

- http://challenges.re/60
- http://challenges.re/61

1.26 Arrays

An array is just a set of variables in memory that lie next to each other and that have the same type

1.26.1 Simple example

```c
#include <stdio.h>

int main()
{
    int a[20];
    int i;
    for (i=0; i<20; i++)
        a[i]=i*i;
    for (i=0; i<20; i++)
```

[119] AKA “homogeneous container”
printf ("a[%d]=%d\n", i, a[i]);
return 0;
}

x86
MSVC

Let's compile:

Listing 1.228: MSVC 2008

```
_TEXT SEGMENT
_i$ = -84 ; size = 4
_a$ = -80 ; size = 80

_main PROC
    push ebp
    mov ebp, esp
    sub esp, 84 ; 00000054H
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN5@main
$LN5@main:
    mov eax, DWORD PTR _i$[ebp]
    add eax, 1
    mov DWORD PTR _i$[ebp], eax

$LN6@main:
    cmp DWORD PTR _i$[ebp], 20 ; 00000014H
    jge SHORT $LN4@main
    mov ecx, DWORD PTR _i$[ebp]
    shl ecx, 1
    mov edx, DWORD PTR _i$[ebp]
    mov DWORD PTR _a$[ebp+edx*4], ecx
    jmp SHORT $LN5@main
$LN5@main:
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN3@main
$LN4@main:
    mov eax, DWORD PTR _i$[ebp]
    add eax, 1
    mov DWORD PTR _i$[ebp], eax

$LN3@main:
    cmp DWORD PTR _i$[ebp], 20 ; 00000014H
    jge SHORT $LN1@main
    mov ecx, DWORD PTR _i$[ebp]
    mov edx, DWORD PTR _a$[ebp+ecx*4]
    push edx
    mov eax, DWORD PTR _i$[ebp]
    push eax
    push OFFSET $SG2463
    call _printf
    add esp, 12 ; 000000cH
    jmp SHORT $LN2@main
$LN2@main:
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP
```

Nothing very special, just two loops: the first is a filling loop and second is a printing loop. The `shl ecx, 1` instruction is used for value multiplication by 2 in ECX, more about it: 1.24.2 on page 216.

80 bytes are allocated on the stack for the array, 20 elements of 4 bytes.
Let's try this example in OllyDbg.

We see how the array gets filled:

each element is 32-bit word of int type and its value is the index multiplied by 2:

![Figure 1.88: OllyDbg: after array filling](image)

Since this array is located in the stack, we can see all its 20 elements there.

### GCC

Here is what GCC 4.4.1 does:

**Listing 1.229: GCC 4.4.1**

```assembly
public main
proc near ; DATA XREF: _start+17

var_70 = dword ptr -70h
var_6C = dword ptr -6Ch
var_68 = dword ptr -68h
i_2 = dword ptr -54h
i = dword ptr -4h

push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
```

266
By the way, variable $a$ is of type int* (the pointer to int)—you can pass a pointer to an array to another function, but it’s more correct to say that a pointer to the first element of the array is passed (the addresses of rest of the elements are calculated in an obvious way).

If you index this pointer as $a[idx]$, idx is just to be added to the pointer and the element placed there (to which calculated pointer is pointing) is to be returned.

An interesting example: a string of characters like string is an array of characters and it has a type of const char[].

An index can also be applied to this pointer.

And that is why it is possible to write things like “string[i]”—this is a correct C/C++ expression!

**ARM**

**Non-optimizing Keil 6/2013 (ARM mode)**
int type requires 32 bits for storage (or 4 bytes),
so to store 20 int variables 80 (0x50) bytes are needed. So that is why the SUB SP, SP, #0x50
instruction in the function’s prologue allocates exactly this amount of space in the stack.
In both the first and second loops, the loop iterator \textit{i} is placed in the R4 register.
The number that is to be written into the array is calculated as \textit{i} \times 2, which is effectively equivalent to
shifting it left by one bit,
so MOV R0, R4,LSL#1 instruction does this.
STR R0, [SP,R4,LSL#2] writes the contents of R0 into the array.
Here is how a pointer to array element is calculated: SP points to the start of the array, R4 is \textit{i}.
So shifting \textit{i} left by 2 bits is effectively equivalent to multiplication by 4 (since each array element has a
size of 4 bytes) and then it’s added to the address of the start of the array.
The second loop has an inverse LDR R2, [SP,R4,LSL#2] instruction. It loads the value we need from the
array, and the pointer to it is calculated likewise.

**Optimizing Keil 6/2013 (Thumb mode)**

```
main
PUSH {R4,R5,LR}
; allocate place for 20 int variables + one more variable
SUB SP, SP, #0x54

; first loop
MOVS R0, #0 ; i
MOV R5, SP ; pointer to first array element

loc_ICE
LSLS R1, R0, #1 ; R1=i<<1 (same as i*2)
LSLS R2, R0, #2 ; R2=i<<2 (same as i*4)
ADDS R0, R0, #1 ; i=i+1
CMP R0, #20 ; i<20?
STR R1, [R5,R2] ; store R1 to *(R5+R2) (same R5+i*4)
BLT loc_ICE ; yes, i<20, run loop body again

; second loop
MOVS R4, #0 ; i=0

loc_1DC
LSLS R0, R4, #2 ; R0=i<<2 (same as i*4)
LDR R2, [R5,R0] ; load from *(R5+R0) (same as R5+i*4)
MOVS R1, R4
```
ADR R0, aADD ; "a[%d]=%d\n"
BL __2printf
ADD R4, R4, #1 ; i=i+1
CMP R4, #20 ; i<20?
BLT loc_1DC ; yes, i<20, run loop body again
MOV R0, #0 ; value to return
; deallocate chunk, allocated for 20 int variables + one more variable
ADD SP, SP, #0
POP {R4,R5,PC}

Thumb code is very similar.

Thumb mode has special instructions for bit shifting (like LSL), which calculates the value to be written into the array and the address of each element in the array as well.

The compiler allocates slightly more space in the local stack, however, the last 4 bytes are not used.

Non-optimizing GCC 4.9.1 (ARM64)

Listing 1.230: Non-optimizing GCC 4.9.1 (ARM64)

.LC0:
.string "a[%d]=%d\n"
main:
; save FP and LR in stack frame:
    stp   x29, x30, [sp, -112]!
; set stack frame (FP=SP)
    add   x29, sp, 0
; setting initial counter variable at 0 (WZR is the register always holding zero):
    str   wzr, [x29,108]
; jump to loop condition checking code:
    b    .L2
.L3:
; load value of “i” variable:
    ldr   w0, [x29,108] ; multiplicate it by 2:
    lsl   w2, w0, 1
; find a place of an array in local stack:
    add   x0, x29, 24
; load 32-bit integer from local stack and sign extend it to 64-bit one:
    ldrsw x1, [x29,108]
; calculate address of element (X0+X1<<2=array address+i*4) and store W2 (i*2) there:
    str   w2, [x0,x1,lsl 2]
; increment counter (i):
    ldr   w0, [x29,108]
    add   w0, w0, 1
    str   w0, [x29,108]
.L2:
; check if we finished:
    ldr   w0, [x29,108]
    cmp   w0, 19
; jump to L3 (loop body begin) if not:
    ble   .L3
; second part of the function begins here.
; setting initial counter variable at 0.
; by the way, the same place in the local stack was used for counter, because the same local variable (i) is being used as counter.
    str   wzr, [x29,108]
    b    .L4
.L5:
; calculate array address:
    add   x0, x29, 24
; load “i” value:
    ldrsw x1, [x29,108]
; load value from the array at the address (X0+X1<<2 = address of array + i*4)
    ldr   w2, [x0,x1,lsl 2]
; load address of the “a[%d]=%d\n” string:
    adrpr x0, .LC0
    add   x0, x0, :lo12:.LC0
; load "i" variable to W1 and pass it to printf() as second argument:
  ldr   w1, [x29,108]
; W2 still contains the value of array element which was just loaded.
; call printf():
  bl    printf
; increment "i" variable:
  ldr   w0, [x29,108]
  add   w0, w0, 1
  str   w0, [x29,108]
.L4:
; are we finished?
  ldr   w0, [x29,108]
  cmp   w0, 19
; jump to the loop body begin if not:
  ble   .L5
; return 0
  mov   w0, 0
; restore FP and LR:
  ldp   x29, x30, [sp], 112
  ret

MIPS

The function uses a lot of S- registers which must be preserved, so that's why its values are saved in the
function prologue and restored in the epilogue.

Listing 1.231: Optimizing GCC 4.4.5 (IDA)

main:

var_70  = -0x70
var_68  = -0x68
var_14  = -0x14
var_10  = -0x10
var_C   = -0xC
var_8   = -8
var_4   = -4

; function prologue:
  lui   $gp, (__gnu_local_gp >> 16)
  addiu $sp, -0x80
  la    $gp, (__gnu_local_gp & 0xFFFF)
  sw    $ra, 0x80+var_4($sp)
  sw    $s3, 0x80+var_8($sp)
  sw    $s2, 0x80+var_C($sp)
  sw    $s1, 0x80+var_10($sp)
  sw    $s0, 0x80+var_14($sp)
  sw    $gp, 0x80+var_70($sp)
  addiu $s1, $sp, 0x80+var_68
  move  $v1, $s1
  move  $v0, $zero

; that value will be used as a loop terminator.
; it was precalculated by GCC compiler at compile stage:
  li    $a0, 0x28 # '('

loc_34:

; store value into memory:
  sw    $v0, 0($v1)
; increase value to be stored by 2 at each iteration:
  addiu $v0, 2
; loop terminator reached?
  bne   $v0, $a0, loc_34
; add 4 to address anyway:
  addiu $v1, 4
; array filling loop is ended
; second loop begin
  la    $s3, $LC0     # "a[%d]=%d\n"
; "i" variable will reside in $s0:
  move  $s0, $zero
  li    $s2, 0x14
Something interesting: there are two loops and the first one doesn’t need \( i \), it needs only \( i \times 2 \) (increased by 2 at each iteration) and also the address in memory (increased by 4 at each iteration).

So here we see two variables, one (in \( V0 \)) increasing by 2 each time, and another (in \( V1 \)) — by 4.

The second loop is where \printf{}()\ is called and it reports the value of \( i \) to the user, so there is a variable which is increased by 1 each time (in \( S0 \)) and also a memory address (in \( S1 \)) increased by 4 each time.

That reminds us of loop optimizations: 3.10 on page 490.

Their goal is to get rid of multiplications.

1.26.2 Buffer overflow

Reading outside array bounds

So, array indexing is just \( \text{array}[\text{index}] \). If you study the generated code closely, you’ll probably note the missing index bounds checking, which could check \( \text{if it is less than 20} \). What if the index is 20 or greater? That’s the one C/C++ feature it is often blamed for.

Here is a code that successfully compiles and works:

```c
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    printf("a[20]=%d\n", a[20]);
    return 0;
}
```

Compilation results (MSVC 2008):

```
Listing 1.232: Non-optimizing MSVC 2008

SG2474 DB 'a[20]=%d', 0AH, 00H

_i$ = -84 ; size = 4
```
The code produced this result:

Listing 1.233: OllyDbg: console output

a[20]=1638280

It is just *something* that has been lying in the stack near to the array, 80 bytes away from its first element.
Let's try to find out where did this value come from, using OllyDbg.

Let's load and find the value located right after the last array element:

![OllyDbg screenshot](image)

**Figure 1.89: OllyDbg: reading of the 20th element and execution of printf()**

What is this? Judging by the stack layout, this is the saved value of the EBP register.
Let’s trace further and see how it gets restored:

![Image of OllyDbg interface](image)

Indeed, how it could be different? The compiler may generate some additional code to check the index value to be always in the array’s bounds (like in higher-level programming languages) but this makes the code slower.

**Writing beyond array bounds**

OK, we read some values from the stack illegally, but what if we could write something to it?

Here is what we have got:

```c
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<30; i++)
        a[i]=i;

    return 0;
}
```

---

120Java, Python, etc.
The compiled program crashes after running. No wonder. Let’s see where exactly does it crash.
Let's load it into OllyDbg, and trace until all 30 elements are written:

Figure 1.91: OllyDbg: after restoring the value of EBP
Trace until the function end:

Figure 1.92: OllyDbg: EIP has been restored, but OllyDbg can’t disassemble at 0x15

Now please keep your eyes on the registers.

EIP is 0x15 now. It is not a legal address for code—at least for win32 code! We got there somehow against our will. It is also interesting that the EBP register contain 0x14, ECX and EDX contain 0x1D.

Let’s study stack layout a bit more.

After the control flow has been passed to main(), the value in the EBP register was saved on the stack. Then, 84 bytes were allocated for the array and the i variable. That’s (20+1)*sizeof(int). ESP now points to the _i variable in the local stack and after the execution of the next PUSH something, something is appearing next to _i.

That’s the stack layout while the control is in main():

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>dump</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>00463300</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
<td>0x41</td>
<td>Nu</td>
</tr>
<tr>
<td>00463300</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
<td>0x41</td>
<td>Nu</td>
</tr>
<tr>
<td>00463326</td>
<td>00 04 75 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>0x04</td>
<td>Nu</td>
</tr>
<tr>
<td>00463348</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>0x00</td>
<td>Nu</td>
</tr>
<tr>
<td>00463360</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>0x00</td>
<td>Nu</td>
</tr>
<tr>
<td>00463370</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>0x00</td>
<td>Nu</td>
</tr>
<tr>
<td>00463380</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>0x00</td>
<td>Nu</td>
</tr>
<tr>
<td>00463390</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>0x00</td>
<td>Nu</td>
</tr>
<tr>
<td>00463398</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
<td>0x41</td>
<td>Nu</td>
</tr>
<tr>
<td>004633A0</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
<td>0x41</td>
<td>Nu</td>
</tr>
<tr>
<td>004633A8</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
<td>0x41</td>
<td>Nu</td>
</tr>
<tr>
<td>004633B0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>0x00</td>
<td>Nu</td>
</tr>
<tr>
<td>004633C0</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
<td>0x41</td>
<td>Nu</td>
</tr>
<tr>
<td>004633D0</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
<td>0x41</td>
<td>Nu</td>
</tr>
<tr>
<td>004633E0</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
<td>0x41</td>
<td>Nu</td>
</tr>
<tr>
<td>004633F0</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
<td>0x41</td>
<td>Nu</td>
</tr>
<tr>
<td>00463400</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
<td>0x00</td>
<td>Nu</td>
</tr>
<tr>
<td>00463410</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
<td>0x41</td>
<td>Nu</td>
</tr>
<tr>
<td>a[19]=something statement writes the last int in the bounds of the array (in bounds so far!).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a[20]=something statement writes something to the place where the value of EBP is saved.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Please take a look at the register state at the moment of the crash. In our case, 20 has been written in the 20th element. At the function end, the function epilogue restores the original EBP value. (20 in decimal
is 0x14 in hexadecimal). Then RET gets executed, which is effectively equivalent to POP EIP instruction. The RET instruction takes the return address from the stack (that is the address in CRT, which has called main()), and 21 is stored there (0x15 in hexadecimal). The CPU traps at address 0x15, but there is no executable code there, so exception gets raised.

Welcome! It is called a buffer overflow\textsuperscript{121}.

Replace the \texttt{int} array with a string (\texttt{char} array), create a long string deliberately and pass it to the program, to the function, which doesn't check the length of the string and copies it in a short buffer, and you'll able to point the program to an address to which it must jump. It’s not that simple in reality, but that is how it emerged. Classic article about it: [Aleph One, \textit{Smashing The Stack For Fun And Profit}, (1996)]\textsuperscript{122}.

\textbf{GCC}

Let's try the same code in GCC 4.4.1. We get:

```
public main
main proc near
a = dword ptr -54h
i = dword ptr -4
push ebp
mov ebp, esp
sub esp, 60h ; 96
jmp short loc_80483D1
loc_80483C3: 
mov eax, [ebp+i]
mov edx, [ebp+i]
mov [ebp+eax*4+a], edx
add [ebp+i], 1
loc_80483D1: 
cmp [ebp+i], 1Dh
jle short loc_80483C3
mov eax, 0
leave
retn
main endp
```

Running this in Linux will produce: Segmentation fault.

If we run this in the GDB debugger, we get this:

```
(gdb) r
Starting program: /home/dennis/RE/1
Program received signal SIGSEGV, Segmentation fault.
0x0000000000000016 in ?? ()
(gdb) info registers
eax 0x0 0
ecx 0xd2f96388 -755407992
edx 0x1d 29
ebx 0x26eff4 2551796
esp 0xbffff4b0 0xbffff4b0
ebp 0x15 0x15
esi 0x0 0
edi 0x0 0
eip 0x16 0x16
eflags 0x10202 [ IF RF ]
```

\textsuperscript{121}\textit{wikipedia}

\textsuperscript{122}Also available as \url{http://go.yurichev.com/17266}
The register values are slightly different than in win32 example, since the stack layout is slightly different too.

### 1.26.3 Buffer overflow protection methods

There are several methods to protect against this scourge, regardless of the C/C++ programmers’ negligence. MSVC has options like:

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

One of the methods is to write a random value between the local variables in stack at function prologue and to check it in function epilogue before the function exits. If value is not the same, do not execute the last instruction `RET`, but stop (or hang). The process will halt, but that is much better than a remote attack to your host.

This random value is called a “canary” sometimes, it is related to the miners’ canary, they were used by miners in the past days in order to detect poisonous gases quickly.

Canaries are very sensitive to mine gases, they become very agitated in case of danger, or even die.

If we compile our very simple array example (1.26.1 on page 264) in MSVC with RTC1 and RTCs option, you can see a call to `@_RTC_CheckStackVars@8` a function at the end of the function that checks if the “canary” is correct.

Let’s see how GCC handles this. Let’s take an `alloca()` (1.9.2 on page 34) example:

```c
#include <alloca.h> // GCC
#include <malloc.h> // MSVC
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
    #ifdef __GNUC__
    sprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
    #else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
    #endif
    puts (buf);
}
```

By default, without any additional options, GCC 4.7.3 inserts a “canary” check into the code:

```
Listing 1.235: GCC 4.7.3

.f:
    push   ebp
    mov    ebp, esp
    push   ebx
    sub    esp, 676
    lea    ebx, [esp+39]
    and    ebx, -16
    mov    DWORD PTR [esp+20], 3
    mov    DWORD PTR [esp+16], 2
    mov    DWORD PTR [esp+12], 1
    mov    DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %d, %d\n"
    mov    DWORD PTR [esp+4], 600
    mov    DWORD PTR [esp], ebx
    mov    eax, DWORD PTR gs:20 ; canary
```

---

123 compiler-side buffer overflow protection methods: [wikipedia.org/wiki/Buffer_overflow_protection](https://wikipedia.org/wiki/Buffer_overflow_protection)

124 [wikipedia.org/wiki/Domestic_canary#Miner.27s_canary](https://wikipedia.org/wiki/Domestic_canary#Miner.27s_canary)
The random value is located in gs:20. It gets written on the stack and then at the end of the function the value in the stack is compared with the correct “canary” in gs:20. If the values are not equal, the __stack_chk_fail function is called and we can see in the console something like that (Ubuntu 13.04 x86):

```
*** buffer overflow detected ***: ./2_1 terminated
Backtrace: ========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
/./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935]
```

== Memory map: ==
```
08048000-08049000 r-xp 00000000 08:01 2097586 /home/dennis/2_1
08049000-0804a000 r--p 00000000 08:01 2097586 /home/dennis/2_1
0804a000-0804b000 rw-p 00010000 08:01 2097586 /home/dennis/2_1
094d1000-094f2000 r-p 00000000 00:00 0 [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b757b000-b757c000 r-p 0001a000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b757c000-b757d000 r-p 001b000 08:01 1048602 /lib/i386-linux-gnu/libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781 /lib/i386-linux-gnu/libc-2.17.so
b7740000-b7742000 r-p 001ad000 08:01 1050781 /lib/i386-linux-gnu/libc-2.17.so
b7742000-b7743000 r-p 001af000 08:01 1050781 /lib/i386-linux-gnu/libc-2.17.so
b7743000-b774d000 r-p 00000000 00:00 0
b775a000-b775d000 r-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0 [vdo]
b777e000-b777e000 r-xp 00000000 08:01 1050794 /lib/i386-linux-gnu/ld-2.17.so
b777f000-b777f000 r-p 0001f000 08:01 1050794 /lib/i386-linux-gnu/ld-2.17.so
b7780000-b7780000 r-p 0020000 08:01 1050794 /lib/i386-linux-gnu/ld-2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0 [stack]
```

gs is the so-called segment register. These registers were used widely in MS-DOS and DOS-extenders times. Today, its function is different.

To say it briefly, the gs register in Linux always points to the TLS\textsuperscript{125} (6.2 on page 734)—some information specific to thread is stored there. By the way, in win32 the fs register plays the same role, pointing to TIB\textsuperscript{126} \textsuperscript{127}.

More information can be found in the Linux kernel source code (at least in 3.11 version), in arch/x86/include/asm/stackprotector.h this variable is described in the comments.

\textbf{Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)}

Let’s get back to our simple array example (1.26.1 on page 264), again, now we can see how LLVM checks the correctness of the “canary”:

\textsuperscript{125}Thread Local Storage
\textsuperscript{126}Thread Information Block
\textsuperscript{127}wikipedia.org/wiki/Win32_Thread_Information_Block
main

var_64 = -0x64
var_60 = -0x60
var_5C = -0x5C
var_58 = -0x58
var_54 = -0x54
var_50 = -0x50
var_4C = -0x4C
var_48 = -0x48
var_44 = -0x44
var_40 = -0x40
var_3C = -0x3C
var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
var_20 = -0x20
var_1C = -0x1C
var_18 = -0x18
canary = -0x14
var_10 = -0x10

PUSH {R4-R7,LR}
ADD R7, SP, #0xC
STR.W R8, [SP,#0xC+var_10]!
SUB SP, SP, #0x54
MOVW R0, #aobjc_methtype ; "objc_methtype"
MOV R2, #0
MOVT.W R0, #0
MOV R5, #0
ADD R0, PC
LDR.W R8, [R8]
LDR.W R0, [R8]
STR R0, [SP,#0x64+canary]
MOV R0, #2
STR R2, [SP,#0x64+var_64]
STR R0, [SP,#0x64+var_60]
MOV R0, #4
STR R0, [SP,#0x64+var_5C]
MOV R0, #6
STR R0, [SP,#0x64+var_58]
MOV R0, #8
STR R0, [SP,#0x64+var_54]
MOV R0, #0xA
STR R0, [SP,#0x64+var_50]
MOV R0, #0xC
STR R0, [SP,#0x64+var_4C]
MOV R0, #0xE
STR R0, [SP,#0x64+var_48]
MOV R0, #0x10
STR R0, [SP,#0x64+var_44]
MOV R0, #0x12
STR R0, [SP,#0x64+var_40]
MOV R0, #0x14
STR R0, [SP,#0x64+var_3C]
MOV R0, #0x16
STR R0, [SP,#0x64+var_38]
MOV R0, #0x18
STR R0, [SP,#0x64+var_34]
MOV R0, #0x1A
STR R0, [SP,#0x64+var_30]
MOV R0, #0x1C
STR R0, [SP,#0x64+var_2C]
MOV R0, #0x1E
STR R0, [SP,#0x64+var_28]
MOV R0, #0x20
First of all, as we see, LLVM “unrolled” the loop and all values were written into an array one-by-one, pre-calculated, as LLVM concluded it can work faster. By the way, instructions in ARM mode may help to do this even faster, and finding this could be your homework.

At the function end we see the comparison of the “canaries”—the one in the local stack and the correct one, to which R8 points.

If they are equal to each other, a 4-instruction block is triggered by ITTTT EQ, which contains writing 0 in R0, the function epilogue and exit. If the “canaries” are not equal, the block being skipped, and the jump to ___stack_chk_fail function will occur, which, perhaps will halt execution.

### 1.26.4 One more word about arrays

Now we understand why it is impossible to write something like this in C/C++ code:

```c
void f(int size)  
{  
  int a[size];
  ...
}
```

That's just because the compiler must know the exact array size to allocate space for it in the local stack layout on at the compiling stage.

If you need an array of arbitrary size, allocate it by using malloc(), then access the allocated memory block as an array of variables of the type you need.

Or use the C99 standard feature [ISO/IEC 9899:TC3 (C C99 standard), (2007)6.7.5/2], and it works like alloca() ([1.9.2 on page 34](#)) internally.

It's also possible to use garbage collecting libraries for C.

And there are also libraries supporting smart pointers for C++.
1.26.5 Array of pointers to strings

Here is an example for an array of pointers.

```
#include <stdio.h>

const char* month1[] = {
    "January", "February", "March", "April",
    "May", "June", "July", "August",
    "September", "October", "November", "December"
};

// in 0..11 range
const char* get_month1(int month) {
    return month1[month];
}
```

x64

```
_DATA SEGMENT

month1 DQ FLAT:$SG3122
DQ FLAT:$SG3123
DQ FLAT:$SG3124
DQ FLAT:$SG3125
DQ FLAT:$SG3126
DQ FLAT:$SG3127
DQ FLAT:$SG3128
DQ FLAT:$SG3129
DQ FLAT:$SG3130
DQ FLAT:$SG3131
DQ FLAT:$SG3132
DQ FLAT:$SG3133

$SG3122 DB 'January', 00H
$SG3123 DB 'February', 00H
$SG3124 DB 'March', 00H
$SG3125 DB 'April', 00H
$SG3126 DB 'May', 00H
$SG3127 DB 'June', 00H
$SG3128 DB 'July', 00H
$SG3129 DB 'August', 00H
$SG3130 DB 'September', 00H
$SG3131 DB 'October', 00H
$SG3132 DB 'November', 00H
$SG3133 DB 'December', 00H
_DATA ENDS

months$ = 8
get_month1 PROC
    movsxd rax, ecx
    lea rcx, OFFSET FLAT:month1
    mov rax, QWORD PTR [rcx+rax*8]
    ret
get_month1 ENDP
```

The code is very simple:

- The first MOVSDX instruction copies a 32-bit value from ECX (where `month` argument is passed) to RAX with sign-extension (because the `month` argument is of type `int`).

  The reason for the sign extension is that this 32-bit value is to be used in calculations with other 64-bit values.
Hence, it has to be promoted to 64-bit\textsuperscript{128}.

- Then the address of the pointer table is loaded into RCX.

- Finally, the input value ($month$) is multiplied by 8 and added to the address. Indeed: we are in a 64-bit environment and all address (or pointers) require exactly 64 bits (or 8 bytes) for storage. Hence, each table element is 8 bytes wide. And that's why to pick a specific element, $month \times 8$ bytes has to be skipped from the start. That's what MOV does. In addition, this instruction also loads the element at this address. For 1, an element would be a pointer to a string that contains “February”, etc.

Optimizing GCC 4.9 can do the job even better\textsuperscript{129}:

\begin{verbatim}
  movsx    rdi, edi
  mov      rax, QWORD PTR month1[0+rdi*8]
  ret
\end{verbatim}

### 32-bit MSVC

Let's also compile it in the 32-bit MSVC compiler:

\begin{verbatim}
_months$ = 8
_get_month1 PROC
  mov      eax, DWORD PTR _months[$esp-4]
  mov      eax, DWORD PTR _month1[eax*4]
  ret
_get_month1 ENDP
\end{verbatim}

The input value does not need to be extended to 64-bit value, so it is used as is. And it's multiplied by 4, because the table elements are 32-bit (or 4 bytes) wide.

### 32-bit ARM

#### ARM in ARM mode

\begin{verbatim}
get_month1 PROC
  LDR     r1, |L0.100|
  LDR     r0, [r1, r0, LSL #2]
  BX      \textbar r
  ENDP

|L0.100|
  DCD    ||.data||
  DB     "January", 0
  DB     "February", 0
  DB     "March", 0
  DB     "April", 0
  DB     "May", 0
  DB     "June", 0
  DB     "July", 0
  DB     "August", 0
  DB     "September", 0
  DB     "October", 0
  DB     "November", 0
  DB     "December", 0

AREA ||.data||, DATA, ALIGN=2
\end{verbatim}

\textsuperscript{128}It is somewhat weird, but negative array index could be passed here as $month$ (negative array indices will have been explained later: \textit{3.22 on page} 593). And if this happens, the negative input value of int type is sign-extended correctly and the corresponding element before table is picked. It is not going to work correctly without sign-extension.

\textsuperscript{129}"0+" was left in the listing because GCC assembler output is not tidy enough to eliminate it. It's displacement, and it's zero here.
The address of the table is loaded in R1.

All the rest is done using just one LDR instruction.

Then input value `month` is shifted left by 2 (which is the same as multiplying by 4), then added to R1 (where the address of the table is) and then a table element is loaded from this address.

The 32-bit table element is loaded into R0 from the table.

**ARM in Thumb mode**

The code is mostly the same, but less dense, because the LSL suffix cannot be specified in the LDR instruction here:

```assembly
get_month1 PROC
    LSLS r0, r0, #2
    LDR r1, [L0, 64]
    LDR r0, [r1, r0]
    BX lr
ENDP
```

**ARM64**

Listing 1.241: Optimizing GCC 4.9 ARM64

```assembly
get_month1:
    adrp x1, .LANCHOR0
    add x1, x1, :lo12:.LANCHOR0
    ldr x0, [x1, w0, sxtw 3]
    ret

.LANCHOR0 = . + 0
.type month1, %object
.size month1, 96

month1:
    .xword .LC2
    .xword .LC3
    .xword .LC4
    .xword .LC5
    .xword .LC6
    .xword .LC7
    .xword .LC8
    .xword .LC9
    .xword .LC10
    .xword .LC11
    .xword .LC12
    .xword .LC13

.LC2:
    .string "January"

.LC3:
    .string "February"

.LC4:
    .string "March"

.LC5:
```

The address of the table is loaded in R1.

All the rest is done using just one LDR instruction.

Then input value `month` is shifted left by 2 (which is the same as multiplying by 4), then added to R1 (where the address of the table is) and then a table element is loaded from this address.

The 32-bit table element is loaded into R0 from the table.
The address of the table is loaded in X1 using ADRP/ADD pair.

Then corresponding element is picked using just one LDR, which takes W0 (the register where input argument month is), shifts it 3 bits to the left (which is the same as multiplying by 8), sign-extends it (this is what “sxtw” suffix implies) and adds to X0. Then the 64-bit value is loaded from the table into X0.

**MIPS**

*Listing 1.242: Optimizing GCC 4.4.5 (IDA)*

```
.get_month1:  
    ; load address of table into $v0:  
    la  $v0, month1  
    ; take input value and multiply it by 4:  
    sll  $a0, 2  
    ; sum up address of table and multiplied value:  
    addu  $a0, $v0  
    ; load table element at this address into $v0:  
    lw  $v0, 0($a0)  
    ; return  
    jr  $ra  
or  $at, $zero ; branch delay slot, NOP

.data #.data.rel.local
.globl month1
month1: .word aJanuary      # "January"
         .word aFebruary    # "February"
         .word aMarch      # "March"
         .word aApril      # "April"
         .word aMay       # "May"
         .word aJune      # "June"
         .word aJuly      # "July"
         .word aAugust    # "August"
         .word aSeptember # "September"
         .word aOctober   # "October"
         .word aNovember  # "November"
         .word aDecember  # "December"

.data #.rodata.str1.4
aJanuary:  .ascii "January"<0>
aFebruary: .ascii "February"<0>
aMarch:    .ascii "March"<0>
aApril:    .ascii "April"<0>
aMay:      .ascii "May"<0>
aJune:     .ascii "June"<0>
aJuly:     .ascii "July"<0>
aAugust:   .ascii "August"<0>
aSeptember: .ascii "September"<0>
aOctober:  .ascii "October"<0>
aNovember: .ascii "November"<0>
aDecember: .ascii "December"<0>
```
Array overflow

Our function accepts values in the range of 0..11, but what if 12 is passed? There is no element in table at this place.

So the function will load some value which happens to be there, and return it.

Soon after, some other function can try to get a text string from this address and may crash.

Let's compile the example in MSVC for win64 and open it in IDA to see what the linker has placed after the table:

Listing 1.243: Executable file in IDA

```plaintext
off_140011000  dq offset AJanuary_1 ; DATA XREF: .text:0000000140001003
               ; "January"
               dq offset AFebruary_1 ; "February"
               dq offset AMarch_1 ; "March"
               dq offset AApril_1 ; "April"
               dq offset AMay_1 ; "May"
               dq offset AJune_1 ; "June"
               dq offset AJuly_1 ; "July"
               dq offset AAugust_1 ; "August"
               dq offset ASeptember_1 ; "September"
               dq offset AOctober_1 ; "October"
               dq offset ANovember_1 ; "November"
               dq offset ADecember_1 ; "December"
AJanuary_1 db 'January',0 ; DATA XREF: sub 140001020+4
AFebruary_1 db 'February',0 ; DATA XREF: .data:0000000140011008
align 4
aMarch_1 db 'March',0 ; DATA XREF: .data:0000000140011010
align 4
aApril_1 db 'April',0 ; DATA XREF: .data:0000000140011018
```

Month names are came right after.

Our program is tiny, so there isn't much data to pack in the data segment, so it just the month names. But it has to be noted that there might be really anything that linker has decided to put by chance.

So what if 12 is passed to the function? The 13th element will be returned.

Let's see how the CPU treats the bytes there as a 64-bit value:

Listing 1.244: Executable file in IDA

```plaintext
off_140011000  dq offset qword_140011060 ; DATA XREF: .text:0000000140001003
               ; 0x797261756E614A
               dq offset AFebruary_1 ; "February"
               dq offset AMarch_1 ; "March"
               dq offset AApril_1 ; "April"
               dq offset AMay_1 ; "May"
               dq offset AJune_1 ; "June"
               dq offset AJuly_1 ; "July"
               dq offset AAugust_1 ; "August"
               dq offset ASeptember_1 ; "September"
               dq offset AOctober_1 ; "October"
               dq offset ANovember_1 ; "November"
               dq offset ADecember_1 ; "December"
qword_140011060 dq 797261756E614Ah ; DATA XREF: sub 140001020+4
               ; .data:off_140011000
aFebruary_1 db 'February',0 ; DATA XREF: .data:0000000140011010
align 4
aMarch_1 db 'March',0 ; DATA XREF: .data:0000000140011018
```

And this is 0x797261756E614A.

Soon after, some other function (presumably, one that processes strings) may try to read bytes at this address, expecting a C-string there.

Most likely it is about to crash, because this value doesn't look like a valid address.
Array overflow protection

If something can go wrong, it will

Murphy’s Law

It’s a bit naïve to expect that every programmer who use your function or library will never pass an argument larger than 11.

There exists the philosophy that says “fail early and fail loudly” or “fail-fast”, which teaches to report problems as early as possible and halt.

One such method in C/C++ is assertions.

We can modify our program to fail if an incorrect value is passed:

```c
const char* get_month1_checked (int month)
{
    assert (month<12);
    return month1[month];
}
```

The assertion macro checks for valid values at every function start and fails if the expression is false.

```c
$LN5:
push  rbx
sub   rsp, 32
movsx rbx, ecx
cmp   ebx, 12
jl    SHORT $LN3@get_month1
lea   rdx, OFFSET FLAT:$SG3143
lea   rcx, OFFSET FLAT:$SG3144
mov   r8d, 29
call  _wassert
$LN3@get_month1:
    lea   rcx, OFFSET FLAT:month1
    mov   rax, QWORD PTR [rcx+rbx*8]
    add   rsp, 32
    pop   rbx
    ret   0
get_month1_checked ENDP
```

In fact, assert() is not a function, but macro. It checks for a condition, then passes also the line number and file name to another function which reports this information to the user.

Here we see that both file name and condition are encoded in UTF-16. The line number is also passed (it's 29).

This mechanism is probably the same in all compilers. Here is what GCC does:

```c
.LC1:
    .string "month.c"
.LC2:
    .string "month<12"
```

```c
get_month1_checked:
    cmp   edi, 11
    jg    .L6
    movsx rdi, edi
```
So the macro in GCC also passes the function name for convenience.

Nothing is really free, and this is true for the sanitizing checks as well. They make your program slower, especially if the assert() macros used in small time-critical functions. So MSVC, for example, leaves the checks in debug builds, but in release builds they all disappear. Microsoft Windows NT kernels come in “checked” and “free” builds. The first has validation checks (hence, “checked”), the second one doesn’t (hence, “free” of checks). Of course, “checked” kernel works slower because of all these checks, so it is usually used only in debug sessions.

**Accessing specific character**

An array of pointers to strings can be accessed like this:

```c
#include <stdio.h>

const char* month[] = {
    "January", "February", "March", "April",
    "May", "June", "July", "August",
    "September", "October", "November", "December"
};

int main()
{
    // 4th month, 5th character:
    printf("%c\n", month[3][4]);
}
```

...since `month[3]` expression has a `const char*` type. And then, 5th character is taken from that expression by adding 4 bytes to its address.

By the way, arguments list passed to `main()` function has the same data type:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("3rd argument, 2nd character: %c\n", argv[3][1]);
}
```

It’s very important to understand, that, despite similar syntax, this is different from two-dimensional arrays, which we will consider later.

Another important thing to notice: strings to be addressed must be encoded in a system, where each character occupies single byte, like ASCII and extended ASCII. UTF-8 wouldn’t work here.

---


131 American Standard Code for Information Interchange
1.26.6 Multidimensional arrays

Internally, a multidimensional array is essentially the same thing as a linear array. Since the computer memory is linear, it is an one-dimensional array. For convenience, this multi-dimensional array can be easily represented as one-dimensional.

For example, this is how the elements of the 3x4 array are placed in one-dimensional array of 12 cells:

<table>
<thead>
<tr>
<th>Offset in memory</th>
<th>array element</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0][0]</td>
</tr>
<tr>
<td>1</td>
<td>[0][1]</td>
</tr>
<tr>
<td>2</td>
<td>[0][2]</td>
</tr>
<tr>
<td>3</td>
<td>[0][3]</td>
</tr>
<tr>
<td>4</td>
<td>[1][0]</td>
</tr>
<tr>
<td>5</td>
<td>[1][1]</td>
</tr>
<tr>
<td>6</td>
<td>[1][2]</td>
</tr>
<tr>
<td>7</td>
<td>[1][3]</td>
</tr>
<tr>
<td>8</td>
<td>[2][0]</td>
</tr>
<tr>
<td>9</td>
<td>[2][1]</td>
</tr>
<tr>
<td>10</td>
<td>[2][2]</td>
</tr>
<tr>
<td>11</td>
<td>[2][3]</td>
</tr>
</tbody>
</table>

Table 1.3: Two-dimensional array represented in memory as one-dimensional

Here is how each cell of 3*4 array are placed in memory:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 1.4: Memory addresses of each cell of two-dimensional array

So, in order to calculate the address of the element we need, we first multiply the first index by 4 (array width) and then add the second index. That’s called row-major order, and this method of array and matrix representation is used in at least C/C++ and Python. The term row-major order in plain English language means: “first, write the elements of the first row, then the second row ...and finally the elements of the last row”.

Another method for representation is called column-major order (the array indices are used in reverse order) and it is used at least in Fortran, MATLAB and R. column-major order term in plain English language means: “first, write the elements of the first column, then the second column ...and finally the elements of the last column”.

Which method is better?

In general, in terms of performance and cache memory, the best scheme for data organization is the one, in which the elements are accessed sequentially.

So if your function accesses data per row, row-major order is better, and vice versa.

Two-dimensional array example

We are going to work with an array of type char, which implies that each element requires only one byte in memory.

Row filling example

Let’s fill the second row with these values 0..3:

Listing 1.248: Row filling example

```c
#include <stdio.h>

char a[3][4];
```
```c
int main()
{
    int x, y;
    // clear array
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;
    // fill second row by 0..3:
    for (y=0; y<4; y++)
        a[1][y]=y;
}
```

All three rows are marked with red. We see that second row now has values 0, 1, 2 and 3:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>00C33370</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00C33380</td>
<td>02 00 00 00</td>
</tr>
<tr>
<td>00C33390</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00C333A0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>00C333B0</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

Figure 1.93: OllyDbg: array is filled

### Column filling example

Let's fill the third column with values: 0..2:

```c
#include <stdio.h>
char a[3][4];
int main()
{
    int x, y;
    // clear array
    for (x=0; x<3; x++)
        for (y=0; y<4; y++)
            a[x][y]=0;
    // fill third column by 0..2:
    for (x=0; x<3; x++)
        a[x][2]=x;
}
```

The three rows are also marked in red here.

We see that in each row, at third position these values are written: 0, 1 and 2.

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex dump</th>
</tr>
</thead>
<tbody>
<tr>
<td>01033380</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>01033390</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>010333A0</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>010333B0</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

Figure 1.94: OllyDbg: array is filled

### Access two-dimensional array as one-dimensional

We can be easily assured that it's possible to access a two-dimensional array as one-dimensional array in at least two ways:

```c
#include <stdio.h>
char a[3][4];
```
char get_by_coordinates1 (char array[3][4], int a, int b)
{
    return array[a][b];
};

char get_by_coordinates2 (char *array, int a, int b)
{
    // treat input array as one-dimensional
    // 4 is array width here
    return array[a*4+b];
};

char get_by_coordinates3 (char *array, int a, int b)
{
    // treat input array as pointer,
    // calculate address, get value at it
    // 4 is array width here
    return *(array+a*4+b);
};

int main()
{
    a[2][3]=123;
    printf ("%d\n", get_by_coordinates1(a, 2, 3));
    printf ("%d\n", get_by_coordinates2(a, 2, 3));
    printf ("%d\n", get_by_coordinates3(a, 2, 3));
};

Compile and run it: it shows correct values.

What MSVC 2013 did is fascinating, all three routines are just the same!

Listing 1.250: Optimizing MSVC 2013 x64

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates3 PROC
    ; RCX=address of array
    ; RDX=a
    ; R8=b
    movsxd rax, r8d
    ; EAX=b
    movsxd r9, edx
    ; R9=a
    add rax, rcx
    ; RAX=a+address of array
    movzx eax, BYTE PTR [rax+r9*4]
    ; AL=load byte at address RAX+R9*4=b+address of array+a*4=address of array+a*4+b
    ret 0
get_by_coordinates3 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates2 PROC
    movsxd rax, r8d
    movsxd r9, edx
    add rax, rcx
    movzx eax, BYTE PTR [rax+r9*4]
    ret 0
get_by_coordinates2 ENDP

array$ = 8
a$ = 16
b$ = 24
get_by_coordinates1 PROC
    movsxd rax, r8d

132This program is to be compiled as a C program, not C++, save it to a file with .c extension to compile it using MSVC
GCC also generates equivalent routines, but slightly different:

Listing 1.251: Optimizing GCC 4.9 x64

; RDI=address of array
; RSI=a
; RDX=b

get_by_coordinates1:
    ; sign-extend input 32-bit int values "a" and "b" to 64-bit ones
    movsx rsi, esi
    movsx rdx, edx
    lea rax, [rdi+rsi*4]
    ; RAX=RDRI+RSI=address of array+a*4
    movzx eax, BYTE PTR [rax+rdx]
    ; AL=load byte at address RAX=address of array+a*4+b
    ret

get_by_coordinates2:
    lea eax, [rdx+rsi*4]
    ; RAX=RDX+RSI*4=b+a*4
    cdqe
    movzx eax, BYTE PTR [rdi+rax]
    ; AL=load byte at address RDI=RAX=address of array+b+a*4
    ret

get_by_coordinates3:
    sal esi, 2
    ; ESI=a<<2=a*4
    ; sign-extend input 32-bit int values "a*4" and "b" to 64-bit ones
    movsx rdx, edx
    movsx rsi, esi
    add rdi, rsi
    ; RDI=RDRI=address of array+a*4
    movzx eax, BYTE PTR [rdi+rdx]
    ; AL=load byte at address RDI=address of array+a*4+b
    ret

Three-dimensional array example

It's the same for multidimensional arrays.

Now we are going to work with an array of type int: each element requires 4 bytes in memory.
Let's see:

Listing 1.252: simple example

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
    a[x][y][z]=value;
};
```

x86

We get (MSVC 2010):
Nothing special. For index calculation, three input arguments are used in the formula \( \text{address} = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z \), to represent the array as multidimensional. Do not forget that the \textit{int} type is 32-bit (4 bytes), so all coefficients must be multiplied by 4.

The GCC compiler does it differently. For one of the operations in the calculation \((30y)\), GCC produces code without multiplication instructions. This is how it done: \((y + y) \ll 4 - (y + y) = 2y \ll 4 - 2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y\). Thus, for the 30y calculation, only one addition operation, one bitwise shift operation and one subtraction operation are used. This works faster.
Non-optimizing LLVM saves all variables in local stack, which is redundant. The address of the array element is calculated by the formula we already saw.

ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb mode)

The tricks for replacing multiplication by shift, addition and subtraction which we already saw are also present here.

Here we also see a new instruction for us: RSB (Reverse Subtract).

It works just as SUB, but it swaps its operands with each other before execution. Why? SUB and RSB are instructions, to the second operand of which shift coefficient may be applied: (LSL#4).

But this coefficient can be applied only to second operand.

That’s fine for commutative operations like addition or multiplication (operands may be swapped there without changing the result).

But subtraction is a non-commutative operation, so RSB exist for these cases.
MIPS

My example is tiny, so the GCC compiler decided to put the `a` array into the 64KiB area addressable by the Global Pointer.

Listing 1.257: Optimizing GCC 4.4.5 (IDA)

```mips
insert:
; $a0=x
; $a1=y
; $a2=z
; $a3=value
  sll  $v0, $a0, 5
; $v0 = $a0<<5 = x*32
  sll  $a0, 3
; $a0 = $a0<<3 = x*8
  addu $a0, $v0
; $a0 = $a0+$v0 = x*8+x*32 = x*40
  sll  $v1, $a1, 5
; $v1 = $a1<<5 = y*32
  sll  $v0, $a0, 4
; $v0 = $a0<<4 = x*40*16 = x*640
  sll  $a1, 1
; $a1 = $a1<<1 = y*2
  subu $a1, $v1, $a1
; $a1 = $v1-$a1 = y*32-y*2 = y*30
  subu $a0, $v0, $a0
; $a0 = $v0-$a0 = x*640-x*40 = x*600
  la  $gp, __gnu_local_gp
  addu $a0, $a1, $a0
; $a0 = $a0+$a0 = y*30+x*600
  addu $a0, $a2
; $a0 = $a0+$a2 = y*30+x*600+z
; load address of table:
  lw  $v0, (a & 0xFFFF)($gp)
; multiply index by 4 to seek array element:
  sll  $a0, 2
; sum up multiplied index and table address:
  addu $a0, $v0, $a0
; store value into table and return:
  jr  $ra
  sw  $a3, 0($a0)

.comm a:0x1770
```

Getting dimensions of multidimensional array

Any string processing function, if an array of characters passed to it, can’t deduce a size of the input array. Likewise, if a function processes 2D array, only one dimension can be deduced.

For example:

```c
int get_element(int array[10][20], int x, int y)
{
    return array[x][y];
};

int main()
{
    int array[10][20];
    get_element(array, 4, 5);
};
```

... if compiled (by any compiler) and then decompiled by Hex-Rays:

```c
int get_element(int *array, int x, int y)
{
    return array[20 * x + y];
}
```
There is no way to find a size of the first dimension. If $x$ value passed is too big, buffer overflow would occur, an element from some random place of memory would be read.

And 3D array:

```c
int get_element(int array[10][20][30], int x, int y, int z)
{
    return array[x][y][z];
};

int main()
{
    int array[10][20][30];
    get_element(array, 4, 5, 6);
};
```

Hex-Rays:

```c
int get_element(int *array, int x, int y, int z)
{
    return array[600 * x + z + 30 * y];
}
```

Again, sizes of only two of 3 dimensions can be deduced.

**More examples**

The computer screen is represented as a 2D array, but the video-buffer is a linear 1D array. We talk about it here: 8.15.2 on page 891.

Another example in this book is Minesweeper game: it’s field is also two-dimensional array: 8.4 on page 794.

### 1.26.7 Pack of strings as a two-dimensional array

Let's revisit the function that returns the name of a month: listing 1.236. As you see, at least one memory load operation is needed to prepare a pointer to the string that’s the month's name.

Is it possible to get rid of this memory load operation?

In fact yes, if you represent the list of strings as a two-dimensional array:

```c
#include <stdio.h>
#include <assert.h>

const char month2[12][10]=
{
    { 'J', 'a', 'n', 'u', 'a', 'r', 'y', 0, 0, 0 },
    { 'F', 'e', 'b', 'r', 'u', 'a', 'r', 'y', 0, 0, 0 },
    { 'M', 'a', 'r', 'c', 'h', 0, 0, 0, 0, 0 },
    { 'A', 'p', 'r', 'i', 'l', 0, 0, 0, 0, 0 },
    { 'M', 'a', 'y', 0, 0, 0, 0, 0, 0, 0 },
    { 'J', 'u', 'n', 'e', 0, 0, 0, 0, 0, 0 },
    { 'J', 'u', 'l', 'y', 0, 0, 0, 0, 0, 0 },
    { 'A', 'u', 'g', 'u', 's', 't', 0, 0, 0, 0 },
    { 'S', 'e', 'p', 't', 'e', 'm', 'b', 'e', 'r', 0 },
    { 'O', 'c', 't', 'o', 'b', 'e', 'r', 0, 0, 0 },
    { 'N', 'o', 'v', 'e', 'm', 'b', 'e', 'r', 0, 0 },
    { 'D', 'e', 'c', 'e', 'm', 'b', 'e', 'r', 0, 0, 0 }
};
// in 0..11 range
const char* get_month2 (int month)
{
    return &month2[month][0];
};
```
Here is what we’ve get:

Listing 1.258: Optimizing MSVC 2013 x64

```
month2 DB 04aH
DB 061H
DB 06eH
DB 075H
DB 061H
DB 072H
DB 079H
DB 00H
DB 00H
DB 00H
...
```

get_month2 PROC
; sign-extend input argument and promote to 64-bit value
    movsx    rax, ecx
    lea      rcx, QWORD PTR [rax+rax*4]
; RCX=month+month*4=month*5
    lea      rax, OFFSET FLAT:month2
; RAX=pointer to table
    lea      rax, QWORD PTR [rax+rcx*2]
; RAX=pointer to table + RCX*2=pointer to table + month*5*2=pointer to table + month*10
    ret
get_month2 ENDP

There are no memory accesses at all.
All this function does is to calculate a point at which the first character of the name of the month is:
`pointer_to_the_table + month * 10`.

There are also two LEA instructions, which effectively work as several MUL and MOV instructions.
The width of the array is 10 bytes.
Indeed, the longest string here—“September”—is 9 bytes, and plus the terminating zero is 10 bytes.
The rest of the month names are padded by zero bytes, so they all occupy the same space (10 bytes).
Thus, our function works even faster, because all string start at an address which can be calculated easily.
Optimizing GCC 4.9 can do it even shorter:

Listing 1.259: Optimizing GCC 4.9 x64

```
movsx    rdi, edi
lea      rax, [rdi+rdi*4]
lea      rax, month2[rax+rax]
ret
```

LEA is also used here for multiplication by 10.
Non-optimizing compilers do multiplication differently.

Listing 1.260: Non-optimizing GCC 4.9 x64

```
push     rbp
mov      rbp, rsp
mov      DWORD PTR [rbp-4], edi
mov      eax, DWORD PTR [rbp-4]
movsx    rdx, eax
; RDX = sign-extended input value
    mov      rax, rdx
; RAX = month
    sal      rax, 2
; RAX = month<<2 = month*4
    add      rax, rdx
; RAX = RAX+RDX = month*4+month = month*5
    add      rax, rax
; RAX = RAX*2 = month*5*2 = month*10
    add      rax, OFFSET FLAT:month2
```
Non-optimizing MSVC just uses IMUL instruction:

Listing 1.261: Non-optimizing MSVC 2013 x64

```cpp
; RAX = month*10 + pointer to the table
pop rbp
ret

month$ = 8
get_month2 PROC
    mov DWORD PTR [rsp+8], ecx
    movsx rax, DWORD PTR month$[rsp]
    ; RAX = sign-extended input value into 64-bit one
    imul rax, rax, 10
    ; RAX = RAX*10
    lea rcx, OFFSET FLAT:month2
    ; RCX = pointer to the table
    add rcx, rax
    ; RCX = RCX*RAX = pointer to the table+month*10
    mov rax, rcx
    ; RAX = pointer to the table+month*10
    mov ecx, 1
    ; RCX = 1
    imul rcx, rcx, 0
    ; RCX = 1*0 = 0
    add rax, rcx
    ; RAX = pointer to the table+month*10 + 0 = pointer to the table+month*10
    ret 0
get_month2 ENDP

```

But one thing is weird here: why add multiplication by zero and adding zero to the final result?

This looks like a compiler code generator quirk, which wasn’t caught by the compiler’s tests (the resulting code works correctly, after all). We intentionally consider such pieces of code so the reader would understand, that sometimes one shouldn’t puzzle over such compiler artifacts.

**32-bit ARM**

Optimizing Keil for Thumb mode uses the multiplication instruction MULS:

Listing 1.262: Optimizing Keil 6/2013 (Thumb mode)

```asm
; R0 = month
MOVS r1,#0xa
; R1 = 10
MULS r0,r1,r0
; R0 = R1*R0 = 10*month
LDR r1,|L0.68|
; R1 = pointer to the table
ADDS r0,r0,r1
; R0 = R0+R1 = 10*month + pointer to the table
BX lr

```

Optimizing Keil for ARM mode uses add and shift operations:

Listing 1.263: Optimizing Keil 6/2013 (ARM mode)

```asm
; R0 = month
LDR r1,|L0.104|
; R1 = pointer to the table
ADDS r0,r0,r0,LSL #2
; R0 = R0+R0<<2 = R0+R0*4 = month*5
ADD r0,r1,r0,LSL #1
; R0 = R1+R0<<2 = pointer to the table + month*5*2 = pointer to the table + month*10
BX lr

```

ARM64
Listing 1.264: Optimizing GCC 4.9 ARM64

; W0 = month
    sxtw x0, w0
; X0 = sign-extended input value
    adrp x1, .LANCHOR1
    add x1, x1, :lo12:.LANCHOR1
; X1 = pointer to the table
    add x0, x0, x0, lsl 2
; X0 = X0+X0<<2 = X0+X0*4 = X0*5
    add x0, x1, x0, lsl 1
; X0 = X1+X0<<1 = X1+X0*2 = pointer to the table + X0*10
    ret

SXTW is used for sign-extension and promoting input 32-bit value into a 64-bit one and storing it in X0.
ADR P/ADD pair is used for loading the address of the table.
The ADD instructions also has a LSL suffix, which helps with multiplications.

MIPS

Listing 1.265: Optimizing GCC 4.4.5 (IDA)

.globl get_month2
get_month2:
    ; $a0=month
    sll $v0, $a0, 3
    ; $v0 = $a0<<3 = month*8
    sll $a0, 1
    ; $a0 = $a0<<1 = month*2
    addu $a0, $v0
    ; $a0 = month*2+month*8 = month*10
    load address of the table:
    la $v0, month2
    ; sum up table address and index we calculated and return:
    jr $ra
    addu $v0, $a0

month2:       .ascii "January"<0>
               .byte 0, 0
aFebruary:    .ascii "February"<0>
               .byte 0
aMarch:       .ascii "March"<0>
               .byte 0, 0, 0, 0
aApril:       .ascii "April"<0>
               .byte 0, 0, 0, 0
aMay:         .ascii "May"<0>
               .byte 0, 0, 0, 0, 0, 0
aJune:        .ascii "June"<0>
               .byte 0, 0, 0, 0, 0
aJuly:        .ascii "July"<0>
               .byte 0, 0, 0, 0, 0
aAugust:      .ascii "August"<0>
               .byte 0, 0, 0
aSeptember:   .ascii "September"<0>
aOctober:     .ascii "October"<0>
               .byte 0, 0
aNovember:    .ascii "November"<0>
               .byte 0
aDecember:    .ascii "December"<0>
               .byte 0, 0, 0, 0, 0, 0, 0, 0

Conclusion
This is a bit old-school technique to store text strings. You may find a lot of it in Oracle RDBMS, for example.
It’s hard to say if it’s worth doing on modern computers. Nevertheless, it is a good example of arrays, so it was added to this book.
1.26.8 Conclusion

An array is a pack of values in memory located adjacently. It’s true for any element type, including structures. Access to a specific array element is just a calculation of its address. So, a pointer to an array and address of a first element—is the same thing. This is why \texttt{ptr[0]} and \texttt{*ptr} expressions are equivalent in C/C++. It’s interesting to note that Hex-Rays often replaces the first by the second. It does so when it have no idea that it works with pointer to the whole array, and thinks that this is a pointer to single variable.

1.26.9 Exercises

- [http://challenges.re/62](http://challenges.re/62)
- [http://challenges.re/63](http://challenges.re/63)
- [http://challenges.re/64](http://challenges.re/64)
- [http://challenges.re/65](http://challenges.re/65)
- [http://challenges.re/66](http://challenges.re/66)

1.27 Example: a bug in Angband

An ancient rogue-like game from 1990’s had a nice bug in the spirit of the “Roadside Picnic” by Strugatsky brothers or “The Lost Room” TV series:

The frog-knows version was abundant of bugs. The funniest of them led to a cunning technique of cheating the game, that was called “mushroom farming”. If there were more than a certain number (about five hundred) of objects in the labyrinth, the game would break, and many old things turned into objects thrown to the floor. Accordingly, the player went into the maze, he made such longitudinal grooves there (with a special spell), and walked along the grooves, creating mushrooms with another special spell. When there were a lot of mushrooms, the player put and took, put and took some useful item, and mushrooms one by one turned into this subject. After that, the player returned with hundreds of copies of the useful item.


And some information from usenet:

```
From: be...@uswest.com (George Bell)
Subject: [Angband] Multiple artifact copies found (bug?)
Date: Fri, 23 Jul 1993 15:55:08 GMT

Up to 2000 ft I found only 4 artifacts, now my house is littered with the suckers (FYI, most I've gotten from killing nasties, like Dracoliches and the like). Something really weird is happening now, as I found multiple copies of the same artifact! My half-elf ranger is down at 2400 ft on one level which is particularly nasty. There is a graveyard plus monsters surrounded by permanent rock and 2 or 3 other special monster rooms! I did so much slashing with my favorite weapon, Crisdurian, that I filled several rooms nearly to the brim with treasure (as usual, mostly junk).

Then, when I found a way into the big vault, I noticed some of the treasure had already been identified (in fact it looked strangely familiar!). Then I found *two* Short Swords named Sting (1d6) (+7,+8), and I just ran across a third copy! I have seen multiple copies of Gurthang on this level as well. Is there some limit on the number of items per level which I have exceeded? This sounds reasonable as all multiple copies I have seen come from this level.

I'm playing PC angband. Anybody else had this problem?
```


\footnote{[https://en.wikipedia.org/wiki/The_Lost_Room](https://en.wikipedia.org/wiki/The_Lost_Room)}
Help! I need a Rod of Restore Life Levels, if there is such a thing. These Graveyards are nasty (Black Reavers and some speed 2 wraith in particular).

(https://groups.google.com/forum/#!original/rec.games.moria/jItmfrdGyL8/8csctQqA7PQJ)

From: Ceri <cm...@andrew.cmu.edu>
Subject: Re: [Angband] Multiple artifact copies found (bug?)
Date: Fri, 23 Jul 1993 23:32:20 -0400

welcome to the mush bug. if there are more than 256 items
on the floor, things start duplicating. learn to harness
this power and you will win shortly :>

--Rick

(https://groups.google.com/forum/#!search/angband$202.4$20bug$20multiplying$20items/rec.
games.moria/jItmfrdGyL8/FoQeiccewHAJ)

From: nwe...@soda.berkeley.edu (Nicholas C. Weaver)
Subject: Re: [Angband] Multiple artifact copies found (bug?)
Date: 24 Jul 1993 18:18:05 GMT

In article <74348474...@unix1.andrew.cmu.edu> Ceri <cm...@andrew.cmu.edu> writes:
> welcome to the mush bug. if there are more than 256 items
> on the floor, things start duplicating. learn to harness
> this power and you will win shortly :>
> >--Rick
>
> Question on this. Is it only the first 256 items which get
duplicated? What about the origional items? Etc Etc Etc...

    Oh, for those who like to know about bugs, though, the -n option
    (start new character) has the following behavior:

    (this is in version 2.4.Frog.knows on unix)

    If you hit controll-p, you keep your old stats.
    YOu loose all record of artifacts founds and named monsters killed.
    YOu loose all items you are carrying (they get turned into error in
    objid()s ).
    You loose your gold.
    You KEEP all the stuff in your house.

    If you kill something, and then quaff a potion of restore life
    levels, you are back up to where you were before in EXPERIENCE POINTS!!

    Gaining spells will not work right after this, unless you have a
    gain int item (for spellcasters) or gain wis item (for priests/palidans), in
    which case after performing the above, then take the item back on and off,
    you will be able to learn spells normally again.

    This can be exploited, if you are a REAL H0ZER (like me), into
    getting multiple artifacts early on. Just get to a level where you can
    pound wormtongue into the ground, kill him, go up, drop your stuff in your
    house, buy a few potions of restore exp and high value spellbooks with your
    leftover gold, angband -n yourself back to what you were before, and repeat
    the process. Yes, you CAN kill wormtongue multiple times. :)

    This also allows the creation of a human rogue with dunedain warrior
    starting stats.
Of course, such practices are evil, vile, and disgusting. I take no liability for the results of spreading this information. Yeah, it's another bug to go onto the pile.

Nicholas C. Weaver perpetual ensign guppy nwe...@soda.berkeley.edu
It is a tale, told by an idiot, full of sound and fury, signifying nothing.
Since C evolved out of B, and a C+ is close to a B, does that mean that C++ is a devolution of the language?

(https://groups.google.com/forum/#!original/rec.games.moria/jItmfrdGyL8/FoQeiccewHAJ)
The whole thread: https://groups.google.com/forum/#!search/angband$202.4$20bug$20multiplying$20items/rec.games.moria/jItmfrdGyL8/FoQeiccewHAJ.
The author of these lines has found the version with the bug (2.4 fk)\textsuperscript{136}, and we can clearly see how the global arrays are declared:

```c
/* Number of dungeon objects */
#define MAX_DUNGEON_OBJ 423
...
int16 sorted_objects[MAX_DUNGEON_OBJ];

/* Identified objects flags */
int8u object_ident[OBJECT_IDENT_SIZE];
int16 t_level[MAX_OBJ_LEVEL+1];
inven_type t_list[MAX_TALLOC];
inven_type inventory[MAX_ARRAY_SIZE];
```

Perhaps this is a reason. The\texttt{MAX_DUNGEON\_OBJ} constant is too small. Perhaps, authors should use linked lists or other data structures, which are unlimited by size. But arrays are simpler to use.

Another example of buffer overflow over globally defined arrays: 3.31 on page 637.

1.28 Manipulating specific bit(s)

A lot of functions define their input arguments as flags in bit fields.
Of course, they could be substituted by a set of\texttt{bool}-typed variables, but it is not frugally.

1.28.1 Specific bit checking

x86

Win32 API example:

```c
HANDLE fh;

fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

We get (MSVC 2010):

```c
Listing 1.266: MSVC 2010
push 0
push 128 ; 00000080H
push 4
push 0
push 1
push -1073741824 ; c0000000H
push OFFSET $SG78813
call DWORD PTR __imp__CreateFileA@28
mov DWORD PTR _fh$[ebp], eax
```

\textsuperscript{136}http://rephial.org/release/2.4.fk, https://yurichev.com/mirrors/angband-2.4.fk.tar
Let's take a look in WinNT.h:

```c
#define GENERIC_READ  (0x80000000L)
#define GENERIC_WRITE (0x40000000L)
#define GENERIC_EXECUTE (0x20000000L)
#define GENERIC_ALL  (0x10000000L)
```

Everything is clear, GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000, and that value is used as the second argument for the CreateFile() function.

How would CreateFile() check these flags?

If we look in KERNEL32.DLL in Windows XP SP3 x86, we'll find this fragment of code in CreateFileW:

```c
.C83D429 test byte ptr [ebp+dwDesiredAccess+3], 40h
.C83D42D mov [ebp+var_8], 1
.C83D434 jz short loc_7C83D417
.C83D436 jmp \loc_7C810817
```

Here we see the TEST instruction, however it doesn't take the whole second argument,
but only the most significant byte (ebp+dwDesiredAccess+3) and checks it for flag 0x40 (which implies the GENERIC_WRITE flag here).

TEST is basically the same instruction as AND, but without saving the result (recall the fact CMP is merely the same as SUB, but without saving the result (1.12.4 on page 86)).

The logic of this code fragment is as follows:

```c
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

If AND instruction leaves this bit, the ZF flag is to be cleared and the JZ conditional jump is not to be triggered. The conditional jump is triggered only if the 0x40000000 bit is absent in dwDesiredAccess variable —then the result of AND is 0, ZF is to be set and the conditional jump is to be triggered.

Let's try GCC 4.4.1 and Linux:

```c
#include <stdio.h>
#include <fcntl.h>

void main()
{
    int handle;

    handle=open ("file", O_RDWR | O_CREAT);
}
```

We get:

```c
.C83D429 test byte ptr [ebp+dwDesiredAccess+3], 40h
.C83D42D mov [ebp+var_8], 1
.C83D434 jz short loc_7C83D417
.C83D436 jmp \loc_7C810817
```

[304]

---

If we take a look in the open() function in the libc.so.6 library, it is only a syscall:

Listing 1.270: open() (libc.so.6)

So, the bit fields for open() are apparently checked somewhere in the Linux kernel.

Of course, it is easy to download both Glibc and the Linux kernel source code, but we are interested in understanding the matter without it.

So, as of Linux 2.6, when the sys_open syscall is called, control eventually passes to do_sys_open, and from there—to the do_filp_open() function (it's located in the kernel source tree in fs/namei.c).

N.B. Aside from passing arguments via the stack, there is also a method of passing some of them via registers. This is also called fastcall (6.1.3 on page 726). This works faster since CPU does not need to access the stack in memory to read argument values. GCC has the option regparm\(^{138}\), through which it's possible to set the number of arguments that can be passed via registers.

The Linux 2.6 kernel is compiled with -mregparm=3 option\(^{139}\)\(^{140}\).

What this means to us is that the first 3 arguments are to be passed via registers EAX, EDX and ECX, and the rest via the stack. Of course, if the number of arguments is less than 3, only part of registers set is to be used.

So, let's download Linux Kernel 2.6.31, compile it in Ubuntu: make vmlinux, open it in IDA, and find the do_filp_open() function. At the beginning, we see (the comments are mine):

Listing 1.271: do_filp_open() (linux kernel 2.6.31)

GCC saves the values of the first 3 arguments in the local stack. If that wasn't done, the compiler would not touch these registers, and that would be too tight environment for the compiler's register allocator.

Let's find this fragment of code:

Listing 1.272: do_filp_open() (linux kernel 2.6.31)

\(^{138}\)ohse.de/uwe/articles/gcc-attributes.html#func-regparm

\(^{139}\)kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f

\(^{140}\)See also arch/x86/include/asm/calling.h file in kernel tree

305
0x40—is what the O_CREAT macro equals to. open_flag gets checked for the presence of the 0x40 bit, and if this bit is 1, the next JNZ instruction is triggered.

ARM

The O_CREAT bit is checked differently in Linux kernel 3.8.0.

Listing 1.273: linux kernel 3.8.0

```c
struct file *do_filp_open(int dfd, struct filename *pathname,
                          const struct open_flags *op)
{
    ...  
    filp = path_openat(dfd, pathname, &nd, op, flags | LOOKUP_RCU);
    ...  
}
static struct file *path_openat(int dfd, struct filename *pathname,
                                struct nameidata *nd, const struct open_flags *op, int flags)
{
    ...  
    error = do_last(nd, &path, file, op, &opened, pathname);
    ...  
}
static int do_last(struct nameidata *nd, struct path *path,
                   struct file *file, const struct open_flags *op, int *opened, struct filename *name)
{
    ...  
    if (!(open_flag & O_CREAT)) {
        ...   
        error = lookup_fast(nd, path, &inode);
        ...  
    } else {
        ...  
        error = complete_walk(nd);
    }
    ...  
}
```

Here is how the kernel compiled for ARM mode looks in IDA:

Listing 1.274: do_last() from vmlinux (IDA)

```assembly
...  
.text:C0169EA8 MOV R9, R3 ; R3 - (4th argument) open_flag
...  
.text:C0169ED4 LDR R6, [R9] ; R6 - open_flag
...  
.text:C0169F68 TST R6, #0x40 ; jumpable C0169F00 default case
.text:C0169F6C BNE loc_C016A128
.text:C0169F70 LDR R2, [R4,#0x10]
.text:C0169F74 ADD R12, R4, #8
.text:C0169F78 LDR R3, [R4,#0xC]
.text:C0169F7C MOV R0, R4
.text:C0169F80 STR R12, [R11,#var_50]
.text:C0169F84 LDRB R3, [R2,R3]
.text:C0169F88 MOV R2, R8
.text:C0169F8C CMP R3, #0
.text:C0169F90 ORRNE R1, R1, #3
.text:C0169F94 STRNE R1, [R4,#0x24]
.text:C0169F98 ANDS R3, R6, #0x200000
.text:C0169F9C MOV R1, R12
.text:C0169FA0 LDRNE R3, [R4,#0x24]
.text:C0169FA4 ANDNE R3, R3, #1
.text:C0169FA8 EORNE R3, R3, #1
.text:C0169FAC STR R3, [R11,#var_54]
.text:C0169FB0 SUB R3, R11, #var_38
.text:C0169FB4 BL lookup_fast
```
TST is analogous to the TEST instruction in x86. We can “spot” visually this code fragment by the fact the `lookup_fast()` is to be executed in one case and `complete_walk()` in the other. This corresponds to the source code of the `do_last()` function. The `O_CREAT` macro equals to `0x40` here too.

### 1.28.2 Setting and clearing specific bits

For example:

```c
#include <stdio.h>

#define IS_SET(flag, bit) ((flag) & (bit))
#define SET_BIT(var, bit) ((var) |= (bit))
#define REMOVE_BIT(var, bit) ((var) &= ~(bit))

int f(int a)
{
    int rt=a;
    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);
    return rt;
}

int main()
{
    f(0x12340678);
}
```

**x86**

**Non-optimizing MSVC**

We get (MSVC 2010):

```assembly
Listing 1.275: MSVC 2010

        _rt$ = -4 ; size = 4
        _a$ = 8  ; size = 4
_f PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR _rt$[ebp], eax
    mov     ecx, DWORD PTR _rt$[ebp]
    or      ecx, 16384 ; 00004000H
    mov     DWORD PTR _rt$[ebp], ecx
    mov     edx, DWORD PTR _rt$[ebp]
    and     edx, -513 ; fffffdffH
    mov     DWORD PTR _rt$[ebp], edx
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f ENDP
```

The `OR` instruction sets one bit into a register while ignoring other 1 bits.

AND resets one bit. It can be said that AND just copies all bits except one. Indeed, in the second AND operand only the bits that need to be saved are set, just the one do not want to copy is not (which is 0 in the bitmask). It is the easier way to memorize the logic.
Let's try this example in OllyDbg.

First, let's see the binary form of the constants we are going to use:
0x200 (0b00000000000000000000000000000001) (i.e., the 10th bit (counting from 1st)).

Inverted 0x200 is 0xFFFFFDFF (0b11111111111111111111111111111111).

0x400 (0b00000000000000000000000000000010) (i.e., the 15th bit).

The input value is: 0x12340678 (0b10010001101000000011001111000).

Figure 1.95: OllyDbg: value is loaded into ECX
OR got executed:

Figure 1.96: OllyDbg: OR executed

15th bit is set: 0x1234678 (0b10010001101000110001100111000).
The value is reloaded again (because the compiler is not in optimizing mode):

Figure 1.97: OllyDbg: value has been reloaded into EDX
AND got executed:

Figure 1.98: OllyDbg: AND executed

The 10th bit has been cleared (or, in other words, all bits were left except the 10th) and the final value now is 0x12344 (0b100100011010001001).

Optimizing MSVC

If we compile it in MSVC with optimization turned on (/Ox), the code is even shorter:

Listing 1.276: Optimizing MSVC

```assembly
_a$ = 8 ; size = 4
_f PROC
    mov eax, DWORD PTR _a$[esp-4]
    and eax, -513 ; ffffffffh
    or eax, 16384 ; 00004000h
    ret 0
_f ENDP
```

Non-optimizing GCC

Let's try GCC 4.4.1 without optimization:

Listing 1.277: Non-optimizing GCC

```assembly
  public_f
  proc near
  var_4 = dword ptr -4
  arg_0 = dword ptr 8
  push ebp
  mov ebp, esp
  sub esp, 10h
  mov eax, [ebp+arg_0]
  mov [ebp+var_4], eax
  or [ebp+var_4], 4000h
  and [ebp+var_4], 0ffffffh
  mov eax, [ebp+var_4]
  leave
```
There is a redundant code present, however, it is shorter than the MSVC version without optimization. Now let’s try GCC with optimization turned on -O3:

**Optimizing GCC**

Listing 1.278: Optimizing GCC

```assembly
public f
f proc near
arg_0 = dword ptr 8
    push ebp
    mov ebp, esp
    mov eax, [ebp+arg_0]
    pop ebp
    or ah, 40h
    and ah, 0FDh
    retn
f endp
```

That’s shorter. It is worth noting the compiler works with the EAX register part via the AH register—that is the EAX register part from the 8th to the 15th bits included.

<table>
<thead>
<tr>
<th>Byte number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>RAX</td>
</tr>
<tr>
<td>AX</td>
</tr>
</tbody>
</table>

N.B. The 16-bit CPU 8086 accumulator was named AX and consisted of two 8-bit halves—AL (lower byte) and AH (higher byte). In 80386 almost all registers were extended to 32-bit, the accumulator was named EAX, but for the sake of compatibility, its older parts may be still accessed as AX/AH/AL.

Since all x86 CPUs are successors of the 16-bit 8086 CPU, these older 16-bit opcodes are shorter than the newer 32-bit ones. That’s why the `or ah, 40h` instruction occupies only 3 bytes. It would be more logical way to emit here `or eax, 04000h` but that is 5 bytes, or even 6 (in case the register in the first operand is not EAX).

**Optimizing GCC and regparm**

It would be even shorter if to turn on the -O3 optimization flag and also set regparm=3.

Listing 1.279: Optimizing GCC

```assembly
public f
f proc near
push ebp
or ah, 40h
mov ebp, esp
and ah, 0FDh
pop ebp
retn
f endp
```

Indeed, the first argument is already loaded in EAX, so it is possible to work with it in-place. It is worth noting that both the function prologue (push ebp / mov ebp,esp) and epilogue (pop ebp) can easily be omitted here, but GCC probably is not good enough to do such code size optimizations. However, such short functions are better to be *inlined functions* (3.14 on page 506).
BIC (Bitwise bit Clear) is an instruction for clearing specific bits. This is just like the AND instruction, but with inverted operand. I.e., it's analogous to a NOT + AND instruction pair.

ORR is “logical or”, analogous to OR in x86.

So far it’s easy.

Seems like Keil decided that the code in Thumb mode, making 0x200 from 0x4000, is more compact than the code for writing 0x200 to an arbitrary register.

So that is why, with the help of ASRS (arithmetic shift right), this value is calculated as 0x4000 >> 5.

The code that was generated by LLVM, in source code form could be something like this:

```
REMOVE_BIT (rt, 0x4200);
SET_BIT (rt, 0x4000);
```

And it does exactly what we need. But why 0x4200? Perhaps that an artifact from LLVM’s optimizer.

Probably a compiler’s optimizer error, but the generated code works correctly anyway.

You can read more about compiler anomalies here (11.4 on page 981).

Optimizing Xcode 4.6.3 (LLVM) for Thumb mode generates the same code.

Let’s rework the example slightly:

```
int f(int a)
{
    int rt=a;
    REMOVE_BIT (rt, 0x1234);
    return rt;
}
```

Then the optimizing Keil 5.03 in ARM mode does:

`141It was LLVM build 2410.2.00 bundled with Apple Xcode 4.6.3`
There are two BIC instructions, i.e., bits 0x1234 are cleared in two passes. This is because it’s not possible to encode 0x1234 in a BIC instruction, but it’s possible to encode 0x1000 and 0x234.

**ARM64: Optimizing GCC (Linaro) 4.9**

Optimizing GCC compiling for ARM64 can use the AND instruction instead of BIC:

```
Listing 1.283: Optimizing GCC (Linaro) 4.9
f:
  and  w0, w0, -513 ; 0xFFFFFFFFFDFF
  orr  w0, w0, 16384 ; 0x4000
  ret
```

**ARM64: Non-optimizing GCC (Linaro) 4.9**

Non-optimizing GCC generates more redundant code, but works just like optimized:

```
Listing 1.284: Non-optimizing GCC (Linaro) 4.9
f:
  sub  sp, sp, #32
  str  w0, [sp,12]
  ldr  w0, [sp,12]
  str  w0, [sp,28]
  ldr  w0, [sp,28]
  orr  w0, w0, 16384 ; 0x4000
  str  w0, [sp,28]
  ldr  w0, [sp,28]
  and  w0, w0, -513 ; 0xFFFFFFFFFDFF
  str  w0, [sp,28]
  ldr  w0, [sp,28]
  add  sp, sp, 32
  ret
```

**MIPS**

```
Listing 1.285: Optimizing GCC 4.4.5 (IDA)
f:
  ; $a0=a
  ori  $a0, 0x4000
  ; $a0=a|0x4000
  li   $v0, 0xFFFFFFFFDFF
  jr   $ra
       and  $v0, $a0, $v0
  ; at finish: $v0 = $a0 & $v0 = a|0x4000 & 0xFFFFFFFFDFF
```

ORI is, of course, the OR operation. “I” in the instruction name means that the value is embedded in the machine code.

But after that we have AND. There is no way to use ANDI because it’s not possible to embed the 0xFFFFFFFFDFF number in a single instruction, so the compiler has to load 0xFFFFFFFFDFF into register $V0 first and then generates AND which takes all its values from registers.
1.28.3 Shifts

Bit shifts in C/C++ are implemented using « and » operators. The x86 ISA has the SHL (SHift Left) and SHR (SHift Right) instructions for this. Shift instructions are often used in division and multiplications by powers of two: 2^n (e.g., 1, 2, 4, 8, etc.): 1.24.1 on page 211, 1.24.2 on page 216.

Shifting operations are also so important because they are often used for specific bit isolation or for constructing a value of several scattered bits.

1.28.4 Setting and clearing specific bits: FPU example

Here is how bits are located in the float type in IEEE 754 form:

```
    | S | exponent | mantissa or fraction |
    +---------------+----------+---------------------+
    | S — sign      | exponent | mantissa or fraction|
```

The sign of number is in the MSB\(^{142}\). Will it be possible to change the sign of a floating point number without any FPU instructions?

```
#include <stdio.h>

float my_abs (float i)
{
    unsigned int tmp = (*(unsigned int*)&i) & 0x7FFFFFFF;
    return *(float*)&tmp;
}

float set_sign (float i)
{
    unsigned int tmp = (*(unsigned int*)&i) | 0x80000000;
    return *(float*)&tmp;
}

float negate (float i)
{
    unsigned int tmp = (*(unsigned int*)&i) ^ 0x80000000;
    return *(float*)&tmp;
}

int main()
{
    printf ("my_abs():
");
    printf ("%f", my_abs (123.456));
    printf ("%f", my_abs (-456.123));
    printf ("set_sign():
");
    printf ("%f", set_sign (123.456));
    printf ("%f", set_sign (-456.123));
    printf ("negate():
");
    printf ("%f", negate (123.456));
    printf ("%f", negate (-456.123));
}
```

We need this trickery in C/C++ to copy to/from float value without actual conversion. So there are three functions: my_abs() resets MSB; set_sign() sets MSB and negate() flips it.

XOR can be used to flip a bit: 2.6 on page 456.

x86

The code is pretty straightforward:

```
  _tmp$ = 8
  _i$ = 8
  _my_abs PROC
```

\(^{142}\)Most Significant Bit
An input value of type float is taken from the stack, but treated as an integer value.

AND and OR reset and set the desired bit. XOR flips it.

Finally, the modified value is loaded into ST0, because floating-point numbers are returned in this register.

Now let’s try optimizing MSVC 2012 for x64:

```assembly
my_abs PROC
movss DWORD PTR [rsp+8], xmm0
mov eax, DWORD PTR i$[rsp]
btr eax, 31
mov DWORD PTR tmp$[rsp], eax
movss xmm0, DWORD PTR tmp$[rsp]
ret

my_abs ENDP

set_sign PROC
movss DWORD PTR [rsp+8], xmm0
mov eax, DWORD PTR i$[rsp]
bts eax, 31
mov DWORD PTR tmp$[rsp], eax
movss xmm0, DWORD PTR tmp$[rsp]
ret

set_sign ENDP

negate PROC
movss DWORD PTR [rsp+8], xmm0
mov eax, DWORD PTR i$[rsp]
btc eax, 31
mov DWORD PTR tmp$[rsp], eax
movss xmm0, DWORD PTR tmp$[rsp]
ret

negate ENDP
```

The input value is passed in XMM0, then it is copied into the local stack and then we see some instructions that are new to us: BTR, BTS, BTC.

These instructions are used for resetting (BTR), setting (BTS) and inverting (or complementing: BTC) specific bits. The 31st bit is MSB, counting from 0.
Finally, the result is copied into XMM0, because floating point values are returned through XMM0 in Win64 environment.

**MIPS**

GCC 4.4.5 for MIPS does mostly the same:

```plaintext
my_abs:
; move from coprocessor 1:
   mfcl $v1, $f12
   li $v0, 0x7FFFFFFF
; $v0=0x7FFFFFFF
; do AND:
   and $v0, $v1
; move to coprocessor 1:
   mtcl $v0, $f0
; return
   jr $ra
or $at, $zero ; branch delay slot

set_sign:
; move from coprocessor 1:
   mfcl $v0, $f12
   lui $v1, 0x80000000
; $v1=0x80000000
; do OR:
   or $v0, $v1, $v0
; move to coprocessor 1:
   mtcl $v0, $f0
; return
   jr $ra
or $at, $zero ; branch delay slot

negate:
; move from coprocessor 1:
   mfcl $v0, $f12
   lui $v1, 0x80000000
; $v1=0x80000000
; do XOR:
   xor $v0, $v1, $v0
; move to coprocessor 1:
   mtcl $v0, $f0
; return
   jr $ra
or $at, $zero ; branch delay slot
```

One single LUI instruction is used to load 0x80000000 into a register, because LUI is clearing the low 16 bits and these are zeros in the constant, so one LUI without subsequent ORI is enough.

**ARM**

**Optimizing Keil 6/2013 (ARM mode)**

```plaintext
my_abs PROC
 ; Clear bit:
     BIC r0,r0,#0x80000000
     BX lr
     ENDP

set_sign PROC
 ; do OR:
     ORR r0,r0,#0x80000000
     BX lr
     ENDP
```

317
negate PROC
; do XOR:
   EOR r0, r0, #0x80000000
   BX lr
ENDP

So far so good.

ARM has the BIC instruction, which explicitly clears specific bit(s). EOR is the ARM instruction name for XOR ("Exclusive OR").

**Optimizing Keil 6/2013 (Thumb mode)**

```assembly
my_abs PROC
   LSLS r0, r0, #1
   ; r0=i<<1
   LSRS r0, r0, #1
   ; r0=(i<<1)>>1
   BX lr
ENDP

set_sign PROC
   MOV r1, #1
   ; r1=1
   LSLS r1, r1, #31
   ; r1=1<<(31-0x80000000)
   ORRS r0, r0, r1
   ; r0=r0 | 0x80000000
   BX lr
ENDP
	negate PROC
   MOV r1, #1
   ; r1=1
   LSLS r1, r1, #31
   ; r1=1<<(31-0x80000000)
   EORS r0, r0, r1
   ; r0=r0 ^ 0x80000000
   BX lr
ENDP
```

Thumb mode in ARM offers 16-bit instructions and not much data can be encoded in them, so here a MOV/S/LSL instruction pair is used for forming the 0x80000000 constant. It works like this: 1 << 31 = 0x80000000.

The code of my_abs is weird and it effectively works like this expression: \((i << 1) >> 1\). This statement looks meaningless. But nevertheless, when input << 1 is executed, the MSB (sign bit) is just dropped. When the subsequent result >> 1 statement is executed, all bits are now in their own places, but MSB is zero, because all “new” bits appearing from the shift operations are always zeros. That is how the LSL/LSR instruction pair clears MSB.

**Optimizing GCC 4.6.3 (Raspberry Pi, ARM mode)**

```assembly
my_abs
; copy from S0 to R2:
   FMRS R2, S0
; clear bit:
   BIC R3, R2, #0x80000000
; copy from R3 to S0:
   FMSR S0, R3
   BX LR
```

318
Let's run Raspberry Pi Linux in QEMU and it emulates an ARM FPU, so S-registers are used here for floating point numbers instead of R-registers.

The FMRS instruction copies data from GPR to the FPU and back.

my_abs() and set_sign() looks as expected, but negate()? Why is there ADD instead of XOR?

It's hard to believe, but the instruction ADD register, 0x80000000 works just like XOR register, 0x80000000. First of all, what's our goal? The goal is to flip the MSB, so let's forget about the XOR operation. From school-level mathematics we may recall that adding values like 1000 to other values never affects the last 3 digits. For example: 1234567 + 10000 = 1244567 (last 4 digits are never affected).

But here we operate in binary base and 0x80000000 is 0b10000000000000000000000000000000, i.e., only the highest bit is set.

Adding 0x80000000 to any value never affects the lowest 31 bits, but affects only the MSB. Adding 1 to 0 is resulting in 1.

Adding 1 to 1 is resulting in 0b10 in binary form, but the 32th bit (counting from zero) gets dropped, because our registers are 32 bit wide, so the result is 0. That's why XOR can be replaced by ADD here.

It's hard to say why GCC decided to do this, but it works correctly.

### 1.28.5 Counting bits set to 1

Here is a simple example of a function that calculates the number of bits set in the input value.

This operation is also called “population count”\(^\text{143}\).

```c
#include <stdio.h>

#define IS_SET(flag, bit) ((flag) & (bit))

int f(unsigned int a) {
    int i;
    int rt=0;
    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i)) rt++;
    return rt;
}

int main() {
    f(0x12345678); // test
}

\(^\text{143}\) modern x86 CPUs (supporting SSE4) even have a POPCNT instruction for it
```
In this loop, the iteration count value \(i\) is counting from 0 to 31, so the \(1 \ll i\) statement is counting from 1 to \(0x80000000\). Describing this operation in natural language, we would say \textit{shift 1 by n bits left}. In other words, \(1 \ll i\) statement consequently produces all possible bit positions in a 32-bit number. The freed bit at right is always cleared.

Here is a table of all possible \(1 \ll i\) for \(i = 0 \ldots 31:\)

<table>
<thead>
<tr>
<th>C/C++ expression</th>
<th>Power of two</th>
<th>Decimal form</th>
<th>Hexadecimal form</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 \ll 0)</td>
<td>(2^0)</td>
<td>1</td>
<td>0x01</td>
</tr>
<tr>
<td>(1 \ll 1)</td>
<td>(2^1)</td>
<td>2</td>
<td>0x02</td>
</tr>
<tr>
<td>(1 \ll 2)</td>
<td>(2^2)</td>
<td>4</td>
<td>0x04</td>
</tr>
<tr>
<td>(1 \ll 3)</td>
<td>(2^3)</td>
<td>8</td>
<td>0x08</td>
</tr>
<tr>
<td>(1 \ll 4)</td>
<td>(2^4)</td>
<td>16</td>
<td>0x10</td>
</tr>
<tr>
<td>(1 \ll 5)</td>
<td>(2^5)</td>
<td>32</td>
<td>0x20</td>
</tr>
<tr>
<td>(1 \ll 6)</td>
<td>(2^6)</td>
<td>64</td>
<td>0x40</td>
</tr>
<tr>
<td>(1 \ll 7)</td>
<td>(2^7)</td>
<td>128</td>
<td>0x80</td>
</tr>
<tr>
<td>(1 \ll 8)</td>
<td>(2^8)</td>
<td>256</td>
<td>0x100</td>
</tr>
<tr>
<td>(1 \ll 9)</td>
<td>(2^9)</td>
<td>512</td>
<td>0x200</td>
</tr>
<tr>
<td>(1 \ll 10)</td>
<td>(2^{10})</td>
<td>1024</td>
<td>0x400</td>
</tr>
<tr>
<td>(1 \ll 11)</td>
<td>(2^{11})</td>
<td>2048</td>
<td>0x800</td>
</tr>
<tr>
<td>(1 \ll 12)</td>
<td>(2^{12})</td>
<td>4096</td>
<td>0x1000</td>
</tr>
<tr>
<td>(1 \ll 13)</td>
<td>(2^{13})</td>
<td>8192</td>
<td>0x2000</td>
</tr>
<tr>
<td>(1 \ll 14)</td>
<td>(2^{14})</td>
<td>16384</td>
<td>0x4000</td>
</tr>
<tr>
<td>(1 \ll 15)</td>
<td>(2^{15})</td>
<td>32768</td>
<td>0x8000</td>
</tr>
<tr>
<td>(1 \ll 16)</td>
<td>(2^{16})</td>
<td>65536</td>
<td>0x10000</td>
</tr>
<tr>
<td>(1 \ll 17)</td>
<td>(2^{17})</td>
<td>131072</td>
<td>0x20000</td>
</tr>
<tr>
<td>(1 \ll 18)</td>
<td>(2^{18})</td>
<td>262144</td>
<td>0x40000</td>
</tr>
<tr>
<td>(1 \ll 19)</td>
<td>(2^{19})</td>
<td>524288</td>
<td>0x80000</td>
</tr>
<tr>
<td>(1 \ll 20)</td>
<td>(2^{20})</td>
<td>1048576</td>
<td>0x100000</td>
</tr>
<tr>
<td>(1 \ll 21)</td>
<td>(2^{21})</td>
<td>2097152</td>
<td>0x200000</td>
</tr>
<tr>
<td>(1 \ll 22)</td>
<td>(2^{22})</td>
<td>4194304</td>
<td>0x400000</td>
</tr>
<tr>
<td>(1 \ll 23)</td>
<td>(2^{23})</td>
<td>8388608</td>
<td>0x800000</td>
</tr>
<tr>
<td>(1 \ll 24)</td>
<td>(2^{24})</td>
<td>16777216</td>
<td>0x1000000</td>
</tr>
<tr>
<td>(1 \ll 25)</td>
<td>(2^{25})</td>
<td>33554432</td>
<td>0x2000000</td>
</tr>
<tr>
<td>(1 \ll 26)</td>
<td>(2^{26})</td>
<td>67108864</td>
<td>0x4000000</td>
</tr>
<tr>
<td>(1 \ll 27)</td>
<td>(2^{27})</td>
<td>134217728</td>
<td>0x80000000</td>
</tr>
<tr>
<td>(1 \ll 28)</td>
<td>(2^{28})</td>
<td>268435456</td>
<td>0x100000000</td>
</tr>
<tr>
<td>(1 \ll 29)</td>
<td>(2^{29})</td>
<td>536870912</td>
<td>0x200000000</td>
</tr>
<tr>
<td>(1 \ll 30)</td>
<td>(2^{30})</td>
<td>1073741824</td>
<td>0x400000000</td>
</tr>
<tr>
<td>(1 \ll 31)</td>
<td>(2^{31})</td>
<td>2147483648</td>
<td>0x800000000</td>
</tr>
</tbody>
</table>

These constant numbers (bit masks) very often appear in code and a practicing reverse engineer must be able to spot them quickly.

Decimal numbers below 65536 and hexadecimal ones are very easy to memorize. While decimal numbers above 65536 are, probably, not worth memorizing.

These constants are very often used for mapping flags to specific bits. For example, here is excerpt from 
ssl_private.h from Apache 2.4.6 source code:

```c
#define SSL_OPT_NONE (0)
#define SSL_OPT_RELSET (1<<0)
#define SSL_OPT_STDENVVARS (1<<1)
#define SSL_OPT_EXPORTCERTDATA (1<<3)
#define SSL_OPT_FAKEBASICAUTH (1<<4)
#define SSL_OPT_STRICTREQUIRE (1<<5)
#define SSL_OPT_OPTRENEGOTIATE (1<<6)
#define SSL_OPT_LEGACYDNFORMAT (1<<7)
```
zero, it might be even 123456, that is why it always works correctly.

**x86**

**MSVC**

Let's compile (MSVC 2010):

```assembly
Listing 1.292: MSVC 2010

_rts = -8 ; size = 4
_i$ = -4 ; size = 4
_a$ = 8 ; size = 4

_f PROC
    push ebp
    mov ebp, esp
    sub esp, 8
    mov DWORD PTR _rts[ebp], 0
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN4@f
$LN3@f:
    mov eax, DWORD PTR _i$[ebp] ; increment of i
    add eax, 1
    mov DWORD PTR _i$[ebp], eax

$LN4@f:
    cmp DWORD PTR _i$[ebp], 32 ; 00000020H
    jge SHORT $LN2@f ; loop finished?
    mov edx, cl
    shl edx, cl ; EDX=EDX<<CL
    and edx, DWORD PTR _a$[ebp]
    je SHORT $LN1@f ; result of AND instruction was 0? ; then skip next instructions
    mov eax, DWORD PTR _rts[ebp] ; no, not zero
    add eax, 1
    mov DWORD PTR _rts[ebp], eax

$LN1@f:
    jmp SHORT $LN3@f
$LN2@f:
    mov eax, DWORD PTR _rts[ebp]
    mov esp, ebp
    pop ebp
    ret 0
_f ENDP
```
Let's load this example into OllyDbg. Let the input value be 0x12345678.

For $i = 1$, we see how $i$ is loaded into ECX:

![CPU - main thread, module shifts](image)

**Figure 1.99: OllyDbg: $i = 1$, $i$ is loaded into ECX**

EDX is 1. SHL is to be executed now.
SHL has been executed:

Figure 1.100: OllyDbg: $i = 1, EDX = 1 \ll 1 = 2$

EDX contain $1 \ll 1$ (or 2). This is a bit mask.
AND sets ZF to 1, which implies that the input value (0x12345678) ANDed with 2 results in 0:

So, there is no corresponding bit in the input value.

The piece of code, which increments the counter is not to be executed: the JZ instruction bypassing it.
Let's trace a bit further and $i$ is now 4. SHL is to be executed now:

Figure 1.102: OllyDbg: $i = 4$, $i$ is loaded into ECX
EDX $= 1 \ll 4$ (or $0 \times 10$ or 16):

Figure 1.103: OllyDbg: $i = 4$, EDX $= 1 \ll 4 = 0 \times 10$

This is another bit mask.
Figure 1.104: OllyDbg: $i = 4$, is there that bit in the input value? Yes. ($ZF = 0$)

$ZF$ is 0 because this bit is present in the input value.
Indeed, $0x12345678 \& 0x10 = 0x10$.

This bit counts: the jump is not triggering and the bit counter incrementing.
The function returns 13. This is total number of bits set in $0x12345678$.

**GCC**

Let’s compile it in GCC 4.4.1:

```
Listing 1.293: GCC 4.4.1

public f
f PROC NEAR

rt = dword ptr -0Ch
i = dword ptr -8
arg_0 = dword ptr 8

push ebp
mov ebp, esp
push ebx
sub esp, 10h
mov [ebp+rt], 0
mov [ebp+i], 0
jmp short loc_80483D0

loc_80483D0:
mov eax, [ebp+i]
mov edx, 1
mov ebx, edx
mov ecx, eax
shl ebx, cl
mov eax, ebx
and eax, [ebp+arg_0]
test eax, eax
jz short loc_80483EB
add [ebp+rt], 1

loc_80483EB:
add [ebp+i], 1
```

327
```
cmp   [ebp+i], 1Fh  
jle   short loc_80483D0
mov   eax, [ebp+rt]
add   esp, 10h
pop   ebx
pop   ebp
retn
```

### x64

Let's modify the example slightly to extend it to 64-bit:

```c
#include <stdio.h> 
#include <stdint.h> 
#define IS_SET(flag, bit) ((flag) & (bit)) 

int f(uint64_t a) 
{
  uint64_t i;
  int rt=0;

  for (i=0; i<64; i++)
    if (IS_SET (a, 1ULL<<i))
      rt++;

  return rt;
}
```

**Non-optimizing GCC 4.8.2**

So far so easy.

---

```
f:
push  rbp
mov   rbp, rsp
mov   QWORD PTR [rbp-24], rdi ; a
mov   DWORD PTR [rbp-12], 0 ; rt=0
mov   DWORD PTR [rbp-8], 0 ; i=0
jmp   .L2
.L4: 
mov   rax, QWORD PTR [rbp-8]
mov   rdx, QWORD PTR [rbp-24]
; RAX = i, RDX = a
mov   ecx, eax
; ECX = i
shr   rdx, cl
; RDX = RDX>>CL = a>>i
mov   rax, rdx
; RAX = RDX = a>>i
and   eax, 1
; EAX = (EAX&1) = (a>>i)&1
test   rax, rax
; the last bit is zero?
; skip the next ADD instruction, if it was so.
je    .L3
add   DWORD PTR [rbp-12], 1 ; rt++
.L3:  
add   QWORD PTR [rbp-8], 1 ; i++
.L2: 
cmp   QWORD PTR [rbp-8], 63 ; i<63?
jbe   .L4 ; jump to the loop body begin, if so
mov   eax, DWORD PTR [rbp-12] ; return rt
pop   ebp
ret
```
Optimizing GCC 4.8.2

Listing 1.295: Optimizing GCC 4.8.2

```assembly
f:
  xor eax, eax  ; rt variable will be in EAX register
  xor ecx, ecx  ; i variable will be in ECX register
.L3:
  mov rsi, rdi  ; load input value
  lea edx, [rax+1]  ; EDX=EAX+1
  ; EDX here is a new version of rt,
  ; which will be written into rt variable, if the last bit is 1
  shr rsi, cl  ; RSI=RSI>>CL
  and esi, 1  ; ESI=ESI&1
  ; the last bit is 1? If so, write new version of rt into EAX
  cmovne eax, edx
  add rcx, 1  ; RCX++
  cmp rcx, 64
  jne .L3
rep ret  ; AKA fatret
```

This code is terser, but has a quirk.

In all examples that we see so far, we were incrementing the “rt” value after comparing a specific bit, but the code here increments “rt” before (line 6), writing the new value into register EDX. Thus, if the last bit is 1, the CMOVNE instruction (which is a synonym for CMOVNZ) commits the new value of “rt” by moving EDX (“proposed rt value”) into EAX (“current rt” to be returned at the end).

Hence, the incrementing is performed at each step of loop, i.e., 64 times, without any relation to the input value.

The advantage of this code is that it contain only one conditional jump (at the end of the loop) instead of two jumps (skipping the “rt” value increment and at the end of loop). And that might work faster on the modern CPUs with branch predictors: 2.10.1 on page 462.

The last instruction is REP RET (opcode F3 C3) which is also called FATRET by MSVC. This is somewhat optimized version of RET, which is recommended by AMD to be placed at the end of function, if RET goes right after conditional jump: [Software Optimization Guide for AMD Family 16h Processors, (2013)p.15]

Optimizing MSVC 2010

Listing 1.296: Optimizing MSVC 2010

```assembly
a$ = 8
f PROC
; RCX = input value
  xor eax, eax
  mov edx, 1
  lea r8d, QWORD PTR [rax+64]
; R8D=64
  npad 5
$LL4$:
  test rdx, rcx
  ; there are no such bit in input value?
  ; skip the next INC instruction then.
  je SHORT $LN3$f
  inc eax  ; rt++
$LN3$f:
  rol rdx, 1  ; RDX=RDX<<1
  dec r8  ; R8--
  jne SHORT $LL4$f
fatret 0
f ENDP
```

\[144\] Conditional MOVe if Not Equal
\[145\] Conditional MOVe if Not Zero
\[146\] More information on it: [http://go.yurichev.com/17328](http://go.yurichev.com/17328)
Here the ROL instruction is used instead of SHL, which is in fact “rotate left” instead of “shift left”, but in this example it works just as SHL.

You can read more about the rotate instruction here: 1.6 on page 1020.

R8 here is counting from 64 to 0. It’s just like an inverted i.

Here is a table of some registers during the execution:

<table>
<thead>
<tr>
<th>RDX</th>
<th>R8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000000000000001</td>
<td>64</td>
</tr>
<tr>
<td>0x0000000000000002</td>
<td>63</td>
</tr>
<tr>
<td>0x0000000000000004</td>
<td>62</td>
</tr>
<tr>
<td>0x0000000000000008</td>
<td>61</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x4000000000000000</td>
<td>2</td>
</tr>
<tr>
<td>0x8000000000000000</td>
<td>1</td>
</tr>
</tbody>
</table>

At the end we see the FATRET instruction, which was explained here: 1.28.5 on the previous page.

Optimizing MSVC 2012

Listing 1.297: Optimizing MSVC 2012

```
a$ = 8
f PROC
; RCX = input value
    xor   eax, eax
    mov   edx, 1
    lea   r8d, QWORD PTR [rax+32]
; EDX = 1, R8D = 32
    npad  5
$LL4@f:
    ; pass 1 -----------------------------
    test  rdx, rcx
    je    SHORT $LN3@f
    inc   eax    ; rt++
$LN3@f:
    rol   rdx, 1 ; RDX=RDX<<1
; -----------------------------
$LN11@f:
    test  rdx, rcx
    je    SHORT $LN11@f
    inc   eax    ; rt++
$LN11@f:
    rol   rdx, 1 ; RDX=RDX<<1
; -----------------------------
    dec   r8    ; R8--
    jne   SHORT $LL4@f
    fatret 0
f ENDP
```

Optimizing MSVC 2012 does almost the same job as optimizing MSVC 2010, but somehow, it generates two identical loop bodies and the loop count is now 32 instead of 64.

To be honest, it’s not possible to say why. Some optimization trick? Maybe it’s better for the loop body to be slightly longer?

Anyway, such code is relevant here to show that sometimes the compiler output may be really weird and illogical, but perfectly working.

**ARM + Optimizing Xcode 4.6.3 (LLVM) (ARM mode)**

Listing 1.298: Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
| MOV        | R1, R0 |
| MOV        | R0, #0 |
| MOV        | R2, #1 |
| MOV        | R3, R0 |
| loc_2E54   |       |
```
TST is the same thing as TEST in x86.

As was noted before (3.12.3 on page 499), there are no separate shifting instructions in ARM mode. However, there are modifiers LSL (Logical Shift Left), LSR (Logical Shift Right), ASR (Arithmetic Shift Right), ROR (Rotate Right) and RRX (Rotate Right with Extend), which may be added to such instructions as MOV, TST, CMP, ADD, SUB, RSB147.

These modifiers define how to shift the second operand and by how many bits.

Thus the “TST R1, R2,LSL R3” instruction works here as $R_1 \land (R_2 \ll R_3)$.

### ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

Almost the same, but here are two LSL.W/TST instructions are used instead of a single TST, because in Thumb mode it is not possible to define LSL modifier directly in TST.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R1, R2, LSL R3</th>
<th>R3++</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>R3, R3, #1</td>
<td></td>
</tr>
<tr>
<td>ADDNE</td>
<td>R0, R0, #1</td>
<td></td>
</tr>
<tr>
<td>CMP</td>
<td>R3, #32</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>loc_2E54</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>LR</td>
<td></td>
</tr>
</tbody>
</table>

ARM64 + Optimizing GCC 4.9

Let’s take the 64-bit example which has been already used: 1.295 on page 329.

Listing 1.299: Optimizing GCC (Linaro) 4.8

```
f:
    mov w2, 0 ; rt=0
    mov x5, 1
    mov w1, w2
.L2:
    lsl x4, x5, x1 ; w4 = w5<<w1 = 1<i
    add w3, w2, 1 ; new_rt=rt+1
    tst x4, x0 ; (1<i) & a
    add w1, w1, 1 ; i++
    ; result of TST was non-zero?
    ; then w2=w3 or rt=new_rt.
    ; otherwise: w2=w2 or rt=rt (idle operation)
    csel w2, w3, w2, ne
    cmp w1, 64 ; i<64?
    bne .L2 ; yes
    mov w0, w2 ; return rt
    ret
```

The result is very similar to what GCC generates for x64: 1.295 on page 329.

The CSEL instruction is “Conditional SELect”. It just chooses one variable of two depending on the flags set by TST and copies the value into W2, which holds the “rt” variable.

147 These instructions are also called “data processing instructions”
ARM64 + Non-optimizing GCC 4.9

And again, we'll work on the 64-bit example which was already used: 1.28.5 on page 328. The code is more verbose, as usual.

Listing 1.300: Non-optimizing GCC (Linaro) 4.8

```assembly
f:
    sub   sp, sp, #32
    str   x0, [sp,8] ; store "a" value to Register Save Area
    str   wzr, [sp,24] ; rt=0
    str   wzr, [sp,28] ; i=0
    b     .L2
.L4:
    ldr   w0, [sp,28]
    mov   x1, 1
    lsl   x0, x1, x0 ; X0 = X1<<X0 = 1<<i
    mov   x1, x0
    ; X1 = 1<<i
    ldr   x0, [sp,8]
    and   x0, x1, x0
    ; X0 = X1&X0 = (1<<i) & a
    cmp   x0, xzr
    beq   .L3
    rt++
    ldr   w0, [sp,24]
    add   w0, w0, 1
    str   w0, [sp,24]
.L3:
    ; i++
    ldr   w0, [sp,28]
    add   w0, w0, 1
    str   w0, [sp,28]
.L2:
    ; i<=63? then jump to .L4
    ldr   w0, [sp,28]
    cmp   w0, 63
    ble   .L4
    ; return rt
    ldr   w0, [sp,24]
    add   sp, sp, 32
    ret
```

MIPS
Non-optimizing GCC

Listing 1.301: Non-optimizing GCC 4.4.5 (IDA)

```assembly
f:
; IDA is not aware of variable names, we gave them manually:
rt    = -0x10
i     = -0xC
var_4 = -4
a     = 0

    addiu  $sp, -0x18
    sw     $fp, 0x18+var_4($sp)
    move   $fp, $sp
    sw     $a0, 0x18+a($fp)
; initialize rt and i variables to zero:
    sw     $zero, 0x18+rt($fp)
    sw     $zero, 0x18+i($fp)
; jump to loop check instructions:
    b     loc_68
    or     $at, $zero ; branch delay slot, NOP
```
loc_20:
  li  $v1, 1
  lw  $v0, 0x18+i($fp)
  or  $at, $zero ; load delay slot, NOP
  sllv $v0, $v1, $v0

 ; $v0 = 1<<i
  move $v1, $v0
  lw  $v0, 0x18+a($fp)
  or  $at, $zero ; load delay slot, NOP
  and  $v0, $v1, $v0

 ; $v0 = a & (1<<i)
; is a & (1<<i) equals to zero? jump to loc_58 then:
  beqz $v0, loc_58
  or  $at, $zero
; no jump occurred, that means a & (1<<i)!=0, so increment "rt" then:
  lw  $v0, 0x18+rt($fp)
  addiu $v0, 1
  sw  $v0, 0x18+rt($fp)

loc_58:
; increment i:
  lw  $v0, 0x18+i($fp)
  or  $at, $zero ; load delay slot, NOP
  addiu $v0, 1
  sw  $v0, 0x18+i($fp)

loc_68:
; load i and compare it with 0x20 (32).
; jump to loc_20 if it is less then 0x20 (32):
  lw  $v0, 0x18+i($fp)
  or  $at, $zero ; load delay slot, NOP
  slti $v0, 0x20 # '
  bnez $v0, loc_20
  or  $at, $zero ; branch delay slot, NOP

; function epilogue. return rt:
  lw  $v0, 0x18+rt($fp)
  move $sp, $fp ; load delay slot
  lw  $fp, 0x18+var_4($sp)
  addiu $sp, 0x18 ; load delay slot
  jr  $ra
  or  $at, $zero ; branch delay slot, NOP

That is verbose: all local variables are located in the local stack and reloaded each time they’re needed.
The SLLV instruction is “Shift Word Left Logical Variable”, it differs from SLL only in that the shift amount
is encoded in the SLL instruction (and is fixed, as a consequence), but SLLV takes shift amount from a
register.

**Optimizing GCC**

That is terser. There are two shift instructions instead of one. Why?
It’s possible to replace the first SLLV instruction with an unconditional branch instruction that jumps right
to the second SLLV. But this is another branching instruction in the function, and it’s always favorable to
get rid of them: 2.10.1 on page 462.

Listing 1.302: Optimizing GCC 4.4.5 (IDA)

f:
; $a0=a
; rt variable will reside in $v0:
  move $v0, $zero
; i variable will reside in $v1:
  move $v1, $zero
  li  $t0, 1
  li  $a3, 32
  sllv $a1, $t0, $v1
; $a1 = $t0<<$v1 = 1<<i
1.28.6 Conclusion

Analogous to the C/C++ shifting operators $\ll$ and $\gg$, the shift instructions in x86 are SHR/SHL (for unsigned values) and SAR/SHL (for signed values).

The shift instructions in ARM are LSR/LSL (for unsigned values) and ASR/LSL (for signed values).

It’s also possible to add shift suffix to some instructions (which are called “data processing instructions”).

**Check for specific bit (known at compile stage)**

Test if the 0b1000000 bit (0x40) is present in the register’s value:

```
Listing 1.303: C/C++

if (input&0x40) {
    ...
}
```

```
Listing 1.304: x86

TEST REG, 40h
JNZ is_set
; bit is not set
```

```
Listing 1.305: x86

TEST REG, 40h
JZ is_cleared
; bit is set
```

```
Listing 1.306: ARM (ARM mode)

TST REG, #0x40
BNE is_set
; bit is not set
```

Sometimes, AND is used instead of TEST, but the flags that are set are the same.

**Check for specific bit (specified at runtime)**

This is usually done by this C/C++ code snippet (shift value by $n$ bits right, then cut off lowest bit):

```
Listing 1.307: C/C++

if ((value>>n)&1) {
    ...
}
```

This is usually implemented in x86 code as:
Listing 1.308: x86

; REG=input_value
; CL=n
SHR REG, CL
AND REG, 1

Or (shift 1 bit \( n \) times left, isolate this bit in input value and check if it's not zero):

Listing 1.309: C/C++

```c
define n
if (value & (1<<n))
    ....
```

This is usually implemented in x86 code as:

Listing 1.310: x86

```assembly
; CL=n
MOV REG, 1
SHL REG, CL
AND input_value, REG
```

**Set specific bit (known at compile stage)**

Listing 1.311: C/C++

```c
value=value|0x40;
```

Listing 1.312: x86

```assembly
OR REG, 40h
```

Listing 1.313: ARM (ARM mode) and ARM64

```assembly
ORR R0, R0, #0x40
```

**Set specific bit (specified at runtime)**

Listing 1.314: C/C++

```c
value=value|(1<<n);
```

This is usually implemented in x86 code as:

Listing 1.315: x86

```assembly
; CL=n
MOV REG, 1
SHL REG, CL
OR input_value, REG
```

**Clear specific bit (known at compile stage)**

Just apply AND operation with the inverted value:

Listing 1.316: C/C++

```c
value=value&(-0x40);
```

Listing 1.317: x86

```assembly
AND REG, 0xFFFFFBFh
```

Listing 1.318: x64

```assembly
AND REG, 0xFFFFFFFFFBFh
```
This is actually leaving all bits set except one.

ARM in ARM mode has BIC instruction, which works like the NOT + AND instruction pair:

Listing 1.319: ARM (ARM mode)

```
BIC R0, R0, #0x40
```

Clear specific bit (specified at runtime)

Listing 1.320: C/C++

```
value = value & (~(1 << n));
```

Listing 1.321: x86

```
: CL=n
MOV REG, 1
SHL REG, CL
NOT REG
AND input_value, REG
```

1.28.7 Exercises

- [http://challenges.re/67](http://challenges.re/67)
- [http://challenges.re/68](http://challenges.re/68)
- [http://challenges.re/69](http://challenges.re/69)
- [http://challenges.re/70](http://challenges.re/70)

1.29 Linear congruential generator as pseudorandom number generator

Perhaps, the linear congruential generator is the simplest possible way to generate random numbers. It’s not in favour nowadays\(^\text{148}\), but it’s so simple (just one multiplication, one addition and AND operation), that we can use it as an example.

```c
#include <stdint.h>

// constants from the Numerical Recipes book
#define RNG_a 1664525
#define RNG_c 1013904223

static uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state = init;
}

int my_rand ()
{
    rand_state = rand_state * RNG_a;
    rand_state = rand_state + RNG_c;
    return rand_state & 0x7fff;
}
```

There are two functions: the first one is used to initialize the internal state, and the second one is called to generate pseudorandom numbers.

We see that two constants are used in the algorithm. They are taken from [William H. Press and Saul A. Teukolsky and William T. Vetterling and Brian P. Flannery, *Numerical Recipes*, (2007)].

Let’s define them using a `#define` C/C++ statement. It’s a macro.

\(^{148}\)Mersenne twister is better
The difference between a C/C++ macro and a constant is that all macros are replaced with their value by C/C++ preprocessor, and they don’t take any memory, unlike variables.

In contrast, a constant is a read-only variable.

It’s possible to take a pointer (or address) of a constant variable, but impossible to do so with a macro.

The last AND operation is needed because by C-standard my_rand() has to return a value in the 0..32767 range.

If you want to get 32-bit pseudorandom values, just omit the last AND operation.

### 1.29.1 x86

Here we see it: both constants are embedded into the code. There is no memory allocated for them.

The my_srand() function just copies its input value into the internal rand_state variable.

my_rand() takes it, calculates the next rand_state, cuts it and leaves it in the EAX register.

The non-optimized version is more verbose:

```assembly
Listing 1.323: Non-optimizing MSVC 2013

BSS SEGMENT
_rand_state DD 01H DUP (?)
BSS ENDS
_init$ = 8
_srand PROC
    mov    eax, DWORD PTR _init$[esp-4]
    mov    DWORD PTR _rand_state, eax
    ret
_srand ENDP

_TEXT SEGMENT
_rand PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _init$[ebp]
    mov    DWORD PTR _rand_state, eax
    pop    ebp
    ret 0
_rand ENDP

_TEXT ENDS
```

Here we see it: both constants are embedded into the code. There is no memory allocated for them.

The my_srand() function just copies its input value into the internal rand_state variable.

my_rand() takes it, calculates the next rand_state, cuts it and leaves it in the EAX register.

The non-optimized version is more verbose:
The x64 version is mostly the same and uses 32-bit registers instead of 64-bit ones (because we are working with int values here).

But my_srand() takes its input argument from the ECX register rather than from stack:

```assembly
  mov eax, DWORD PTR _rand_state
  and eax, 32767 ; 00007fffH
  pop ebp
  ret 0

/rand ENDP

.TEXT ENDS
```

1.29.3 32-bit ARM

GCC compiler generates mostly the same code.
It’s not possible to embed 32-bit constants into ARM instructions, so Keil has to place them externally and load them additionally. One interesting thing is that it’s not possible to embed the 0x7FFF constant as well. So what Keil does is shifting rand_state left by 17 bits and then shifting it right by 17 bits. This is analogous to the \((\text{rand\_state} \ll 17) \gg 17\) statement in C/C++. It seems to be useless operation, but what it does is clearing the high 17 bits, leaving the low 15 bits intact, and that’s our goal after all.

Optimizing Keil for Thumb mode generates mostly the same code.

### 1.29.4 MIPS

```
my_srand:
    ; store $a0 to rand_state:
    lui $v0, (rand_state >> 16)
    jr $ra
    sw $a0, rand_state

my_rand:
    ; load rand_state to $v0:
    lui $v1, (rand_state >> 16)
    lw $v0, rand_state
    or $at, $zero ; load delay slot
    ; multiplicate rand_state in $v0 by 1664525 (RNG_a):
    sll $a1, $v0, 2
    sll $a0, $v0, 4
    addu $a0, $a1, $a0
    sll $a1, $a0, 6
    subu $a0, $a1, $a0
    addu $a0, $v0
    sll $a1, $a0, 5
    addu $a0, $a1
    sll $a0, 3
    addu $v0, $a0, $v0
    sll $a0, $v0, 2
    addu $v0, $a0

    ; add 1013904223 (RNG_c)
    ; the LI instruction is coalesced by IDA from LUI and ORI
    li $a0, 0x3C6EF35F
    addu $v0, $a0

    ; store to rand_state:
    sw $v0, (rand_state & 0xFFFF)($v1)
    jr $ra
    andi $v0, 0x7FFF ; branch delay slot
```

Wow, here we see only one constant (0x3C6EF35F or 1013904223). Where is the other one (1664525)? It seems that multiplication by 1664525 is performed by just using shifts and additions! Let’s check this assumption:

```
#define RNG_a 1664525

int f (int a)
{
    return a*RNG_a;
}
```

```
f:
    sll $v1, $a0, 2
    sll $v0, $a0, 4
```
addu $v0, $v1, $v0
sll $v1, $v0, 6
subu $v0, $v1, $v0
addu $v0, $a0
sll $v1, $v0, 5
addu $v0, $v1
sll $v0, 3
addu $a0, $v0, $a0
sll $v0, $a0, 2
jr $ra
addu $v0, $a0, $v0 ; branch delay slot

Indeed!

**MIPS relocations**

We will also focus on how such operations as load from memory and store to memory actually work.

The listings here are produced by IDA, which hides some details.

We’ll run objdump twice: to get a disassembled listing and also relocations list:

Listing 1.328: Optimizing GCC 4.4.5 (objdump)

```bash
# objdump -D rand_O3.o
...
00000000 <my_srand>:
  0:  3c020000 lui v0,0x0
  4:  03e00008 jr ra
  8: ac440000 sw a0,0(v0)
 12:  3c030000 lui v1,0x0
 16:  8c620000 lw v0,0(v1)
 20:  00200825 move at,at
 24:  00022880 sll a1,v0,0x2
 28:  00022100 sll a0,v0,0x4
 32:  00a42021 addu a0,a1,a0
 36:  00042980 sll a1,a0,0x6
 40:  00a42023 subu a0,a1,a0
 44:  00822021 addu a0,a0,v0
 48:  00042940 sll a1,a0,0x5
 52:  00852021 addu a0,a0,a1
 56:  00042940 sll a0,a0,0x3
 60:  00821021 addu v0,a0,v0
 64:  00220800 sll a0,v0,0x2
 68:  00441021 addu v0,v0,a0
 72:  3c043c6e lui a0,0x3c6e
 76:  3484f35f ori a0,a0,0xf35f
 80:  00441021 addu v0,v0,a0
 84:  ac620000 sw v0,0(v1)
 88:  03e00008 jr ra
 92:  30427fff andi v0,v0,0x7fff

# objdump -r rand_O3.o
...

RELOCATION RECORDS FOR [.text]:
OFFSET TYPE VALUE
00000000 R_MIPS_HI16 .bss
00000008 R_MIPS_LO16 .bss
000000c R_MIPS_HI16 .bss
0000010 R_MIPS_LO16 .bss
0000054 R_MIPS_LO16 .bss
...

340
Let's consider the two relocations for the `my_srand()` function.
The first one, for address 0 has a type of `R_MIPS_HI16` and the second one for address 8 has a type of `R_MIPS_LO16`.
That implies that address of the beginning of the .bss segment is to be written into the instructions at address of 0 (high part of address) and 8 (low part of address).
The `rand_state` variable is at the very start of the .bss segment.
So we see zeros in the operands of instructions LUI and SW, because nothing is there yet— the compiler don't know what to write there.
The linker will fix this, and the high part of the address will be written into the operand of LUI and the low part of the address—to the operand of SW.
SW will sum up the low part of the address and what is in register $V0 (the high part is there).
It's the same story with the `my_rand()` function: `R_MIPS_HI16` relocation instructs the linker to write the high part of the .bss segment address into instruction LUI.
So the high part of the `rand_state` variable address is residing in register $V1.
The LW instruction at address 0x10 sums up the high and low parts and loads the value of the `rand_state` variable into $V0.
The SW instruction at address 0x54 do the summing again and then stores the new value to the `rand_state` global variable.
IDA processes relocations while loading, thus hiding these details, but we should keep them in mind.

1.29.5   Thread-safe version of the example

The thread-safe version of the example is to be demonstrated later: 6.2.1 on page 734.

1.30   Structures

A C/C++ structure, with some assumptions, is just a set of variables, always stored in memory together, not necessary of the same type.\(^{149}\)

1.30.1   MSVC: SYSTEMTIME example

Let's take the `SYSTEMTIME`\(^{150}\) win32 structure that describes time.
This is how it's defined:

\begin{verbatim}
Listing 1.329: WinBase.h

typedef struct __SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
\end{verbatim}

Let's write a C function to get the current time:

\begin{verbatim}
#include <windows.h>
#include <stdio.h>

void main()
{

149AKA “heterogeneous container”
150MSDN: SYSTEMTIME structure

341
SYSTEMTIME t;
GetSystemTime (&t);
printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
    t.wYear, t.wMonth, t.wDay,
    t.wHour, t.wMinute, t.wSecond);
return;
};

We get (MSVC 2010):

Listing 1.330: MSVC 2010 /GS-

_t$ = -16 : size = 16
_main PROC
  push ebp
  mov ebp, esp
  sub esp, 16
  lea eax, DWORD PTR _t$[ebp]
push eax
call DWORD PTR __imp__GetSystemTime@4
  movzx ecx, WORD PTR _t$[ebp+12] ; wSecond
  push ecx
  movzx edx, WORD PTR _t$[ebp+10] ; wMinute
  push edx
  movzx eax, WORD PTR _t$[ebp+8] ; wHour
  push eax
  movzx ecx, WORD PTR _t$[ebp+6] ; wDay
  push ecx
  movzx edx, WORD PTR _t$[ebp+2] ; wMonth
  push edx
  movzx eax, WORD PTR _t$[ebp] ; wYear
  push eax
  push OFFSET $SG78811 ; '%%04d-%%02d-%%02d %%02d:%%02d:', 0AH, 00H
  call _printf
  add esp, 28
  xor eax, eax
  mov esp, ebp
  pop ebp
  ret 0
_main ENDP

16 bytes are allocated for this structure in the local stack —that is exactly sizeof(WORD)*8 (there are 8 WORD variables in the structure).

Pay attention to the fact that the structure begins with the wYear field. It can be said that a pointer to the SYSTEMTIME structure is passed to the GetSystemTime() function, but it is also can be said that a pointer to the wYear field is passed, and that is the same! GetSystemTime() writes the current year to the WORD pointer pointing to, then shifts 2 bytes ahead, writes current month, etc., etc.

151 MSDN: SYSTEMTIME structure
OllyDbg

Let's compile this example in MSVC 2010 with /GS- /MD keys and run it in OllyDbg.

Let's open windows for data and stack at the address which is passed as the first argument of the GetSystemTime() function, and let's wait until it's executed. We see this:

![OllyDbg screenshot](image)

**Figure 1.105: OllyDbg: GetSystemTime() just executed**

The system time of the function execution on my computer is 9 December 2014, 22:29:52:

**Listing 1.331: printf() output**

```
2014-12-09 22:29:52
```

So we see these 16 bytes in the data window:

```
DE 07 0C 00 02 00 09 00 16 00 1D 00 34 00 04 03
```

Each two bytes represent one field of the structure. Since the endianness is little endian, we see the low byte first and then the high one.

Hence, these are the values currently stored in memory:

<table>
<thead>
<tr>
<th>Hexadecimal number</th>
<th>decimal number</th>
<th>field name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07DE</td>
<td>2014</td>
<td>wYear</td>
</tr>
<tr>
<td>0x000C</td>
<td>12</td>
<td>wMonth</td>
</tr>
<tr>
<td>0x0002</td>
<td>2</td>
<td>wDayOfWeek</td>
</tr>
<tr>
<td>0x0009</td>
<td>9</td>
<td>wDay</td>
</tr>
<tr>
<td>0x0016</td>
<td>22</td>
<td>wHour</td>
</tr>
<tr>
<td>0x001D</td>
<td>29</td>
<td>wMinute</td>
</tr>
<tr>
<td>0x0034</td>
<td>52</td>
<td>wSecond</td>
</tr>
<tr>
<td>0x03D4</td>
<td>980</td>
<td>wMilliseconds</td>
</tr>
</tbody>
</table>

The same values are seen in the stack window, but they are grouped as 32-bit values.

And then printf() just takes the values it needs and outputs them to the console.

Some values aren't output by printf() (wDayOfWeek and wMilliseconds), but they are in memory right now, available for use.

**Replacing the structure with array**

The fact that the structure fields are just variables located side-by-side, can be easily demonstrated by doing the following. Keeping in mind the SYSTEMTIME structure description, it's possible to rewrite this simple example like this:
include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf("%04d-%02d-%02d %02d:%02d:%02d\n",
            array[0] /* wYear */, array[1] /* wMonth */, array[2] /* wDay */,

    return;
};

The compiler grumbles a bit:

systemtime2.c(7) : warning C4133: 'function' : incompatible types - from 'WORD [8]' to 'LPSYSTEMTIME'

But nevertheless, it produces this code:

Listing 1.332: Non-optimizing MSVC 2010

And it works just as the same!

It is very interesting that the result in assembly form cannot be distinguished from the result of the previous compilation.

So by looking at this code, one cannot say for sure if there was a structure declared, or an array.

Nevertheless, no sane person would do it, as it is not convenient.

Also the structure fields may be changed by developers, swapped, etc.

We will not study this example in OllyDbg, because it will be just the same as in the case with the structure.

1.30.2 Let’s allocate space for a structure using malloc()

Sometimes it is simpler to place structures not the in local stack, but in the heap:
```c
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;
    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));
    GetSystemTime (t);
    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            t->wYear, t->wMonth, t->wDay,
            t->wHour, t->wMinute, t->wSecond);
    free (t);
    return;
}
```

Let's compile it now with optimization (/Ox) so it would be easy to see what we need.

Listing 1.333: Optimizing MSVC

```assembly
_main PROC
  push esi
  push 16
  call malloc
  mov esi, eax
  call DWORD PTR _imp_GetSystemTime@4
  movzx eax, WORD PTR [esi+12]; wSecond
  movzx ecx, WORD PTR [esi+10]; wMinute
  movzx edx, WORD PTR [esi+8]; wHour
  push eax
  movzx eax, WORD PTR [esi+6]; wDay
  push ecx
  movzx ecx, WORD PTR [esi+2]; wMonth
  push edx
  movzx edx, WORD PTR [esi]; wYear
  push eax
  push ecx
  push edx
  push OFFSET $SG78833
  call printf
  push esi
  call _free
  add esp, 32
  xor eax, eax
  pop esi
  ret 0

_main ENDP
```

So, sizeof(SYSTEMTIME) = 16 and that is exact number of bytes to be allocated by malloc(). It returns a pointer to a freshly allocated memory block in the EAX register, which is then moved into the ESI register. GetSystemTime() win32 function takes care of saving value in ESI, and that is why it is not saved here and continues to be used after the GetSystemTime() call.

New instruction —MOVZX (Move with Zero eXtend). It may be used in most cases as MOVZX, but it sets the remaining bits to 0. That's because printf() requires a 32-bit int, but we got a WORD in the structure — that is 16-bit unsigned type. That's why by copying the value from a WORD into int, bits from 16 to 31 must be cleared, because a random noise may be there, which is left from the previous operations on the register(s).

In this example, it's possible to represent the structure as an array of 8 WORDs:
void main()
{
    WORD *t;
    t=(WORD *)malloc (16);
    GetSystemTime (t);
    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
    free (t);
    return;
};
unix_time=time(NULL);

localtime_r(&unix_time, &t);

printf("Year: %d\n", t.tm_year+1900);
printf("Month: %d\n", t.tm_mon);
printf("Day: %d\n", t.tm_mday);
printf("Hour: %d\n", t.tm_hour);
printf("Minutes: %d\n", t.tm_min);
printf("Seconds: %d\n", t.tm_sec);
}

Let's compile it in GCC 4.4.1:

Listing 1.335: GCC 4.4.1

main proc near
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 40h
mov dword ptr [esp], 0 ; first argument for time()
call time
mov [esp+3Ch], eax
lea eax, [esp+3Ch] ; take pointer to what time() returned
lea edx, [esp+10h] ; at ESP+10h struct tm will begin
mov [esp+4], edx ; pass pointer to the structure begin
mov [esp], eax ; pass pointer to result of time()
call localtime_r
mov eax, [esp+24h] ; tm_year
lea edx, [eax+76Ch] ; edx=eax+1900
mov eax, offset format ; "Year: %d"
mov [esp+4], edx
mov [esp], eax
call printf
mov edx, [esp+20h] ; tm_mon
mov eax, offset aMonthD ; "Month: %d"
mov [esp+4], edx
mov [esp], eax
call printf
mov edx, [esp+1Ch] ; tm_mday
mov eax, offset aDayD ; "Day: %d"
mov [esp+4], edx
mov [esp], eax
call printf
mov edx, [esp+18h] ; tm_hour
mov eax, offset aHourD ; "Hour: %d"
mov [esp+4], edx
mov [esp], eax
call printf
mov edx, [esp+14h] ; tm_min
mov eax, offset aMinutesD ; "Minutes: %d"
mov [esp+4], edx
mov [esp], eax
call printf
mov edx, [esp+10h]
mov eax, offset aSecondsD ; "Seconds: %d"
mov [esp+4], edx ; tm_sec
mov [esp], eax
call printf
leave
retn
main endp

Somehow, IDA did not write the local variables’ names in the local stack. But since we already are experienced reverse engineers :-) we may do it without this information in this simple example.

Please also pay attention to the lea edx, [eax+76Ch] —this instruction just adds 0x76C (1900) to value in EAX, but doesn’t modify any flags. See also the relevant section about LEA (.1.6 on page 1014).
Let’s try to load the example into GDB:

```
Listing 1.336: GDB

let's@ubuntuvm:~$ gnu gdb (gdb) ...
reading symbols from /home/dennis/polygon/GCC___tm... (no debugging symbols found)... done.
(gdb) b printf
breakpoint 1 at 0x8048330
(gdb) run
starting program: /home/dennis/polygon/GCC___tm
breakpoint 1, ___printf (format=0x80485c0 "Year: %d\n") at printf.c:29
29 printf.c: No such file or directory.
(gdb)
```

We can easily find our structure in the stack. First, let’s see how it’s defined in `time.h`:

```
Listing 1.337: time.h

```struct tm {
  int tm_sec;
  int tm_min;
  int tm_hour;
  int tm_mday;
  int tm_mon;
  int tm_year;
  int tm_wday;
  int tm_yday;
  int tm_isdst;
};```

Pay attention that 32-bit `int` is used here instead of `WORD` in `SYSTEMTIME`. So, each field occupies 32-bit.

Here are the fields of our structure in the stack:

```
0xbffff0dc: 0x080484c3 0x080485c0 0x0000007de 0x00000000
0xbffff0ec: 0x08048301 0x538c93ed 0x00000025 0x0000000a
0xbffff0fc: 0x00000012 0x00000002 0x00000005 0x00000072
0xbffff10c: 0x00000010 0x00000001 0x00000072 0x000002a30
0xbffff11c: 0x00000001 0x00000098 0x00000001 0x00002a30
(gdb)
```

Or as a table:

<table>
<thead>
<tr>
<th>Hexadecimal number</th>
<th>decimal number</th>
<th>field name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000000025</td>
<td>37</td>
<td>tm_sec</td>
</tr>
<tr>
<td>0x00000000a</td>
<td>10</td>
<td>tm_min</td>
</tr>
<tr>
<td>0x000000012</td>
<td>18</td>
<td>tm_hour</td>
</tr>
<tr>
<td>0x000000002</td>
<td>2</td>
<td>tm_mday</td>
</tr>
<tr>
<td>0x000000005</td>
<td>5</td>
<td>tm_mon</td>
</tr>
<tr>
<td>0x000000072</td>
<td>114</td>
<td>tm_year</td>
</tr>
<tr>
<td>0x000000001</td>
<td>1</td>
<td>tm_wday</td>
</tr>
<tr>
<td>0x000000098</td>
<td>152</td>
<td>tm_yday</td>
</tr>
<tr>
<td>0x000000000</td>
<td>1</td>
<td>tm_isdst</td>
</tr>
</tbody>
</table>

The date result is slightly corrected for demonstration purposes. Of course, it’s not possible to run GDB that quickly, in the same second.
Just like SYSTEMTIME (1.30.1 on page 341),
there are also other fields available that are not used, like tm_wday, tm_yday, tm_isdst.

ARM

Optimizing Keil 6/2013 (Thumb mode)

Same example:

<table>
<thead>
<tr>
<th>Listing 1.338: Optimizing Keil 6/2013 (Thumb mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_38 = -0x38</td>
</tr>
<tr>
<td>var_34 = -0x34</td>
</tr>
<tr>
<td>var_30 = -0x30</td>
</tr>
<tr>
<td>var_2C = -0x2C</td>
</tr>
<tr>
<td>var_28 = -0x28</td>
</tr>
<tr>
<td>var_24 = -0x24</td>
</tr>
<tr>
<td>timer = -0xC</td>
</tr>
</tbody>
</table>

PUSH {LR}
MOV R0, #0 ; timer
SUB SP, SP, #0x34
BL time
STR R0, [SP,#0x38+timer]
MOV R1, SP ; tp
ADD R0, SP, #0x38+timer ; timer
BL localtime_r
LDR R1, =0x76C
LDR R0, [SP,#0x38+var_24]
ADD6 R1, R0, R1
ADR R0, aYearD ; "Year: %d"
BL __2printf
LDR R1, [SP,#0x38+var_28]
ADR R0, aMonthD ; "Month: %d"
BL __2printf
LDR R1, [SP,#0x38+var_2C]
ADR R0, aDayD ; "Day: %d"
BL __2printf
LDR R1, [SP,#0x38+var_30]
ADR R0, aHourD ; "Hour: %d"
BL __2printf
LDR R1, [SP,#0x38+var_34]
ADR R0, aMinutesD ; "Minutes: %d"
BL __2printf
LDR R1, [SP,#0x38+var_38]
ADR R0, aSecondsD ; "Seconds: %d"
BL __2printf
ADD SP, SP, #0x34
POP {PC}

Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

IDA “knows” the tm structure (because IDA “knows” the types of the arguments of library functions like localtime_r()),
so it shows here structure elements accesses and their names.

<table>
<thead>
<tr>
<th>Listing 1.339: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_38 = -0x38</td>
</tr>
<tr>
<td>var_34 = -0x34</td>
</tr>
</tbody>
</table>

PUSH {R7,LR}
MOV R7, SP
SUB SP, SP, #0x30
MOVS R0, #0 ; time_t *
BLX _time
ADD R1, SP, #0x38+var_34 ; struct tm *
MIPS

Listing 1.340: Optimizing GCC 4.4.5 (IDA)

main:
;

; IDA is not aware of structure field names, we named them manually:

var_40 = -0x40
var_38 = -0x38
seconds = -0x34
minutes = -0x30
hour = -0x2C
day = -0x28
month = -0x24
year = -0x20
var_4 = -4

lui $gp, (__gnu_local_gp >> 16)
addiu $sp, -0x50
la $gp, (__gnu_local_gp & 0xFFFF)
sw $ra, 0x50+var_4($sp)
This is an example where the branch delay slots can confuse us.

For example, there is the instruction addiu $a1, 1900 at line 35 which adds 1900 to the year number.
It's executed before the corresponding JALR at line 34, do not forget about it.

**Structure as a set of values**

In order to illustrate that the structure is just variables laying side-by-side in one place, let's rework our example while looking at the *tm* structure definition again: listing.1.337.
#include <stdio.h>
#include <time.h>

void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday, tm_yday, tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);
    localtime_r (&unix_time, &tm_sec);
    printf("Year: %d\n", tm_year+1900);
    printf("Month: %d\n", tm_mon);
    printf("Day: %d\n", tm_mday);
    printf("Hour: %d\n", tm_hour);
    printf("Minutes: %d\n", tm_min);
    printf("Seconds: %d\n", tm_sec);
}

N.B. The pointer to the tm_sec field is passed into localtime_r, i.e., to the first element of the “structure”. The compiler warns us:

Listing 1.341: GCC 4.7.3

GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' from incompatible pointer type
GCC_tm2.c:11:5: note: expected 'struct tm *' but argument is of type 'int *'

But nevertheless, it generates this:

Listing 1.342: GCC 4.7.3

main proc near
var_30 = dword ptr -30h
var_2C = dword ptr -2Ch
unix_time = dword ptr -1Ch
tm_sec = dword ptr -0Ch
tm_min = dword ptr -14h
tm_hour = dword ptr -10h
tm_mday = dword ptr -0Ch
tm_mon = dword ptr -8
tm_year = dword ptr -4

push ebp
mov ebp, esp
and esp, 0FFFFFF0h
sub esp, 30h
call __main
mov [esp+30h+var_30], 0 ; arg 0
call time
mov [esp+30h+unix_time], eax
lea eax, [esp+30h+tm_sec]
mov [esp+30h+var_2C], eax
lea eax, [esp+30h+tm_year]
mov [esp+30h+tm_year], eax
add eax, 1900
mov [esp+30h+var_2C], eax
mov [esp+30h+var_30], offset aYearD ; "Year: %d\n"
call printf
mov eax, [esp+30h+tm_mon]
mov [esp+30h+var_2C], eax
mov [esp+30h+var_30], offset aMonthD ; "Month: %d\n"
call printf
mov eax, [esp+30h+tm_mday]
mov [esp+30h+var_2C], eax
call printf
mov eax, [esp+30h+tm_hour]
mov [esp+30h+var_2C], eax
mov [esp+30h+var_30], offset aDayD ; "Day: %d\n"
call printf
mov eax, [esp+30h+tm_min]
mov [esp+30h+var_2C], eax
mov [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
call printf
mov eax, [esp+30h+tm_sec]
mov [esp+30h+var_2C], eax
mov [esp+30h+var_30], offset aMinutesD ; "Minutes: %d\n"
call printf
mov eax, [esp+30h+tm_nsec]
mov [esp+30h+var_2C], eax
mov [esp+30h+var_30], offset aSecondsD ; "Seconds: %d\n"
call printf
leave
retn
main endp

This code is identical to what we saw previously and it is not possible to say, was it a structure in original source code or just a pack of variables.

And this works. However, it is not recommended to do this in practice.

Usually, non-optimizing compilers allocates variables in the local stack in the same order as they were declared in the function.

Nevertheless, there is no guarantee.

By the way, some other compiler may warn about the tm_year, tm_mon, tm_mday, tm_hour, tm_min variables, but not tm_sec are used without being initialized.

Indeed, the compiler is not aware that these are to be filled by localtime_r() function.

We chose this example, since all structure fields are of type int.

This would not work if structure fields are 16-bit (WORD), like in the case of the SYSTEMTIME structure—GetSystemTime() will fill them incorrectly (because the local variables are aligned on a 32-bit boundary). Read more about it in next section: “Fields packing in structure” (1.30.4 on page 356).

So, a structure is just a pack of variables laying in one place, side-by-side. We could say that the structure is the instruction to the compiler, directing it to hold variables in one place. By the way, in some very early C versions (before 1972), there were no structures at all [Dennis M. Ritchie, The development of the C language, (1993)]^{153}. There is no debugger example here: it is just the same as you already saw.

**Structure as an array of 32-bit words**

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i;
    unix_time=time(NULL);
    localtime_r (&unix_time, &t);
    for (i=0; i<9; i++)
    {
        int tmp=((int *)&t)[i];
        printf ("0x%08X (%d)\n", tmp, tmp);
    }
}
```

^{153}Also available as [http://go.yurichev.com/17264](http://go.yurichev.com/17264)
We just cast a pointer to structure to an array of int’s. And that works! We run the example at 23:51:45 26-July-2014.

The variables here are in the same order as they are enumerated in the definition of the structure: \textbf{1.337 on page 348}.

Here is how it gets compiled:

\textbf{Listing 1.343: Optimizing GCC 4.8.1}

\begin{lstlisting}[language=C] main proc near
push ebp
mov ebp, esp
push esi
push ebx
and esi, 0FFFFFFFFh
sub esp, 40h
mov dword ptr [esp], 0 ; timer
lea ebx, [esp+14h]
call _time
lea esi, [esp+38h]
mov [esp+4], ebx ; tp
mov [esp+10h], eax
lea eax, [esp+10h]
mov [esp], eax ; timer
call _localtime_r
nop
lea esi, [esi+0] ; NOP
loc_80483D8:
; EBX here is pointer to structure, ESI is the pointer to the end of it.
mov eax, [ebx] ; get 32-bit word from array
add ebx, 4 ; next field in structure
mov dword ptr [esp+4], 0x08xD ; "0x%08X (%d)\n"
mov dword ptr [esp], 1
mov [esp+0Ch], eax ; pass value to printf()
mov [esp+8], eax ; pass value to printf()
call ___printf_chk
cmp ebx, esi ; meet structure end?
jnz short loc_80483D8 ; no - load next value then
lea esp, [ebp-8]
pop ebp
pop esi
pop ebx
retn
main endp
\end{lstlisting}

Indeed: the space in the local stack is first treated as a structure, and then it’s treated as an array.

It’s even possible to modify the fields of the structure through this pointer.

And again, it’s dubiously hackish way to do things, not recommended for use in production code.

\textbf{Exercise}

As an exercise, try to modify (increase by 1) the current month number, treating the structure as an array.

\textbf{Structure as an array of bytes}

We can go even further. Let’s \textit{cast} the pointer to an array of bytes and dump it:
```c
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;
    int i, j;

    unix_time=time(NULL);
    localtime_r(&unix_time, &t);

    for (i=0; i<9; i++)
    {
        for (j=0; j<4; j++)
            printf("0x%02X ", ((unsigned char*)&t)[i*4+j]);
        printf("\n");
    }
}
```

We also run this example at 23:51:45 26-July-2014. The values are just the same as in the previous dump (1.30.3 on the preceding page), and of course, the lowest byte goes first, because this is a little-endian architecture (2.8 on page 460).

Listing 1.344: Optimizing GCC 4.8.1

```
main proc near
push ebp
mov ebp, esp
push edi
push esi
push ebx
and esp, 0FFFFFFFF0h
sub esp, 40h
mov dword ptr [esp], 0 ; timer
lea esi, [esp+14h]
call __time
lea edi, [esp+38h] ; struct end
mov [esp+4], esi ; tp
mov [esp+10h], eax
lea eax, [esp+10h]
mov [esp], eax ; timer
call __localtime_r
lea esi, [esi+0] ; NOP
; EDI here is the pointer to structure in local stack. EDI is the pointer to structure end.
loc_8048408:
xor ebx, ebx ; j=0

loc_804840A:
movzx eax, byte ptr [esi+ebx] ; load byte
add ebx, 1 ; j=j+1
mov dword ptr [esp+4], offset a0x02x ; "0x%02X ",
mov dword ptr [esp+8], eax ; pass loaded byte to printf()
call ___printf_chk
cmp ebx, 4
```

The time and date are the same for demonstration purposes. Byte values are fixed up.
GNU Scientific Library: Representation of complex numbers

This is a relatively rare case when an array is used instead of a structure, on purpose:

Representation of complex numbers

Complex numbers are represented using the type `gsl_complex`. The internal representation of this type may vary across platforms and should not be accessed directly. The functions and macros described below allow complex numbers to be manipulated in a portable way.

For reference, the default form of the `gsl_complex` type is given by the following struct:

```c
typedef struct
{
    double dat[2];
} gsl_complex;
```

The real and imaginary part are stored in contiguous elements of a two element array. This eliminates any padding between the real and imaginary parts, `dat[0]` and `dat[1]`, allowing the struct to be mapped correctly onto packed complex arrays.


1.30.4 Fields packing in structure

One important thing is fields packing in structures.

Let's take a simple example:

```c
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s)
{
    printf("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};

int main()
{
    struct s tmp;
    tmp.a=1;
    tmp.b=2;
}```
As we see, we have two \texttt{char} fields (each is exactly one byte) and two more \texttt{—int} (each \texttt{—} 4 bytes).

\textbf{x86}

This compiles to:

```
Listing 1.345: MSVC 2012 /GS- /Ob0

1  _tmp$ = -16
2  _main PROC
3    push ebp
4    mov ebp, esp
5    sub esp, 16
6    mov BYTE PTR _tmp$[ebp], 1 ; set field \texttt{a}
7    mov DWORD PTR _tmp$[ebp+4], 2 ; set field \texttt{b}
8    mov BYTE PTR _tmp$[ebp+8], 3 ; set field \texttt{c}
9    mov DWORD PTR _tmp$[ebp+12], 4 ; set field \texttt{d}
10   sub esp, 16 ; allocate place for temporary structure
11   mov eax, esp
12   mov ecx, DWORD PTR _tmp$[ebp] ; copy our structure to the temporary one
13   mov DWORD PTR [eax], ecx
14   mov edx, DWORD PTR _tmp$[ebp+4]
15   mov DWORD PTR [eax+4], edx
16   mov ecx, DWORD PTR _tmp$[ebp+8]
17   mov DWORD PTR [eax+8], ecx
18   mov edx, DWORD PTR _tmp$[ebp+12]
19   mov DWORD PTR [eax+12], edx
20   call \texttt{f}
21   add esp, 16
22   xor eax, eax
23   mov esp, ebp
24   pop ebp
25   ret 0

26  _main ENDP

27  \texttt{s$} = 8 ; size = 16
28  ?f@YAXUs@@Z PROC ; \texttt{f}
29    push ebp
30    mov ebp, esp
31    mov eax, DWORD PTR _s$[ebp+12]
32    push eax
33    movsx ecx, BYTE PTR _s$[ebp+8]
34    push ecx
35    mov edx, DWORD PTR _s$[ebp+4]
36    push edx
37    movsx eax, BYTE PTR _s$[ebp]
38    push eax
39    push OFFSET $SG3842
40    call \texttt{printf}
41    add esp, 20
42    pop ebp
43    ret 0
44  \texttt{f@YAXUs@@Z ENDP ; \texttt{f}}
45  \_TEXT ENDS
```

We pass the structure as a whole, but in fact, as we can see, the structure is being copied to a temporary one (a place in stack is allocated in line 10 for it, and then all 4 fields, one by one, are copied in lines 12 \ldots 19), and then its pointer (address) is to be passed.

The structure is copied because it's not known whether the \texttt{f()} function going to modify the structure or not. If it gets changed, then the structure in \texttt{main()} has to remain as it has been.

We could use C/C++ pointers, and the resulting code will be almost the same, but without the copying.
As we can see, each field’s address is aligned on a 4-byte boundary. That’s why each `char` occupies 4 bytes here (like `int`). Why? Because it is easier for the CPU to access memory at aligned addresses and to cache data from it.

However, it is not very economical.

Let’s try to compile it with option `/Zp1` (`/Zp[n] pack structures on n-byte boundary`).

Now the structure takes only 10 bytes and each `char` value takes 1 byte. What does it give to us? Size economy. And as drawback —the CPU accessing these fields slower than it could.

The structure is also copied in `main()` not field-by-field, but directly 10 bytes, using three pairs of MOV. Why not 4?

The compiler decided that it’s better to copy 10 bytes using 3 MOV pairs than to copy two 32-bit words and two bytes using 4 MOV pairs.

By the way, such copy implementation using MOV instead of calling the `memcpy()` function is widely used, because it’s faster than a call to `memcpy()`—for short blocks, of course: 3.14.1 on page 511.

As it can be easily guessed, if the structure is used in many source and object files, all these must be compiled with the same convention about structures packing.

Aside from MSVC `/Zp` option which sets how to align each structure field, there is also the `#pragma pack`
compiler option, which can be defined right in the source code. It is available in both MSVC\textsuperscript{155} and GCC\textsuperscript{156}.

Let’s get back to the \texttt{SYSTEMTIME} structure that consists of 16-bit fields. How does our compiler know to pack them on 1-byte alignment boundary?

\texttt{WinNT.h} file has this:

\textbf{Listing 1.347: WinNT.h}

```c
#include "pshpack1.h"
```

And this:

\textbf{Listing 1.348: WinNT.h}

```c
#include "pshpack4.h" // 4 byte packing is the default
```

The file \texttt{PshPack1.h} looks like:

\textbf{Listing 1.349: PshPack1.h}

```c
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86) ) || defined(_PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif
```

This tell the compiler how to pack the structures defined after \texttt{#pragma pack}.

\textsuperscript{155}MSDN: Working with Packing Structures
\textsuperscript{156}Structure-Packing Pragmas
OllyDbg + fields are packed by default

Let’s try our example (where the fields are aligned by default (4 bytes)) in OllyDbg:

![OllyDbg: Before printf() execution](image)

We see our 4 fields in the data window.

But where do the random bytes (0x30, 0x37, 0x01) come from, that are next to the first (a) and third (c) fields?

By looking at our listing 1.345 on page 357, we can see that the first and third fields are `char`, therefore only one byte is written, 1 and 3 respectively (lines 6 and 8).

The remaining 3 bytes of the 32-bit words are not being modified in memory! Hence, random garbage is left there.

This garbage doesn’t influence the printf() output in any way, because the values for it are prepared using the MOVVSX instruction, which takes bytes, not words: listing.1.345 (lines 34 and 38).

By the way, the MOVVSX (sign-extending) instruction is used here, because `char` is signed by default in MSVC and GCC. If the unsigned `char` data type or `uint8_t` was used here, MOVZX instruction would have been used instead.
OllyDbg + fields aligning on 1 byte boundary

Things are much clearer here: 4 fields occupy 10 bytes and the values are stored side-by-side

Figure 1.107: OllyDbg: Before printf() execution

ARM

Optimizing Keil 6/2013 (Thumb mode)

Listing 1.350: Optimizing Keil 6/2013 (Thumb mode)

As we may recall, here a structure is passed instead of pointer to one, and since the first 4 function arguments in ARM are passed via registers, the structure’s fields are passed via R0-R3.

LDRB loads one byte from memory and extends it to 32-bit, taking its sign into account. This is similar to MOVSX in x86. Here it is used to load fields \(a\) and \(c\) from the structure.
One more thing we spot easily is that instead of function epilogue, there is jump to another function’s epilogue! Indeed, that was quite different function, not related in any way to ours, however, it has exactly the same epilogue (probably because, it hold 5 local variables too \((5 \times 4 = 0x14)\)).

Also it is located nearby (take a look at the addresses).

Indeed, it doesn’t matter which epilogue gets executed, if it works just as we need.

Apparently, Keil decides to reuse a part of another function to economize.

The epilogue takes 4 bytes while jump—only 2.

**ARM + Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)**

![Listing 1.351: Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)](image)

SXTB (*Signed Extend Byte*) is analogous to MOVSX in x86. All the rest—just the same.

**MIPS**

![Listing 1.352: Optimizing GCC 4.4.5 (IDA)](image)
Structure fields come in registers $A0..A3$ and then get reshuffled into $A1..A3$ for printf(), while 4th field (from $A3$) is passed via local stack using SW.

But there are two SRA ("Shift Word Right Arithmetic") instructions, which prepare char fields. Why?

MIPS is a big-endian architecture by default 2.8 on page 460, and the Debian Linux we work in is big-endian as well.

So when byte variables are stored in 32-bit structure slots, they occupy the high 31..24 bits.

And when a char variable needs to be extended into a 32-bit value, it must be shifted right by 24 bits.

char is a signed type, so an arithmetical shift is used here instead of logical.

One more word

Passing a structure as a function argument (instead of a passing pointer to structure) is the same as passing all structure fields one by one.

If the structure fields are packed by default, the f() function can be rewritten as:

```c
void f(char a, int b, char c, int d)
{
    printf("a=%d; b=%d; c=%d; d=%d\n", a, b, c, d);
}
```

And that leads to the same code.

1.30.5 Nested structures

Now what about situations when one structure is defined inside of another?

```c
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};

void f(struct outer_struct s)
{
    printf("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
        s.a, s.b, s.c.a, s.c.b, s.d, s.e);
}

int main()
{
```
struct outer_struct s;
s.a=1;
s.b=2;
s.c.a=100;
s.c.b=101;
s.d=3;
s.e=4;
f(s);

...in this case, both inner_struct fields are to be placed between the a,b and d,e fields of the outer_struct.

Let's compile (MSVC 2010):

Listing 1.353: Optimizing MSVC 2010 /Ob0

`$SG2802 DB 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d', 0aH, 00H

_TEXT SEGMENT
_s$ = 8
_f PROC
    mov eax, DWORD PTR _s$[esp+16]
    movsx ecx, BYTE PTR _s$[esp+16]
    mov edx, DWORD PTR _s$[esp+8]
    push eax
    mov eax, DWORD PTR _s$[esp+8]
    push ecx
    mov ecx, DWORD PTR _s$[esp+8]
    push edx
    movsx edx, BYTE PTR _s$[esp+8]
    push eax
    push ecx
    push edx
    push OFFSET $SG2802 ; 'a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d'
    call _printf
    add esp, 28
    ret 0
_f ENDP

_s$ = -24
_main PROC
    sub esp, 24
    push ebx
    push esi
    push edi
    mov ecx, 2
    sub esp, 24
    mov eax, esp

; from this moment, EAX is synonymous to ESP:
    mov BYTE PTR _s$[esp+60], 1
    mov ebx, DWORD PTR _s$[esp+60]
    mov DWORD PTR [eax], ebx
    mov DWORD PTR [eax+4], ecx
    lea edx, DWORD PTR [ecx+98]
    lea esi, DWORD PTR [ecx+99]
    lea edi, DWORD PTR [ecx+2]
    mov DWORD PTR [eax+8], edx
    mov BYTE PTR _s$[esp+76], 3
    mov ecx, DWORD PTR _s$[esp+76]
    mov DWORD PTR [eax+12], esi
    mov DWORD PTR [eax+16], ecx
    mov DWORD PTR [eax+20], edi
    call _f
    add esp, 24
    pop edi
    pop esi
    xor eax, eax
    pop ebx
    add esp, 24
    ret 0
_main ENDP

364
One curious thing here is that by looking onto this assembly code, we do not even see that another structure was used inside of it! Thus, we would say, nested structures are unfolded into linear or one-dimensional structure.

Of course, if we replace the `struct inner_struct c;` declaration with `struct inner_struct *c;` (thus making a pointer here) the situation will be quite different.
OllyDbg

Let's load the example into OllyDbg and take a look at \texttt{outer\_struct} in memory:

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{OllyDbg.png}
\caption{OllyDbg: Before \texttt{printf()} execution}
\end{figure}

That's how the values are located in memory:

- \texttt{(outer\_struct.a)} (byte) 1 + 3 bytes of random garbage;
- \texttt{(outer\_struct.b)} (32-bit word) 2;
- \texttt{(inner\_struct.a)} (32-bit word) 0x64 (100);
- \texttt{(inner\_struct.b)} (32-bit word) 0x65 (101);
- \texttt{(outer\_struct.d)} (byte) 3 + 3 bytes of random garbage;
- \texttt{(outer\_struct.e)} (32-bit word) 4.

1.30.6 Bit fields in a structure

CPUID example

The C/C++ language allows to define the exact number of bits for each structure field. It is very useful if one needs to save memory space. For example, one bit is enough for a \texttt{bool} variable. But of course, it is not rational if speed is important.

Let's consider the CPUID\textsuperscript{157} instruction example. This instruction returns information about the current CPU and its features.

If the EAX is set to 1 before the instruction's execution, CPUID returning this information packed into the EAX register:

\begin{itemize}
\item 3:0 (4 bits)
\item 7:4 (4 bits)
\item 11:8 (4 bits)
\item 13:12 (2 bits)
\item 19:16 (4 bits)
\item 27:20 (8 bits)
\end{itemize}

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stepping</td>
<td>EAX:7-3</td>
</tr>
<tr>
<td>Model</td>
<td>EAX:2-0</td>
</tr>
<tr>
<td>Family</td>
<td>EAX:11-9</td>
</tr>
<tr>
<td>Processor Type</td>
<td>EAX:19-16</td>
</tr>
<tr>
<td>Extended Model</td>
<td>EAX:27-20</td>
</tr>
</tbody>
</table>

MSVC 2010 has CPUID macro, but GCC 4.4.1 does not. So let's make this function by ourselves for GCC with the help of its built-in assembler\textsuperscript{158}.

\textsuperscript{157}wikipedia
\textsuperscript{158}More about internal GCC assembler
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *d) {
    asm volatile("cpuid":="a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"(code));
} #endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX {
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main() {
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
#endif

#ifdef __GNUC__
    cpuid(1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    printf("stepping=%d\n", tmp->stepping);
    printf("model=%d\n", tmp->model);
    printf("family_id=%d\n", tmp->family_id);
    printf("processor_type=%d\n", tmp->processor_type);
    printf("extended_model_id=%d\n", tmp->extended_model_id);
    printf("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
};

After CPUID fills EAX/EBX/ECX/EDX, these registers are to be written in the b[] array. Then, we have a pointer to the CPUID_1_EAX structure and we point it to the value in EAX from the b[] array.

In other words, we treat a 32-bit int value as a structure. Then we read specific bits from the structure.

**MSVC**

Let's compile it in MSVC 2008 with /Ox option:

Listing 1.354: Optimizing MSVC 2008

```
b$ = -16  : size = 16
_main   PROC
  sub     esp, 16
  push    ebx
  xor     ecx, ecx
  mov     eax, 1
  cpuid
  push    esi
```
The SHR instruction shifting the value in EAX by the number of bits that must be skipped, e.g., we ignore some bits at the right side.

The AND instruction clears the unneeded bits on the left, or, in other words, leaves only those bits in the EAX register we need.
MSVC + OllyDbg

Let's load our example into OllyDbg and see, what values are set in EAX/EBX/ECX/EDX after the execution of CPUID:

![OllyDbg screenshot](image)

**Figure 1.109: OllyDbg: After CPUID execution**

EAX has 0x000206A7 (my CPU is Intel Xeon E3-1220).
This is 0b00000000000000100000011010100111 in binary form.

Here is how the bits are distributed by fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>In binary form</th>
<th>In decimal form</th>
</tr>
</thead>
<tbody>
<tr>
<td>reserved2</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>extended_family_id</td>
<td>0000000000</td>
<td>0</td>
</tr>
<tr>
<td>extended_model_id</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>reserved1</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>processor_id</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>family_id</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>model</td>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>stepping</td>
<td>0111</td>
<td>7</td>
</tr>
</tbody>
</table>

**Listing 1.355: Console output**

stepping=7
model=10
family_id=6
processor_type=0
extended_model_id=2
extended_family_id=0

GCC

Let's try GCC 4.4.1 with -O3 option.

**Listing 1.356: Optimizing GCC 4.4.1**

```
main proc near ; DATA XREF: _start+17
    push ebp
    mov ebp, esp
    and esp, 0FFFFFFF0h
    push esi
    mov esi, 1
```
Almost the same. The only thing worth noting is that GCC somehow combines the calculation of extended_model_id and extended_family_id into one block, instead of calculating them separately before each printf() call.

**Handling float data type as a structure**

As we already noted in the section about FPU ([1.25 on page 217](#)), both float and double types consist of a sign, a significand (or fraction) and an exponent. But will we be able to work with these fields directly? Let’s try this with float.

![Exponent and mantissa or fraction](chart)

\( (S — \text{sign}) \)
```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // fractional part
    unsigned int exponent : 8; // exponent + 0x3FF
    unsigned int sign : 1; // sign bit
};

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;
    assert(sizeof(struct float_as_struct) == sizeof(float));
    memcpy(&t, &f, sizeof(float));
    t.sign=1; // set negative sign
    t.exponent=t.exponent+2; // multiply d by 2^n here is 2
    memcpy(&f, &t, sizeof(float));
    return f;
}

int main()
{
    printf("%f\n", f(1.234));
}
```

The float_as_struct structure occupies the same amount of memory as float, i.e., 4 bytes or 32 bits. Now we are setting the negative sign in the input value and also, by adding 2 to the exponent, we thereby multiply the whole number by $2^n$, i.e., by 4.

Let’s compile in MSVC 2008 without optimization turned on:

Listing 1.357: Non-optimizing MSVC 2008

```
_t$ = -8 ; size = 4
_f$ = -4 ; size = 4
__in$ = 8 ; size = 4
?f@YAMM@Z PROC ; f
    push ebp
    mov ebp, esp
    sub esp, 8
    fld DWORD PTR __in$[ebp]
    fstp DWORD PTR _f$[ebp]
    push 4
    lea eax, DWORD PTR _f$[ebp]
    push eax
    lea ecx, DWORD PTR _t$[ebp]
    push ecx
    call _memcpy
    add esp, 12
    mov edx, DWORD PTR _t$[ebp]
    or edx, -2147483648 ; 80000000H - set minus sign
    mov DWORD PTR _t$[ebp], edx
    mov eax, DWORD PTR _t$[ebp]
    shr eax, 23 ; 00000017H - drop significand
    and eax, 255 ; 000000ffH - leave here only exponent
    add eax, 2 ; add 2 to it
```
and eax, 255 ; 000000ffH
shl eax, 23 ; 00000017H - shift result to place of bits 30:23
mov ecx, DWORD PTR t$[ebp]
and ecx, -2139095041 ; 807fffffh - drop exponent

; add original value without exponent with new calculated exponent:
or ecx, eax
mov DWORD PTR _t$[ebp], ecx
push 4
lea edx, DWORD PTR _t$[ebp]
push edx
lea eax, DWORD PTR _f$[ebp]
push eax
call __memcpy
add esp, 12
fld DWORD PTR _f$[ebp]
mov esp, ebp
pop ebp
ret 0
?f@@YAMM@Z ENDP ; f

A bit redundant. If it was compiled with /Ox flag there would be no memcpy() call, the f variable is used directly. But it is easier to understand by looking at the unoptimized version.

What would GCC 4.4.1 with -O3 do?

Listing 1.358: Optimizing GCC 4.4.1

; f(float)
public _Z1ff
_Z1ff proc near
var_4 = dword ptr -4
arg_0 = dword ptr 8

push ebp
mov ebp, esp
sub esp, 4
mov eax, [ebp+arg_0]
or eax, 80000000h ; set minus sign
mov edx, eax
and eax, 807FFFFfh ; leave only sign and significand in EAX
shr edx, 23 ; prepare exponent
add edx, 2 ; add 2
movzx edx, dl ; clear all bits except 7:0 in EDX
shl edx, 23 ; shift new calculated exponent to its place
or eax, edx ; join new exponent and original value without exponent
mov [ebp+var_4], eax
fld [ebp+var_4]
leave
retn
_Z1ff endp

public main
proc near
push ebp
mov ebp, esp
and esp, 0FFFFFF0h
sub esp, 10h
fld ds:dword_8048614 ; -4.936
fstp qword ptr [esp+8]
mov dword ptr [esp+4], offset asc_8048610 ; "%f\n"
mov dword ptr [esp+4], 1
call __printf_chk
xor eax, eax
leave
retn
main endp

372
The \( f() \) function is almost understandable. However, what is interesting is that GCC was able to calculate the result of \( f(1.234) \) during compilation despite all this hodge-podge with the structure fields and prepared this argument to \( \text{printf()} \) as precalculated at compile time!

1.30.7 Exercises

- [http://challenges.re/71](http://challenges.re/71)
- [http://challenges.re/72](http://challenges.re/72)

1.31 The classic \textit{struct} bug

This is a classic \textit{struct} bug.

Here is a sample definition:

```c
struct test {
    int field1;
    int field2;
};
```

And then C files:

```c
void setter(struct test *t, int a, int b) {
    t->field1=a;
    t->field2=b;
}
```

```c
#include <stdio.h>
void printer(struct test *t) {
    printf ("%d\n", t->field1);
    printf ("%d\n", t->field2);
}
```

So far so good.

Now you add a third field into the structure, some place between two fields:

```c
struct test {
    int field1;
    int inserted;
    int field2;
};
```

And you probably modify \textit{setter()} function, but forget about \textit{printer()}:

```c
void setter(struct test *t, int a, int b, int c) {
    t->field1=a;
    t->inserted=b;
    t->field2=c;
}
```

You compile your project, but the C file where \textit{printer()} is residing, isn’t recompiling, because your IDE\textsuperscript{159} or build system has no idea that module depends on a test struct definition. Maybe because \texttt{#include <new.h>} is omitted. Or maybe, \texttt{new.h} header file is included in \texttt{printer.c} via some other header file. The object file remains untouched (IDE thinks it doesn’t need to be recompiled), while \textit{setter()} function is already a new version. These two object files (old and new) eventually linked into an executable file.

\textsuperscript{159}Integrated development environment
Then you run it, and the setter() sets 3 fields at +0, +4 and +8 offsets. However, the printer() only knows about 2 fields, and gets them from +0 and +4 offsets during printing.

This leads to very obscure and nasty bugs. The reason is that IDE or build system or Makefile doesn’t know the fact that both C files (or modules) depends on the header file with test definition. A popular remedy is to clean everything and recompile.

This is true for C++ classes as well, since they works just like structures: 3.21.1 on page 541.

This is a C/C++’s malady, and a source of criticism, yes. Many newer PLs has better support of modules and interfaces. But keep in mind, when C compiler was created: 1970s, on old PDP computers. So everything was simplified down to this by C creators.

1.32 Unions

C/C++ union is mostly used for interpreting a variable (or memory block) of one data type as a variable of another data type.

1.32.1 Pseudo-random number generator example

If we need float random numbers between 0 and 1, the simplest thing is to use a PRNG like the Mersenne twister. It produces random unsigned 32-bit values (in other words, it produces random 32 bits). Then we can transform this value to float and then divide it by RAND_MAX (0xffffffff in our case)—we getting a value in the 0..1 interval.

But as we know, division is slow. Also, we would like to issue as few FPU operations as possible. Can we get rid of the division?

Let’s recall what a floating point number consists of: sign bit, significand bits and exponent bits. We just have to store random bits in all significand bits to get a random float number!

The exponent cannot be zero (the floating number is denormalized in this case), so we are storing 0b01111111 to exponent—this means that the exponent is 1. Then we filling the significand with random bits, set the sign bit to 0 (which means a positive number) and voilà. The generated numbers is to be between 1 and 2, so we must also subtract 1.

A very simple linear congruential random numbers generator is used in my example, it produces 32-bit numbers. The PRNG is initialized with the current time in UNIX timestamp format.

Here we represent the float type as an union—it is the C/C++ construction that enables us to interpret a piece of memory as different types. In our case, we are able to create a variable of type union and then access to it as it is float or as it is uint32_t. It can be said, it is just a hack. A dirty one.

The integer PRNG code is the same as we already considered: 1.29 on page 336. So this code in compiled form is omitted.

```c
#include <stdio.h>
#include <stdint.h>
#include <time.h>

// integer PRNG definitions, data and routines:

// constants from the Numerical Recipes book
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;
uint32_t RNG_state; // global variable

void my_srand(uint32_t i)
{
    RNG_state=i;
};

uint32_t my_rand()
{
    RNG_state=RNG_state*RNG_a+RNG_c;
    return RNG_state;
};
```

160 the idea was taken from: http://go.yurichev.com/17308
// FPU PRNG definitions and routines:

union uint32_t_float
{
    uint32_t i;
    float f;
};

float float_rand()
{
    union uint32_t_float tmp;
    tmp.i = my_rand() & 0x007ffffff | 0x3F800000;
    return (tmp.f-1);
}

// test

int main()
{
    my_srand(time(NULL));  // PRNG initialization
    for (int i=0; i<100; i++)
        printf("%f\n", float_rand());
    return 0;
}

x86

Listing 1.359: Optimizing MSVC 2010

$SG4238 DB 'f', 0AH, 00H

__real@3ff0000000000000 DQ 03ff0000000000000r ; 1

tv130 = -.4
_tmp$ = -.4

.float_rand@YAMXZ PROC
    push ecx
    call ?my_rand@YAI
    ; EAX=pseudorandom value
    and eax, B388607 ; 007ffffffH
    or eax, 1065353216 ; 3f800000H
    ; EAX=pseudorandom value & 0x007ffffff | 0x3f800000
    ; store it into local stack:
    mov DWORD PTR _tmp$[esp+4], eax
    ; reload it as float point number:
    fld DWORD PTR _tmp$[esp+4]
    ; subtract 1.0:
    fsub QWORD PTR __real@3ff0000000000000
    ; store value we got into local stack and reload it:
    fstp DWORD PTR tv130[esp+4]; \ these instructions are redundant
    fld DWORD PTR tv130[esp+4]; /
    pop ecx
    ret 0
.float_rand@YAMXZ ENDP

_main PROC
    push esi
    xor eax, eax
    call _time
    push eax
    call ?my_srand@YAI
    add esp, 4
    mov esi, 100
$LL3@main:
    call ?float_rand@YAMXZ
    sub esp, 8
    fstp QWORD PTR [esp]
Function names are so strange here because this example was compiled as C++ and this is name mangling in C++, we will talk about it later: 3.21.1 on page 542. If we compile this in MSVC 2012, it uses the SIMD instructions for the FPU, read more about it here: 1.38.5 on page 436.

ARM (ARM mode)

Listing 1.360: Optimizing GCC 4.6.3 (IDA)

Listing 1.361: Optimizing GCC 4.6.3 (objdump)

We'll also make a dump in objdump and we'll see that the FPU instructions have different names than in IDA. Apparently, IDA and binutils developers used different manuals? Perhaps it would be good to know both instruction name variants.
40: ed9f0a05 vldr s0, [pc, #20] ; 5c <float_rand+0x24>
44: e3c034ff bic r3, r0, #-16777216 ; 0xff000000
48: e3c33502 bic r3, r3, #8388608 ; 0x800000
4c: e38335fe orr r3, r3, #1065353216 ; 0x3f800000
50: ee073a90 vmov s15, r3
54: ee370ac0 vsub.f32 s0, s15, s0
58: e8bd8008 pop {r3, pc}
5c: 3f800000 svccc 0x00800000

The instructions at 0x5c in float_rand() and at 0x38 in main() are (pseudo-)random noise.

### 1.32.2 Calculating machine epsilon

The machine epsilon is the smallest possible value the FPU can work with. The more bits allocated for floating point number, the smaller the machine epsilon. It is $2^{-23} = 1.19e-07$ for float and $2^{-52} = 2.22e-16$ for double. See also: Wikipedia article.

It’s interesting, how easy it’s to calculate the machine epsilon:

```c
#include <stdio.h>
#include <stdint.h>

union uint_float
{
    uint32_t t i;
    float f;
};

float calculate_machine_epsilon(float start)
{
    union uint_float v;
    v.f=start;
    v.i++;
    return v.f-start;
}

void main()
{
    printf ("%g\n", calculate_machine_epsilon(1.0));
}
```

What we do here is just treat the fraction part of the IEEE 754 number as integer and add 1 to it. The resulting floating number is equal to starting_value+machine_epsilon, so we just have to subtract the starting value (using floating point arithmetic) to measure, what difference one bit reflects in the single precision (float). The union serves here as a way to access IEEE 754 number as a regular integer. Adding 1 to it in fact adds 1 to the fraction part of the number, however, needless to say, overflow is possible, which will add another 1 to the exponent part.

x86
Listing 1.362: Optimizing MSVC 2010

tv130 = 8
_v$ = 8
_start$ = 8
_calculate_machine_epsilon PROC
    fldDWORD PTR _start$[esp-4]
    fstDWORD PTR _v$[esp-4] ; this instruction is redundant
    incDWORD PTR _v$[esp-4]
    fsubrDWORD PTR _v$[esp-4]
    fstpDWORD PTR tv130[esp-4] ; \ this instruction pair is also redundant
    fldDWORD PTR tv130[esp-4] ; /
    ret 0
_calculate_machine_epsilon ENDP

The second FST instruction is redundant: there is no necessity to store the input value in the same place (the compiler decided to allocate the \( v \) variable at the same point in the local stack as the input argument). Then it is incremented with INC, as it is a normal integer variable. Then it is loaded into the FPU as a 32-bit IEEE 754 number, FSUBR does the rest of job and the resulting value is stored in ST0. The last FSTP/FLD instruction pair is redundant, but the compiler didn’t optimize it out.

**ARM64**

Let’s extend our example to 64-bit:

```c
#include <stdio.h>
#include <stdint.h>

typedef union
{
    uint64_t i;
    double d;
} uint_double;

double calculate_machine_epsilon(double start)
{
    uint_double v;
    v.d=start;
    v.i++;    
    return v.d-start;
}

void main()
{
    printf("%g\n", calculate_machine_epsilon(1.0));
};
```

ARM64 has no instruction that can add a number to a FPU D-register, so the input value (that came in D0) is first copied into GPR, incremented, copied to FPU register D1, and then subtraction occurs.

Listing 1.363: Optimizing GCC 4.9 ARM64

calculate_machine_epsilon:
    fmov  x0, d0 ; load input value of double type into X0
    add   x0, x0, 1 ; X0++
    fmov  d1, x0 ; move it to FPU register
    fsub  d0, d1, d0 ; subtract
    ret

See also this example compiled for x64 with SIMD instructions: 1.38.4 on page 435.

**MIPS**

The new instruction here is MTC1 (“Move To Coprocessor 1”), it just transfers data from GPR to the FPU’s registers.

Listing 1.364: Optimizing GCC 4.4.5 (IDA)
calculate_machine_epsilon:
    mfc1  $v0, $f12
Conclusion

It's hard to say whether someone may need this trickery in real-world code, but as was mentioned many times in this book, this example serves well for explaining the IEEE 754 format and *unions* in C/C++.

### 1.32.3 FSCALE instruction replacement

Agner Fog in his *Optimizing subroutines in assembly language / An optimization guide for x86 platforms* work[^1] states that FSCALE FPU instruction (calculating $2^n$) may be slow on many CPUs, and he offers faster replacement.

Here is my translation of his assembly code to C/C++:

```c
#include <stdint.h>
#include <stdio.h>

union uint_float
{
    uint32_t i;
    float f;
};

float flt_2n(int N)
{
    union uint_float tmp;
    tmp.i=((N<<23)+0x3f800000);
    return tmp.f;
};

struct float_as_struct
{
    unsigned int fraction : 23;
    unsigned int exponent : 8;
    unsigned int sign : 1;
};

float flt_2n_v2(int N)
{
    struct float_as_struct tmp;
    tmp.fraction=0;
    tmp.sign=0;
    tmp.exponent=N+0x7f;
    return *(float*)(&tmp);
};

union uint64_double
{
    uint64_t i;
    double d;
};

double dbl_2n(int N)
{
    union uint64_double tmp;
    tmp.i=((uint64_t)N<52)+0x3ff000000000000UL;
    return tmp.d;
};
```

struct double_as_struct
{
    uint64_t fraction : 52;
    int exponent : 11;
    int sign : 1;
};

double dbl_2n_v2(int N)
{
    struct double_as_struct tmp;
    tmp.fraction=0;
    tmp.sign=0;
    tmp.exponent=N+0x3ff;
    return *(double*)(&tmp);
};

int main()
{
    // 2^{11} = 2048
    printf("%f\n", flt_2n(11));
    printf("%f\n", flt_2n_v2(11));
    printf("%lf\n", dbl_2n(11));
    printf("%lf\n", dbl_2n_v2(11));
};

FScale instruction may be faster in your environment, but still, it’s a good example of union’s and the fact that exponent is stored in $2^m$ form, so an input $n$ value is shifted to the exponent in IEEE 754 encoded number. Then exponent is then corrected with addition of 0x3f800000 or 0x3ff0000000000000.

The same can be done without shift using struct, but internally, shift operations still occurred.

### 1.32.4 Fast square root calculation

Another well-known algorithm where float is interpreted as integer is fast calculation of square root.

Listing 1.365: The source code is taken from Wikipedia: [http://go.yurichev.com/17364](http://go.yurichev.com/17364)

```c
/* Assumes that float is in the IEEE 754 single precision floating point format *
* and that int is 32 bits. */
float sqrt_approx(float z)
{
    int val_int = *(int*)&z; /* Same bits, but as an int */
    /*
     * To justify the following code, prove that
     * (((val_int / 2^m) - b) / 2) + b * 2^m = ((val_int - 2^m) / 2) + ((b + 1) / 2) * 2^m
     * where
     * b = exponent bias
     * m = number of mantissa bits
     * */
    val_int -= 1 << 23; /* Subtract 2^m. */
    val_int >>= 1; /* Divide by 2. */
    val_int += 1 << 29; /* Add ((b + 1) / 2) * 2^m. */
    return *(float*)&val_int; /* Interpret again as float */
}
```

As an exercise, you can try to compile this function and to understand, how it works.

There is also well-known algorithm of fast calculation of $\frac{1}{\sqrt{x}}$. Algorithm became popular, supposedly, because it was used in Quake III Arena.

Algorithm description can be found in Wikipedia: [http://go.yurichev.com/17360](http://go.yurichev.com/17360).
1.33  Pointers to functions

A pointer to a function, as any other pointer, is just the address of the function’s start in its code segment. They are often used for calling callback functions.

Well-known examples are:

- qsort(), atexit() from the standard C library;
- *NIX OS signals.
- thread starting: CreateThread() (win32), pthread_create() (POSIX);
- lots of win32 functions, like EnumChildWindows().
- lots of places in the Linux kernel, for example the filesystem driver functions are called via callbacks.
- The GCC plugin functions are also called via callbacks.

So, the qsort() function is an implementation of quicksort in the C/C++ standard library. The function is able to sort anything, any type of data, as long as you have a function to compare these two elements and qsort() is able to call it.

The comparison function can be defined as:

```
int (*compare)(const void *, const void *)
```

Let's use the following example:

```
/* ex3 Sorting ints with qsort */
#include <stdio.h>
#include <stdlib.h>

int comp(const void * _a, const void * _b)
{
    const int *a=(const int *)_a;
    const int *b=(const int *)_b;

    if (*a==*b)
        return 0;
    else
        if (*a < *b)
            return -1;
        else
            return 1;
}

int main(int argc, char* argv[])
{
    int numbers[10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
    int i;

    /* Sort the array */
    qsort(numbers,10,sizeof(int),comp);
    for (i=0;i<9;i++)
        printf("Number = %d\n",numbers[i]);
    return 0;
}
```

1.33.1 MSVC

Let's compile it in MSVC 2010 (some parts were omitted for the sake of brevity) with /Ox option:

```
Listing 1.366: Optimizing MSVC 2010: /GS- /MD

_a$ = 8        ; size = 4
__b$ = 12      ; size = 4
__comp PROC
    mov    eax, DWORD PTR __a$[esp-4]
    mov    ecx, DWORD PTR __b$[esp-4]
    mov    eax, DWORD PTR [eax]
```

381
Nothing surprising so far. As a fourth argument, the address of label _comp is passed, which is just a place where comp() is located, or, in other words, the address of the very first instruction of that function.

How does qsort() call it?

Let's take a look at this function, located in MSVCR80.DLL (a MSVC DLL module with C standard library functions):

Listing 1.367: MSVCR80.DLL
comp—is the fourth function argument. Here the control gets passed to the address in the comp argument. Before it, two arguments are prepared for comp(). Its result is checked after its execution.

That’s why it is dangerous to use pointers to functions. First of all, if you call qsort() with an incorrect function pointer, qsort() may pass control flow to an incorrect point, the process may crash and this bug will be hard to find.

The second reason is that the callback function types must comply strictly, calling the wrong function with wrong arguments of wrong types may lead to serious problems, however, the crashing of the process is not a problem here—the problem is how to determine the reason for the crash—because the compiler may be silent about the potential problems while compiling.
MSVC + OllyDbg

Let's load our example into OllyDbg and set a breakpoint on `comp()`. We can see how the values are compared at the first `comp()` call:

![OllyDbg screenshot](image)

Figure 1.110: OllyDbg: first call of `comp()`

OllyDbg shows the compared values in the window under the code window, for convenience. We can also see that the SP points to RA, where the `qsort()` function is (located in MSVCR100.DLL).
By tracing (F8) until the RETN instruction and pressing F8 one more time, we return to the qsort() function:

That has been a call to the comparison function.

Figure 1.111: OllyDbg: the code in qsort() right after comp() call
Here is also a screenshot of the moment of the second call of `comp()`—now values that have to be compared are different:

![Image of OllyDbg: second call of `comp()`](image)

**Figure 1.112: OllyDbg: second call of `comp()`**

**MSVC + tracer**

Let's also see which pairs are compared. These 10 numbers are being sorted: 1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000.

We got the address of the first `CMP` instruction in `comp()`, it is `0x0040100C` and we’ve set a breakpoint on it:

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

Now we get some information about the registers at the breakpoint:

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000076 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

Let’s filter out `EAX` and `ECX` and we got:

```
EAX=0x00000076 ECX=0x00000005
EAX=0x00000005 ECX=0xffffe7960
EAX=0x00000005 ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0xffffffe7960 ECX=0x00000005
EAX=0x0000000058 ECX=0x00000005
EAX=0x000000058 ECX=0x00000005
EAX=0x0000000058 ECX=0x00000005
EAX=0x0000000058 ECX=0x00000005
EAX=0x0000000058 ECX=0x00000005
```

```
386
That's 34 pairs. Therefore, the quick sort algorithm needs 34 comparison operations to sort these 10 numbers.
MSVC + tracer (code coverage)

We can also use the tracer’s feature to collect all possible register values and show them in IDA.

Let’s trace all instructions in `comp()`:

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

We get an .idc-script for loading into IDA and load it:

```
.text:00401010 .text:00401000 ; int cdecl PtFuncCompare(const void *, const void *)
.text:00401060 PtFuncCompare proc near

.text:00401000 arg_0 = dword ptr 4
.text:00401000 arg_4 = dword ptr 8
.text:00401000
.text:00401000 ; ---

.text:00401010 ; _loc_401010
.text:00401010 xor eax, eax

.text:00401030 ;_loc_401030: ; CODE XREF: PtFuncCompare+E1]
.text:00401013 xor edx, edx
.text:00401015 cmp eax, ecx

.text:00401017 setnl dl ; SF=false,true OF=Fake

.text:0040101A lea eax, [edx+edx-1]
.text:0040101E ret

.text:0040101E PtFuncCompare endp

.text:004011FF
```

Figure 1.113: tracer and IDA. N.B.: some values are cut at right

IDA gave the function a name `(PtFuncCompare)`—because IDA sees that the pointer to this function is passed to `qsort()`.

We see that the \(a\) and \(b\) pointers are pointing to various places in the array, but the step between them is 4, as 32-bit values are stored in the array.

We see that the instructions at `0x401010` and `0x401012` were never executed (so they left as white): indeed, `comp()` has never returned 0, because there no equal elements in the array.

### 1.33.2 GCC

Not a big difference:

```
lea   eax, [esp+4h+var_28]
mov   [esp+4h+var_40], eax
mov   [esp+4h+var_28], 764h
mov   [esp+4h+var_24], 2dh
mov   [esp+4h+var_20], 0ch
mov   [esp+4h+var_1C], 0FFFFFFE9h
mov   [esp+4h+var_18], 0FF7h
mov   [esp+4h+var_14], 5
mov   [esp+4h+var_10], 0FFFFFFC7h
mov   [esp+4h+var_C], 43Fh
mov   [esp+4h+var_8], 58h
mov   [esp+4h+var_4], 0FFFFFFE7960h
mov   [esp+4h+var_34], offset comp
mov   [esp+4h+var_38], 4
mov   [esp+4h+var_3C], 0Ah
```

```
call  _qsort
```

```
The implementation of `qsort()` is located in `libc.so.6` and it is in fact just a wrapper \(^{162}\) for `qsort_r()`. In turn, it is calling `quicksort()`, where our defined function is called via a passed pointer:

```
Listing 1.369: (file libc.so.6, glibc version—2.10.1)
```

```
... .text:0002DDF6        mov    edx, [ebp+arg_10]
 .text:0002DDF9        mov    [esp+4], esi
 .text:0002DDFD        mov    [esp], edi
 .text:0002DE00        mov    [esp+8], edx
 .text:0002DE04        call   [ebp+arg_c]
...
```

**GCC + GDB (with source code)**

Obviously, we have the C-source code of our example (\ref{1.33} on page 381), so we can set a breakpoint (\texttt{b}) on line number 11—the line where the first comparison occurs). We also have to compile the example with debugging information included (\texttt{-g}), so the table with addresses and corresponding line numbers is present.

We can also print values using variable names (\texttt{p}): the debugging information also has tells us which register and/or local stack element contains which variable.

We can also see the stack (\texttt{bt}) and find out that there is some intermediate function `msort_with_tmp()` used in Glibc.

```
Listing 1.370: GDB session
```

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out
Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
11 if (*a==*b)
(gdb) p *a
```

\(^{162}\)A concept like thunk function
$1 = 1892
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17_1.c:11
11    if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0 comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#1 0x87e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=2)
     at msort.c:65
#2 0x87e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0xbffff07c@entry) at msort.c:45
#3 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=5) at msort.c:53
#4 0x87e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0xbffff07c@entry) at msort.c:45
#5 msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, n=n@entry=10) at msort.c:53
#6 0x87e42cf in msort_with_tmp (n=10, b=0xbffff0f8, p=0xbffff07c@entry) at msort.c:45
#7 __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry=4, cmp=cmp@entry=0x804844d <comp>)
     at __gmon_start__.c:307
#8 0x800484d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)

GCC + GDB (no source code)

But often there is no source code at all, so we can disassemble the comp() function (disas), find the very
first CMP instruction and set a breakpoint (b) at that address.

At each breakpoint, we are going to dump all register contents
(info registers). The stack information is also available (bt),
but partially: there is no line number information for comp().

Listing 1.371: GDB session

dennis@ubuntuvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
...
Reading symbols from /home/dennis/polygon/a.out...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
 0x804844d <+0>: push ebp
 0x804844e <+1>: mov ebp,esp
 0x8048450 <+3>: sub esp,0x10
 0x8048453 <+6>: mov eax,DWORD PTR [ebp+0x8]
 0x8048456 <+9>: mov DWORD PTR [eax],eax
 0x8048459 <+12>: mov eax,DWORD PTR [ebp+0xc]
 0x804845c <+15>: mov DWORD PTR [eax],eax
 0x804845f <+18>: mov eax,DWORD PTR [ebp-0x8]
 0x8048462 <+21>: mov edx,DWORD PTR [eax]
 0x8048464 <+23>: mov eax,DWORD PTR [ebp-0x4]
 0x8048467 <+26>: mov eax,DWORD PTR [eax]
 0x8048469 <+28>: cmp edx,eax
 0x804846b <+30>: jne 0x8048474 <comp+39>
 0x804846d <+32>: mov eax,0x0
 0x8048472 <+37>: jmp 0x804848e <comp+65>
 0x8048474 <+39>: mov eax,DWORD PTR [ebp-0x8]
 0x8048477 <+42>: mov edx,DWORD PTR [eax]
 0x8048479 <+44>: mov eax,DWORD PTR [ebp-0x4]
 0x804847c <+47>: mov eax,DWORD PTR [eax]
 0x804847e <+49>: cmp edx,eax
 0x8048480 <+51>: jge 0x8048489 <comp+60>
0x08048482 <+53>:  mov     eax,0xffffffff
0x08048487 <+58>:  jmp     0x0804848e <comp+65>
0x08048489 <+60>:  mov     eax,0x1
0x0804848e <+65>:  leave
0x0804848f <+66>:  ret

End of assembler dump.

(gdb) b *0x08048469
Breakpoint 1 at 0x8048469
(gdb) run
Starting program: /home/dennis/polygon./a.out

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax 0x2d   45
ecx 0xbffff0f8  -1073745672
edx 0x764  1892
ebx 0xb7fc0000  -1208221696
esp 0xbffffeeb8  0xbffffeeb8
ebp 0xbffffec8  0xbffffec8
esi 0xbffff0fc  -1073745668
edi 0xbffff010  -1073745904
eip 0x8048469  0x8048469 <comp+28>
eflags 0x286  [ PF SF IF ]
cs 0x73  115
ss 0x7b  123
ds 0x7b  123
es 0x7b  123
fs 0x0  0
gs 0x33  51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax 0xff7  4087
ecx 0xbffff104  -1073745660
edx 0xffffffff9e  -98
ebx 0xb7fc0000  -1208221696
esp 0xbffffeeb8  0xbffffeeb8
ebp 0xbffffec8  0xbffffec8
esi 0xbffff108  -1073745656
edi 0xbffff010  -1073745904
eip 0x8048469  0x8048469 <comp+28>
eflags 0x282  [ SF IF ]
cs 0x73  115
ss 0x7b  123
ds 0x7b  123
es 0x7b  123
fs 0x0  0
gs 0x33  51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax 0xffffffff9e  -98
ecx 0xbffff100  -1073745664
edx 0xc8  200
ebx 0xb7fc0000  -1208221696
esp 0xbffffeeb8  0xbffffeeb8
ebp 0xbffffec8  0xbffffec8
esi 0xbffff104  -1073745660
edi 0xbffff010  -1073745904
eip 0x8048469  0x8048469 <comp+28>
eflags 0x286  [ PF SF IF ]
cs 0x73  115
ss 0x7b  123
ds 0x7b  123
es 0x7b  123
fs 0x0  0

391
1.33.3 Danger of pointers to functions

As we can see, qsort() function expects a pointer to function which takes two void* arguments and returning integer. If you have several comparison functions in your code (one compares string, another—integers, etc), it’s very easy to mix them up with each other. You could try to sort array of string using function which compares integers, and compiler will not warn you about bug.

1.34 64-bit values in 32-bit environment

In a 32-bit environment, GPR’s are 32-bit, so 64-bit values are stored and passed as 32-bit value pairs.

1.34.1 Returning of 64-bit value

```c
#include <stdint.h>

uint64_t f () {
    return 0x1234567890ABCDEF;
}
```

x86

In a 32-bit environment, 64-bit values are returned from functions in the EDX:EAX register pair.

```
_f PROC
    mov    eax, -1867788817 ; 90abcdefH
    mov    edx, 305419896 ; 12345678H
    ret
_f ENDP
```

ARM

A 64-bit value is returned in the R0-R1 register pair (R1 is for the high part and R0 for the low part):

```
| f | PROC
---|---
LDR | r8,[L0.12]
LDR | r1,[L0.16]
BX  | lr
ENDP
```

```
<table>
<thead>
<tr>
<th>L0.12</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCD 0x90abcdef</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L0.16</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCD 0x12345678</td>
</tr>
</tbody>
</table>
```

163 By the way, 32-bit values are passed as pairs in 16-bit environment in the same way: 3.34.4 on page 645
MIPS

A 64-bit value is returned in the V0-V1 ($2$-$3$) register pair (V0 ($2$) is for the high part and V1 ($3$) for the low part):

```
Listing 1.374: Optimizing GCC 4.4.5 (assembly listing)
li $3, -1867841536 # 0xffffffff90ab0000
li $2, 305397760 # 0x12340000
ori $3, $3, 0xcdef
j $31
ori $2, $2, 0x5678
```

```
Listing 1.375: Optimizing GCC 4.4.5 (IDA)
lui $v1, 0x90AB
lui $v0, 0x1234
li $v1, 0x90ABCDEF
jr $ra
li $v0, 0x12345678
```

### 1.3.4.2 Arguments passing, addition, subtraction

```
#include <stdint.h>

uint64_t f_add (uint64_t a, uint64_t b)
{
    return a+b;
};

void f_add_test ()
{
    #ifdef __GNUC__
        printf ("%lld\n", f_add(12345678901234, 23456789012345));
    #else
        printf ("%I64d\n", f_add(12345678901234, 23456789012345));
    #endif
}

uint64_t f_sub (uint64_t a, uint64_t b)
{
    return a-b;
};
```

x86

```
Listing 1.376: Optimizing MSVC 2012 /Ob1

_a$ = 8   ; size = 8
_b$ = 16  ; size = 8
_f_add PROC
    mov eax, DWORD PTR _a$[esp-4]
    add eax, DWORD PTR _b$[esp-4]
    mov edx, DWORD PTR _a$[esp]
    adc edx, DWORD PTR _b$[esp]
    ret 0
_f_add ENDP
_f_add_test PROC
    push 5461  ; 00001555H
    push 1972608889 ; 75939f79H
    push 2874   ; 00000b3aH
    push 1942892530 ; 73ce2ff2H
    call _f_add
    push edx
    push eax
    push OFFSET $SG1436 ; %I64d', 0aH, 00H
    call _printf
```

393
We can see in the \_f_add\_test() function that each 64-bit value is passed using two 32-bit values, high part first, then low part.

Addition and subtraction occur in pairs as well.

In addition, the low 32-bit part are added first. If carry has been occurred while adding, the CF flag is set.

The following ADC instruction adds the high parts of the values, and also adds 1 if CF = 1.

Subtraction also occurs in pairs. The first SUB may also turn on the CF flag, which is to be checked in the subsequent SBB instruction: if the carry flag is on, then 1 is also to be subtracted from the result.

It is easy to see how the \_f_add() function result is then passed to printf().

GCC code is the same.

ARM

Listing 1.378: Optimizing Keil 6/2013 (ARM mode)
The first 64-bit value is passed in R0 and R1 register pair, the second in R2 and R3 register pair. ARM has the ADC instruction as well (which counts carry flag) and SBC ("subtract with carry"). Important thing: when the low parts are added/subtracted, ADDS and SUBS instructions with -S suffix are used. The -S suffix stands for "set flags", and flags (esp. carry flag) is what consequent ADC/SBC instructions definitely need. Otherwise, instructions without the -S suffix would do the job (ADD and SUB).

**MIPS**

Listing 1.379: Optimizing GCC 4.4.5 (IDA)

```
f_add:  
; $a0 - high part of a  
; $a1 - low part of a  
; $a2 - high part of b  
; $a3 - low part of b  
    addu $v1, $a3, $a1 ; sum up low parts  
    addu $a0, $a2, $a0 ; sum up high parts  
    ; will carry generated while summing up low parts?  
    ; if yes, set $v0 to 1  
    sltu $v0, $v1, $a3  
    jr $ra  
    ; add 1 to high part of result if carry should be generated:  
    addu $v0, $a0 ; branch delay slot  
    ; $v0 - high part of result  
    ; $v1 - low part of result

f_sub:  
; $a0 - high part of a  
; $a1 - low part of a  
; $a2 - high part of b  
; $a3 - low part of b  
    subu $v1, $a1, $a3 ; subtract low parts  
    subu $v0, $a0, $a2 ; subtract high parts  
    ; will carry generated while subtracting low parts?  
    ; if yes, set $a0 to 1  
    sltu $a1, $v1  
    jr $ra  
    ; subtract 1 from high part of result if carry should be generated:  
    subu $v0, $a1 ; branch delay slot  
    ; $v0 - high part of result  
    ; $v1 - low part of result
```
MIPS has no flags register, so there is no such information present after the execution of arithmetic operations. So there are no instructions like x86’s ADC and SBB. To know if the carry flag would be set, a comparison (using SLTU instruction) also occurs, which sets the destination register to 1 or 0. This 1 or 0 is then added or subtracted to/from the final result.

### 1.34.3 Multiplication, division

```c
#include <stdint.h>

uint64_t f_mul (uint64_t a, uint64_t b) {
    return a*b;
};

uint64_t f_div (uint64_t a, uint64_t b) {
    return a/b;
};

uint64_t f_rem (uint64_t a, uint64_t b) {
    return a % b;
};
```

### x86

Listing 1.380: Optimizing MSVC 2013 /Ob1

```assembly
a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_f_mul PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _b$[ebp+4]
    push eax
    mov ecx, DWORD PTR _a$[ebp]
    push ecx
    mov edx, DWORD PTR _a$[ebp+4]
```

396
Multiplication and division are more complex operations, so usually the compiler embeds calls to a library functions doing that.

These functions are described here: .5 on page 1029.

Listing 1.381: Optimizing GCC 4.8.1 -fno-inline

```
_f_mul:
push ebx
mov edx, DWORD PTR [esp+8]
mov eax, DWORD PTR [esp+16]
mov ebx, DWORD PTR [esp+12]
imul ebx, eax
imul ecx, edx
mul edx
add ecx, ebx
add edx, ecx
pop ebx
ret

_f_div:
sub esp, 28
mov eax, DWORD PTR [esp+40]
mov edx, DWORD PTR [esp+44]
mov DWORD PTR [esp+8], eax
mov eax, DWORD PTR [esp+32]
```
mov DWORD PTR [esp+12], edx
mov edx, DWORD PTR [esp+36]
mov DWORD PTR [esp], eax
mov DWORD PTR [esp+4], edx
call ___udivdi3 ; unsigned division
add esp, 28
ret

_f_rem:
sub esp, 28
mov eax, DWORD PTR [esp+40]
mov edx, DWORD PTR [esp+44]
mov DWORD PTR [esp+8], eax
mov eax, DWORD PTR [esp+32]
mov DWORD PTR [esp+12], edx
mov edx, DWORD PTR [esp+36]
mov DWORD PTR [esp], eax
mov DWORD PTR [esp+4], edx
call ___umodi3 ; unsigned modulo
add esp, 28
ret

GCC does the expected, but the multiplication code is inlined right in the function, thinking it could be more efficient. GCC has different library function names: .4 on page 1029.

ARM

Keil for Thumb mode inserts library subroutine calls:

Listing 1.382: Optimizing Keil 6/2013 (Thumb mode)

```
||f_mul|| PROC
  PUSH {r4,lr}
  BL __aeabi_lmul
  POP {r4,pc}
ENDP

||f_div|| PROC
  PUSH {r4,lr}
  BL __aeabi_uldivmod
  POP {r4,pc}
ENDP

||f_rem|| PROC
  PUSH {r4,lr}
  BL __aeabi_uldivmod
  MOV r0,r2
  MOV r1,r3
  POP {r4,pc}
ENDP
```

Keil for ARM mode, on the other hand, is able to produce 64-bit multiplication code:

Listing 1.383: Optimizing Keil 6/2013 (ARM mode)

```
||f_mul|| PROC
  PUSH {r4,lr}
  UMULL r12,r4,r0,r2
  MLA r1,r2,r1,r4
  MLA r1,r0,r3,r1
  MOV r0,r12
  POP {r4,pc}
ENDP

||f_div|| PROC
  PUSH {r4,lr}
  BL __aeabi_uldivmod
  POP {r4,pc}
ENDP
```
MIPS

Optimizing GCC for MIPS can generate 64-bit multiplication code, but has to call a library routine for 64-bit division:

Listing 1.384: Optimizing GCC 4.4.5 (IDA)

f_mul:

```mips
mult $a2, $a1
mflo $v0
or $at, $zero ; NOP
or $at, $zero ; NOP
mult $a0, $a3
mflo $a0
addu $v0, $a0
or $at, $zero ; NOP
multu $a3, $a1
mfhi $a2
mflo $v1
jr $ra
addu $v0, $a2
```

f_div:

```mips
var_10 = -0x10
var_4 = -4

lui $gp, (__gnu_local_gp >> 16)
addiu $sp, -0x20
la $gp, (__gnu_local_gp & 0xFFFF)
sw $ra, 0x20+var_4($sp)
sw $gp, 0x20+var_10($sp)
lw $t9, (__udivdi3 & 0xFFFF)($gp)
or $at, $zero
jalr $t9
or $at, $zero
lw $ra, 0x20+var_4($sp)
or $at, $zero
jr $ra
addiu $sp, 0x20
```

f_rem:

```mips
var_10 = -0x10
var_4 = -4

lui $gp, (__gnu_local_gp >> 16)
addiu $sp, -0x20
la $gp, (__gnu_local_gp & 0xFFFF)
sw $ra, 0x20+var_4($sp)
sw $gp, 0x20+var_10($sp)
lw $t9, (__umoddi3 & 0xFFFF)($gp)
or $at, $zero
jalr $t9
or $at, $zero
lw $ra, 0x20+var_4($sp)
or $at, $zero
jr $ra
addiu $sp, 0x20
```
There are a lot of NOPs, probably delay slots filled after the multiplication instruction (it's slower than other instructions, after all).

### 1.34.4 Shifting right

```c
#include <stdint.h>

uint64_t f(uint64_t a)
{
    return a>>7;
};
```

**x86**

Listing 1.385: Optimizing MSVC 2012 /Ob1

```
_a$ = 8 ; size = 8
_f PROC
    mov  eax, DWORD PTR _a$[esp-4]
    mov  edx, DWORD PTR _a$[esp]
    shrd eax, edx, 7
    shr  edx, 7
    ret
_f ENDP
```

Listing 1.386: Optimizing GCC 4.8.1 -fno-inline

```
f:
    mov  edx, DWORD PTR [esp+8]
    mov  eax, DWORD PTR [esp+4]
    shrd eax, edx, 7
    shr  edx, 7
    ret
```

Shifting also occurs in two passes: first the lower part is shifted, then the higher part. But the lower part is shifted with the help of the SHRD instruction, it shifts the value of EAX by 7 bits, but pulls new bits from EDX, i.e., from the higher part. In other words, 64-bit value from EDX:EAX register’s pair, as a whole, is shifted by 7 bits and lowest 32 bits of result are placed into EAX. The higher part is shifted using the much more popular SHR instruction: indeed, the freed bits in the higher part must be filled with zeros.

**ARM**

ARM doesn’t have such instruction as SHRD in x86, so the Keil compiler ought to do this using simple shifts and OR operations:

Listing 1.387: Optimizing Keil 6/2013 (ARM mode)

```
||f|| PROC
    LSR  r0,r0,#7
    ORR  r0,r0,r1,LSL #25
    LSR  r1,r1,#7
    BX   lr
ENDP
```

Listing 1.388: Optimizing Keil 6/2013 (Thumb mode)

```
||f|| PROC
    LSLS  r2,r1,#25
    LSRS  r0,r0,#7
    ORRS  r0,r0,r2
    LSRS  r1,r1,#7
    BX   lr
ENDP
```
MIPS

GCC for MIPS follows the same algorithm as Keil does for Thumb mode:

```
Listing 1.389: Optimizing GCC 4.4.5 (IDA)
f:
  sll $v0, $a0, 25
  srl $v1, $a1, 7
  or $v1, $v0, $v1
  jr $ra
  srl $v0, $a0, 7
```

1.34.5 Converting 32-bit value into 64-bit one

```c
#include <stdint.h>

int64_t f (int32_t a)
{
    return a;
}
```

x86

```
Listing 1.390: Optimizing MSVC 2012

_a$ = 8
_f PROC
    mov eax, DWORD PTR _a$[esp-4]
    cdq
    ret 0
_f ENDP
```

Here we also run into necessity to extend a 32-bit signed value into a 64-bit signed one. Unsigned values are converted straightforwardly: all bits in the higher part must be set to 0. But this is not appropriate for signed data types: the sign has to be copied into the higher part of the resulting number.

The CDQ instruction does that here, it takes its input value in EAX, extends it to 64-bit and leaves it in the EDX:EAX register pair. In other words, CDQ gets the number sign from EAX (by getting the most significant bit in EAX), and depending of it, sets all 32 bits in EDX to 0 or 1. Its operation is somewhat similar to the MOVSX instruction.

ARM

```
Listing 1.391: Optimizing Keil 6/2013 (ARM mode)

||f|| PROC
    ASR r1,r0,#31
    BX lr
ENDP
```

Keil for ARM is different: it just arithmetically shifts right the input value by 31 bits. As we know, the sign bit is MSB, and the arithmetical shift copies the sign bit into the “emerged” bits. So after “ASR r1,r0,#31”, R1 containing 0xFFFFFFFF if the input value has been negative and 0 otherwise. R1 contains the high part of the resulting 64-bit value. In other words, this code just copies the MSB (sign bit) from the input value in R0 to all bits of the high 32-bit part of the resulting 64-bit value.

MIPS

GCC for MIPS does the same as Keil did for ARM mode:

```
Listing 1.392: Optimizing GCC 4.4.5 (IDA)
f:
  sra $v0, $a0, 31
  jr $ra
  move $v1, $a0
```
1.35 LARGE_INTEGER structure case

Imagine this: late 1980s, you’re Microsoft, and you’re developing a new serious OS (Windows NT), that will compete with Unices. Target platforms has both 32-bit and 64-bit CPUs. And you need a 64-bit integer datatype for all sort of purposes, starting at FILETIME\textsuperscript{164} structure.

The problem: not all target C/C++ compilers support 64-bit integer yet (this is late 1980s). Surely, this will be changed in (near) future, but not now. What would you do?

While reading this, try to stop (and/or close this book) and think, how can you solve this problem.

\textsuperscript{164}\url{https://docs.microsoft.com/en-us/windows/desktop/api/minwinbase/ns-minwinbase-filetime}
This is what Microsoft did, something like this:

```c
union ULARGE_INTEGER
{
    struct backward_compatibility
    {
        DWORD LowPart;
        DWORD HighPart;
    };
#ifdef NEW_FANCY_COMPILER_SUPPORTING_64_BIT
    ULONGLONG QuadPart;
#endif
};
```

This is a chunk of 8 bytes, which can be accessed via 64-bit integer QuadPart (if compiled using newer compiler), or using two 32-bit integers (if compiled using old one).

QuadPart field is just absent here when compiled using old compiler.

Order is crucial: first field (LowPart) maps to lower 4 bytes of 64-bit value, second field (HighPart) maps to higher 4 bytes.

Microsoft also added utility functions for all the arithmetical operation, in a same manner as I already described: 1.34 on page 392.

And this is from the leaked Windows 2000 source code base:

**Listing 1.393: i386 arch**

```c
+++;
;
; LARGE_INTEGER
; RtlLargeIntegerAdd (;
; IN LARGE_INTEGER Addend1,;
; IN LARGE_INTEGER Addend2
; )
;
; Routine Description:
;
; This function adds a signed large integer to a signed large integer and
; returns the signed large integer result.
;
; Arguments:
;
; (TOS+4) = Addend1 - first addend value
; (TOS+12) = Addend2 - second addend value
;
; Return Value:
;
; The large integer result is stored in (edx:eax)
;
);

cPublicProc_RtlLargeIntegerAdd,4

cPublicFpo 4,0

    mov    eax,[esp]+4 ; (eax)=add1.low
    add    eax,[esp]+12 ; (eax)=sum.low
    mov    edx,[esp]+8  ; (edx)=add1.hi
    adc    edx,[esp]+16 ; (edx)=sum.hi

stdRET_RtlLargeIntegerAdd
```

**Listing 1.394: MIPS arch**

```c
LEAF_ENTRY(RtlLargeIntegerAdd)

lw     t0,4 * 4(sp)  // get low part of addend2 value
lw     t1,4 * 5(sp)  // get high part of addend2 value
```

---

\(^{165}\) Not a copypasted source code, I wrote this
Now two 64-bit architectures:

Listing 1.395: Itanium arch

```
LEAF_ENTRY(RtlLargeIntegerAdd)
add v0 = a0, a1 // add both quadword arguments
LEAF_RETURN
```

Listing 1.396: DEC Alpha arch

```
LEAF_ENTRY(RtlLargeIntegerAdd)
addq a0, a1, v0 // add both quadword arguments
ret zero, (ra) // return
```

No need using 32-bit instructions on Itanium and DEC Alpha—64-bit ones are here already.

And this is what we can find in Windows Research Kernel:

```
DECLSPEC_DEPRECATED_DDK // Use native __int64 math
inline LARGE_INTEGER NTAPI RtlLargeIntegerAdd (  
    LARGE_INTEGER Addend1,
    LARGE_INTEGER Addend2  
)  
{
    LARGE_INTEGER Sum;
    return Sum;
}
```

All these functions can be dropped (in future), but now they just operate on QuadPart field. If this piece of code is to be compiled using a modern 32-bit compiler (that supports 64-bit integer), it will generate two 32-bit additions under the hood. From this moment, LowPart/HighPart fields can be dropped from the LARGE_INTEGER union/structure.

Would you use such a technique today? Probably not, but if someone would need 128-bit integer data type, you can implement it just like this.

Also, needless to say, this works thanks to little-endian (2.8 on page 460) (all architectures Windows NT was developed for are little-endian). This trick wouldn’t be possible on a big-endian architecture.

### 1.36 SIMD

**SIMD** is an acronym: *Single Instruction, Multiple Data*.

As its name implies, it processes multiple data using only one instruction.

Like the FPU, that CPU subsystem looks like a separate processor inside x86.

SIMD began as MMX in x86. 8 new 64-bit registers appeared: MM0-MM7.
Each MMX register can hold 2 32-bit values, 4 16-bit values or 8 bytes. For example, it is possible to add 8 8-bit values (bytes) simultaneously by adding two values in MMX registers.

One simple example is a graphics editor that represents an image as a two dimensional array. When the user changes the brightness of the image, the editor must add or subtract a coefficient to/from each pixel value. For the sake of brevity if we say that the image is grayscale and each pixel is defined by one 8-bit byte, then it is possible to change the brightness of 8 pixels simultaneously.

By the way, this is the reason why the saturation instructions are present in SIMD.

When the user changes the brightness in the graphics editor, overflow and underflow are not desirable, so there are addition instructions in SIMD which are not adding anything if the maximum value is reached, etc.

When MMX appeared, these registers were actually located in the FPU's registers. It was possible to use either FPU or MMX at the same time. One might think that Intel saved on transistors, but in fact the reason of such symbiosis was simpler — older OSes that are not aware of the additional CPU registers would not save them at the context switch, but saving the FPU registers. Thus, MMX-enabled CPU + old OS + process utilizing MMX features will still work.

SSE—is extension of the SIMD registers to 128 bits, now separate from the FPU.

AVX—another extension, to 256 bits.

Now about practical usage.

Of course, this is memory copy routines (memcpy), memory comparing (memcmp) and so on.

One more example: the DES encryption algorithm takes a 64-bit block and a 56-bit key, encrypt the block and produces a 64-bit result. The DES algorithm may be considered as a very large electronic circuit, with wires and AND/OR/NOT gates.

Bitslice DES— is the idea of processing groups of blocks and keys simultaneously. Let's say, variable of type unsigned int on x86 can hold up to 32 bits, so it is possible to store there intermediate results for 32 block-key pairs simultaneously, using 64+56 variables of type unsigned int.

There is an utility to brute-force Oracle RDBMS passwords/hashes (ones based on DES), using slightly modified bitslice DES algorithm for SSE2 and AVX—now it is possible to encrypt 128 or 256 block-keys pairs simultaneously.

1.36.1 Vectorization

Vectorization is when, for example, you have a loop taking couple of arrays for input and producing one array. The loop body takes values from the input arrays, does something and puts the result into the output array. Vectorization is to process several elements simultaneously.

Vectorization is not very fresh technology: the author of this textbook saw it at least on the Cray Y-MP supercomputer line from 1988 when he played with its “lite” version Cray Y-MP EL.

For example:

```c
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

This fragment of code takes elements from A and B, multiplies them and saves the result into C. If each array element we have is 32-bit int, then it is possible to load 4 elements from A into a 128-bit XMM-register, from B to another XMM-registers, and by executing PMULLD (Multiply Packed Signed Dword Integers and Store Low Result) and PMULHW (Multiply Packed Signed Integers and Store High Result), it is possible to get 4 64-bit products at once.

Thus, loop body execution count is 1024/4 instead of 1024, that is 4 times less and, of course, faster.

---

166 [http://go.yurichev.com/17329](http://go.yurichev.com/17329)
167 [Wikipedia: vectorization](http://go.yurichev.com/17881)
168 Remotely. It is installed in the museum of supercomputers: [http://go.yurichev.com/17081](http://go.yurichev.com/17081)
**Addition example**

Some compilers can do vectorization automatically in simple cases, e.g., Intel C++\textsuperscript{169}.

Here is tiny function:

```c
int f (int sz, int *ar1, int *ar2, int *ar3)
{
    for (int i=0; i<sz; i++)
        ar3[i]=ar1[i]+ar2[i];
    return 0;
}
```

**Intel C++**

Let's compile it with Intel C++ 11.1.051 win32:

```sh
cpl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

We got (in IDA):

```
; int __cdecl f(int, int *, int *, int *)
    public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near
var_10 = dword ptr -10h
sz    = dword ptr 4
ar1   = dword ptr 8
ar2   = dword ptr 0Ch
ar3   = dword ptr 10h
    push    edi
    push    esi
    push    ebx
    push    esi
    mov     edx, [esp+10h+sz]
    test    edx, edx
    jle     loc_15B
    mov     eax, [esp+10h+ar3]
    cmp     edx, 6
    jle     loc_143
    cmp     eax, [esp+10h+ar2]
    jbe     short loc_36
    mov     esi, [esp+10h+ar2]
    sub     esi, eax
    lea     ecx, ds:0[edx*4]
    neg     esi
    cmp     ecx, esi
    jbe     short loc_55

loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
    cmp     eax, [esp+10h+ar2]
    jnb     loc_143
    mov     esi, [esp+10h+ar2]
    sub     esi, eax
    lea     ecx, ds:0[edx*4]
    cmp     esi, ecx
    jb      loc_143

loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
    cmp     eax, [esp+10h+ar1]
    jbe     short loc_67
    mov     esi, [esp+10h+ar1]
    sub     esi, eax
    neg     esi
    cmp     ecx, esi
    jbe     short loc_7F
```

\textsuperscript{169}More about Intel C++ automatic vectorization: Excerpt: Effective Automatic Vectorization

406
loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
    cmp    eax, [esp+10h+ar1]
    jnb    loc_143
    mov    esi, [esp+10h+ar1]
    sub    esi, eax
    cmp    esi, ecx
    jb     loc_143

loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
    mov    edi, eax ; edi = ar3
    and    edi, 0
    jz     short loc_9A ; yes!
    test   edi, 3
    jnz    loc_162
    neg    edi
    add    edi, 10h
    shr    edi, 2

loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84
    lea    ecx, [edi+4]
    cmp    edx, ecx
    jl     loc_162
    mov    ecx, edx
    sub    ecx, edi
    and    ecx, 3
    neg    ecx
    add    ecx, edx
    test   edi, edi
    jbe    short loc_D6
    mov    ebx, [esp+10h+ar2]
    mov    [esp+10h+var 10], ecx
    mov    ecx, [esp+10h+ar1]
    xor    esi, esi

loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD
    mov    edx, [ecx+esi*4]
    add    edx, [ebx+esi*4]
    mov    [eax+esi*4], edx
    inc    esi
    cmp    esi, edi
    jb     short loc_C1
    mov    ecx, [esp+10h+var_10]
    mov    edx, [esp+10h+sz]

loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2
    mov    esi, [esp+10h+ar2]
    lea    esi, [esi+edi*4] ; is ar2+i*4 16-byte aligned?
    test   esi, 0
    jz     short loc_109 ; yes!
    mov    ebx, [esp+10h+ar1]
    mov    esi, [esp+10h+ar2]

loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105
    movdqu xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
    movdqu xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte aligned, so load it to
    paddd  xmm1, xmm0
    movdqa xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
    add    edi, 4
    cmp    edi, ecx
    jb     short loc_ED
    jmp    short loc_127

loc_109: ; CODE XREF: f(int,int *,int *,int *)+E3
    mov    ebx, [esp+10h+ar1]
    mov    esi, [esp+10h+ar2]

loc_111: ; CODE XREF: f(int,int *,int *,int *)+125
    movdqu xmm0, xmmword ptr [ebx+edi*4]
The SSE2-related instructions are:

- **MOVQ** (*Move Unaligned Double Quadword*)—just loads 16 bytes from memory into a XMM-register.
- **PADDD** (*Add Packed Integers*)—adds 4 pairs of 32-bit numbers and leaves the result in the first operand. By the way, no exception is raised in case of overflow and no flags are to be set, just the low 32 bits of the result are to be stored. If one of PADDD’s operands is the address of a value in memory, then the address must be aligned on a 16-byte boundary. If it is not aligned, an exception will be triggered.
- **MOVQ** (*Move Aligned Double Quadword*) is the same as MOVQ, but requires the address of the value in memory to be aligned on a 16-bit boundary. If it is not aligned, exception will be raised. MOVQ works faster than MOVQ, but requires aforesaid.

So, these SSE2-instructions are to be executed only in case there are more than 4 pairs to work on and the pointer ar3 is aligned on a 16-byte boundary.

Also, if ar2 is aligned on a 16-byte boundary as well, this fragment of code is to be executed:

```assembly
movdqa xmm0, xmmword ptr [eax+edi*4], xmm0
add edi, 4
cmp edi, ecx
jb short loc_111

loc_127: ; CODE XREF: f(int,int *,int *,int *)+107
    ; f(int,int *,int *,int *)+164
cmp ecx, edx
jnb short loc_15B
mov esi, [esp+10h+ar1]
mov edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
    mov ebx, [esi+ecx*4]
    add ebx, [edi+ecx*4]
    mov [eax+ecx*4], ebx
    inc ecx
    cmp ecx, edx
    jb short loc_133
    jmp short loc_15B

loc_143: ; CODE XREF: f(int,int *,int *,int *)+17
    ; f(int,int *,int *,int *)+3A ...
    mov esi, [esp+10h+ar1]
    mov edi, [esp+10h+ar2]
xor ecx, ecx

loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159
    mov ebx, [esi+ecx*4]
    add ebx, [edi+ecx*4]
    mov [eax+ecx*4], ebx
    inc ecx
    cmp ecx, edx
    jb short loc_14D

loc_15B: ; CODE XREF: f(int,int *,int *,int *)+A
    ; f(int,int *,int *,int *)+129 ...
xor eax, eax
    pop ecx
    pop ebx
    pop esi
    pop edi
    retn

loc_162: ; CODE XREF: f(int,int *,int *,int *)+8C
    ; f(int,int *,int *,int *)+9F
    xor ecx, ecx
    jmp short loc_127

?f@@YAHHPAH00@Z endp
```
Otherwise, the value from ar2 is to be loaded into XMM0 using MOVDQU, which does not require aligned pointer, but may work slower:

```assembly
movdqu xmm1, xmmword ptr [ebx+edi*4]; ar1+i*4
movdqu xmm0, xmmword ptr [esi+edi*4]; ar2+i*4 is not 16-byte aligned, so load it to XMM0
paddb xmm1, xmm0
movdqa xmmword ptr [eax+edi*4], xmm1; ar3+i*4
```

In all other cases, non-SSE2 code is to be executed.

**GCC**

GCC may also vectorize in simple cases\(^{170}\), if the -O3 option is used and SSE2 support is turned on: -msse2. What we get (GCC 4.4.1):

```assembly
; f(int, int *, int *, int *)
   public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near
var_18 = dword ptr -18h
var_14 = dword ptr -14h
var_10 = dword ptr -10h
arg_0 = dword ptr 8
arg_4 = dword ptr 0Ch
arg_8 = dword ptr 10h
arg_C = dword ptr 14h

push   ebp
mov    ebp, esp
push   edi
push   esi
push   ebx
sub    esp, 0Ch
mov    ecx, [ebp+arg_0]
mov    esi, [ebp+arg_4]
mov    edi, [ebp+arg_8]
mov    ebx, [ebp+arg_C]
test   ecx, ecx
jle    short loc_80484D8
cmp    ecx, 6
lea    eax, [ebx+10h]
ja     short loc_80484E8

loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
                  ; f(int,int *,int *,int *)+61 ...
   xor    eax, eax
   nop
   lea    esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
mov    edx, [edi+eax*4]
add    edx, [esi+eax*4]
mov    [ebx+eax*4], edx
add    eax, 1
cmp    eax, ecx
jnz    short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
                  ; f(int,int *,int *,int *)+A5
   add    esp, 0Ch
   xor    eax, eax
   pop    ebx
   pop    esi
   pop    edi
```

\(^{170}\)More about GCC vectorization support: [http://go.yurichev.com/17083](http://go.yurichev.com/17083)
pop   ebp
ret
align 8

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
test   bl, 0Fh
jnz    short loc_80484C1
lea    edx, [esi+10h]
cmp    ebx, edx
jbe    loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
lea    edx, [edi+10h]
cmp    ebx, edx
ja     short loc_8048503
cmp    edi, eax
jbe    short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
mov    eax, ecx
shr    eax, 2
mov    [ebp+var_14], eax
shl    eax, 2
test   eax, eax
mov    [ebp+var_10], eax
jz     short loc_8048547
mov    [ebp+var_18], ecx
mov    ecx, [ebp+var_14]
xor    eax, eax
xor    edx, edx
nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
movdqu xmm1, xmmword ptr [edi+eax]
movdqu xmm0, xmmword ptr [esi+eax]
add    edx, 1
paddq  xmm0, xmm1
movdqa xmmword ptr [ebx+eax], xmm0
add    eax, 10h
cmp    edx, ecx
jb     short loc_8048520
mov    ecx, [ebp+var_18]
mov    eax, [ebp+var_10]
cmp    ecx, eax
jz     short loc_80484D8

loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
lea    edx, ds:0[eax*4]
add    esi, edx
add    edi, edx
add    ebx, edx
lea    esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
mov    edx, [edi]
add    eax, 1
add    edi, 4
add    edx, [esi]
add    esi, 4
mov    [ebx], edx
add    ebx, 4
cmp    ecx, eax
jg     short loc_8048558
add    esp, 0Ch
xor    eax, eax
pop    ebx
pop    esi
pop    edi
pop    ebp
Almost the same, however, not as meticulously as Intel C++.

**Memory copy example**

Let’s revisit the simple memcpy() example (1.22.2 on page 194):

```c
#include <stdio.h>

void my_memcpy (unsigned char* dst, unsigned char* src, size_t cnt)
{
    size_t i;
    for (i=0; i<cnt; i++)
        dst[i]=src[i];
}
```

And that’s what optimizations GCC 4.9.1 did:

```assembly
my_memcpy:
; RDI = destination address
; RSI = source address
; RDX = size of block
    test    rdx, rdx
    je      .L41
    lea     rax, [rdi+16]
    cmp     rsi, rax
    lea     rax, [rsi+16]
    setae   cl
    cmp     rdi, rax
    setae   al
    or      cl, al
    je      .L13
    cmp     rdx, 22
    jbe     .L13
    mov     rcx, rsi
    push    rbp
    push    rbx
    neg     rcx
    and     ecx, 15
    cmp     rcx, rdx
    cmova   rcx, rdx
    xor     eax, eax
    test    rcx, rcx
    je      .L4
    movzx   eax, BYTE PTR [rsi]
    cmp     rcx, 1
    mov     BYTE PTR [rdi], al
    je      .L15
    movzx   eax, BYTE PTR [rsi+1]
    cmp     rcx, 2
    mov     BYTE PTR [rdi+1], al
    je      .L16
    movzx   eax, BYTE PTR [rsi+2]
    cmp     rcx, 3
    mov     BYTE PTR [rdi+2], al
    je      .L17
    movzx   eax, BYTE PTR [rsi+3]
    cmp     rcx, 4
    mov     BYTE PTR [rdi+3], al
    je      .L18
    movzx   eax, BYTE PTR [rsi+4]
```

411
cmp   rcx, 5
mov   BYTE PTR [rdi+4], al
je    .L19
movzx  eax, BYTE PTR [rsi+5]
cmp   rcx, 6
mov   BYTE PTR [rdi+5], al
je    .L20
movzx  eax, BYTE PTR [rsi+6]
cmp   rcx, 7
mov   BYTE PTR [rdi+6], al
je    .L21
movzx  eax, BYTE PTR [rsi+7]
cmp   rcx, 8
mov   BYTE PTR [rdi+7], al
je    .L22
movzx  eax, BYTE PTR [rsi+8]
cmp   rcx, 9
mov   BYTE PTR [rdi+8], al
je    .L23
movzx  eax, BYTE PTR [rsi+9]
cmp   rcx, 10
mov   BYTE PTR [rdi+9], al
je    .L24
movzx  eax, BYTE PTR [rsi+10]
cmp   rcx, 11
mov   BYTE PTR [rdi+10], al
je    .L25
movzx  eax, BYTE PTR [rsi+11]
cmp   rcx, 12
mov   BYTE PTR [rdi+11], al
je    .L26
movzx  eax, BYTE PTR [rsi+12]
cmp   rcx, 13
mov   BYTE PTR [rdi+12], al
je    .L27
movzx  eax, BYTE PTR [rsi+13]
cmp   rcx, 15
mov   BYTE PTR [rdi+13], al
jne   .L28
movzx  eax, BYTE PTR [rsi+14]
mov   BYTE PTR [rdi+14], al
mov   eax, 15

.L4:
mov   r10, rdx
lea    r9, [rdx-1]
sub    r10, rcx
lea    r8, [r10-16]
sub    r9, rcx
shr    r8, 4
add    r8, 1
mov    r11, r8
sal    r11, 4
cmp    r9, 14
jbe    .L6
lea    rbp, [rsi+rcx]
xor    r9d, r9d
add    rcx, rdi
xor    ebx, ebx

.L7:
movdqa  xmm0, XMMWORD PTR [rbp+0+r9]
add    rbx, 1
movups  XMMWORD PTR [rcx+r9], xmm0
add    r9, 16
cmp    rbx, r8
jb     .L7
add    rax, r11
cmp    r10, r11
je     .L1

.L6:
movzx  ecx, BYTE PTR [rsi+rax]
movBYTE PTR [rdi+rax], cl
lea rcx, [rax+1]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+1+rax]
lea rcx, [rax+2]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+2+rax]
lea rcx, [rax+3]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+3+rax]
lea rcx, [rax+4]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+4+rax]
lea rcx, [rax+5]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+5+rax]
lea rcx, [rax+6]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+6+rax]
lea rcx, [rax+7]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+7+rax]
lea rcx, [rax+8]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+8+rax]
lea rcx, [rax+9]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+9+rax]
lea rcx, [rax+10]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+10+rax]
lea rcx, [rax+11]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+11+rax]
lea rcx, [rax+12]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+12+rax]
lea rcx, [rax+13]
cmp rdx, rcx
jbe .L1
movzx ecx, BYTE PTR [rsi+13+rax]
lea rcx, [rax+14]
cmp rdx, rcx
jbe .L1
movzx edx, BYTE PTR [rsi+14+rax]
1.36.2 **SIMD strlen() implementation**

It has to be noted that the **SIMD** instructions can be inserted in C/C++ code via special macros\(^{171}\). For MSVC, some of them are located in the `intrin.h` file.

It is possible to implement the `strlen()` function\(^{172}\) using SIMD instructions that works 2-2.5 times faster than the common implementation. This function loads 16 characters into a XMM-register and check each

\(^{171}\)MSDN: MMX, SSE, and SSE2 Intrinsics
\(^{172}\)strlen() —standard C library function for calculating string length
against zero.\footnote{The example is based on source code from: \url{http://go.yurichev.com/17330}.}

```c
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=((unsigned int)str)&0xFFFFF0 == (unsigned int)str;
    if (str_is_aligned==false)
        return strlen (str);
    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;
    for (;;)
    {
        xmm1 = _mm_load_si128((&m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;
            BitScanForward(&pos, mask);
            len += (size_t)pos;
            break;
        }
        s += sizeof(__m128i);
        len += sizeof(__m128i);
    }
    return len;
}
```

Let's compile it in MSVC 2010 with /Ox option:

```
Listing 1.398: Optimizing MSVC 2010

    pos$75552 = -4 ; size = 4
    str$ = 8 ; size = 4
?strlen_sse2@YAIPLBD@Z PROC ; strlen_sse2
push    ebp
mov    ebp, esp
and    esp, -16 ; ffffff0H
mov    eax, DWORD PTR _str$[ebp]
sub    esp, 12 ; 000000cH
push    esi
mov    esi, eax
and    esi, -16 ; ffffff0H
xor    edx, edx
mov    ecx, eax
cmp    esi, eax
je     SHORT $LN4@strlen_sse
lea    edx, DWORD PTR [eax+1]
npad    3 ; align next label
$LL11@strlen_sse:
    mov    cl, BYTE PTR [eax]
    inc    eax
test    cl, cl
    jne     SHORT $LL11@strlen_sse
sub    eax, edx
pop    esi
mov    esp, ebp
pop    ebp
ret    0
$LN4@strlen_sse:
    movdqa    xmm1, XMMWORD PTR [eax]
    pxor    xmm0, xmm0
```

\footnote{The example is based on source code from: \url{http://go.yurichev.com/17330}.}
pcmpeqb  xmm1, xmm0
pmovmskb eax, xmm1
test  eax, eax
jne  SHORT $LN9@strlen_sse

$LL3@strlen_sse:
movdqa  xmm1, XMMWORD PTR [ecx+16]
add   ecx, 16 ; 00000010H
pcmpeqb  xmm1, xmm0
add   edx, 16 ; 00000010H
pmovmskb eax, xmm1
test  eax, eax
je  SHORT $LL3@strlen_sse

$LN9@strlen_sse:
bsf   eax, eax
mov   ecx, eax
mov   DWORD PTR _pos$75552[esp+16], eax
lea   eax, DWORD PTR [ecx+edx]
pop   esi
mov   esp, ebp
pop   ebp
ret  0

?strlen_sse2@YAIIPBD@Z ENDP ; strlen_sse2

How it works? First of all, we must understand goal of the function. It calculates C-string length, but we can use different terms: it's task is searching for zero byte, and then calculating its position relatively to string start.

First, we check if the str pointer is aligned on a 16-byte boundary. If not, we call the generic strlen() implementation.

Then, we load the next 16 bytes into the XMM1 register using MOVSDA.

An observant reader might ask, why can't MOVSDQU be used here since it can load data from the memory regardless pointer alignment?

Yes, it might be done in this way: if the pointer is aligned, load data using MOVSDA, if not —use the slower MOVSDQU.

But here we are may hit another caveat:

In the Windows NT line of OS (but not limited to it), memory is allocated by pages of 4 KiB (4096 bytes). Each win32-process has 4 GiB available, but in fact, only some parts of the address space are connected to real physical memory. If the process is accessing an absent memory block, an exception is to be raised. That's how VM works.

So, a function loading 16 bytes at once may step over the border of an allocated memory block. Let's say that the OS has allocated 8192 (0x2000) bytes at address 0x008c0000. Thus, the block is the bytes starting from address 0x008c0000 to 0x008c1fff inclusive.

After the block, that is, starting from address 0x008c2000 there is nothing at all, e.g. the OS not allocated any memory there. Any attempt to access memory starting from that address will raise an exception.

And let's consider the example in which the program is holding a string that contains 5 characters almost at the end of a block, and that is not a crime.

| 0x008c1f88 | 'h' |
| 0x008c1f89 | 'e' |
| 0x008c1f8a | 'l' |
| 0x008c1f8b | 'l' |
| 0x008c1f8c | 'o' |
| 0x008c1f8d | '\00' |
| 0x008c1ffe | random noise |
| 0x008c1fff | random noise |

So, in normal conditions the program calls strlen(), passing it a pointer to the string 'hello' placed in memory at address 0x008c1f8. strlen() reads one byte at a time until 0x008c1fffd, where there's a zero byte, and then it stops.

Now if we implement our own strlen() reading 16 bytes at once, starting at any address, aligned or not, MOVSDQU may attempt to load 16 bytes at once at address 0x008c1f8 up to 0x008c2008, and then an

174 wikipedia
exception will be raised. That situation is to be avoided, of course.

So then we'll work only with the addresses aligned on a 16 bytes boundary, which in combination with the knowledge that the OS' page size is usually aligned on a 16-byte boundary gives us some warranty that our function will not read from unallocated memory.

Let's get back to our function.

_mm_setzero_si128()—is a macro generating pxor xmm0, xmm0—it just clears the XMM0 register.

_mm_load_si128()—is a macro for MOVDQA, it just loads 16 bytes from the address into the XMM1 register.

_mm_cmpeq_epi8()—is a macro for PCMPEQB, an instruction that compares two XMM-registers bytewise. And if some byte is equals to the one in the other register, there will be 0xff at this point in the result or 0 if otherwise.

For example:

| XMM1: 0x11223344556677880000000000000000 | XMM0: 0x11ab3444007877881111111111111111 |

After the execution of pcmpeqb xmm1, xmm0, the XMM1 register contains:

| XMM1: 0xff0000ff0000ffff0000000000000000 |

In our case, this instruction compares each 16-byte block with a block of 16 zero-bytes, which has been set in the XMM0 register by pxor xmm0, xmm0.

The next macro is _mm_movemask_epi8()—that is the PMOVMSKB instruction.

It is very useful with PCMPEQB.

pmovmskb eax, xmm1

This instruction sets first EAX bit to 1 if the most significant bit of the first byte in XMM1 is 1. In other words, if the first byte of the XMM1 register is 0xff, then the first bit of EAX is to be 1, too.

If the second byte in the XMM1 register is 0xff, then the second bit in EAX is to be set to 1. In other words, the instruction is answering the question “which bytes in XMM1 has the most significant bit set, or greater than 0x7f”, and returns 16 bits in the EAX register. The other bits in the EAX register are to be cleared.

By the way, do not forget about this quirk of our algorithm. There might be 16 bytes in the input like:

\[
\begin{array}{cccccccccccccc}
\text{'h'} & \text{'e'} & \text{'l'} & \text{'l'} & \text{'o'} & 0 & \text{garbage} & 0 & \text{garbage}
\end{array}
\]

It is the 'hello' string, terminating zero, and some random noise in memory.

If we load these 16 bytes into XMM1 and compare them with the zeroed XMM0, we are getting something like 175:

| XMM1: 0x0000ff00000000000000ff0000000000 |

This means that the instruction found two zero bytes, and it is not surprising.

PMOVMSKB in our case will set EAX to 0b0010000000100000.

Obviously, our function must take only the first zero bit and ignore the rest.

The next instruction is BSF (Bit Scan Forward).

This instruction finds the first bit set to 1 and stores its position into the first operand.

EAX=0b0010000000100000

After the execution of bsf eax, eax, EAX contains 5, meaning 1 has been found at the 5th bit position (starting from zero).

MSVC has a macro for this instruction: _BitScanForward.

Now it is simple. If a zero byte has been found, its position is added to what we have already counted and now we have the return result.

Almost all.

175 An order from MSB to LSB is used here.
By the way, it is also has to be noted that the MSVC compiler emitted two loop bodies side by side, for optimization.

By the way, SSE 4.2 (that appeared in Intel Core i7) offers more instructions where these string manipulations might be even easier: [http://go.yurichev.com/17331](http://go.yurichev.com/17331)

1.37 64 bits

1.37.1 x86-64

It is a 64-bit extension to the x86 architecture.

From the reverse engineer’s perspective, the most important changes are:

- Almost all registers (except FPU and SIMD) were extended to 64 bits and got a R- prefix. 8 additional registers were added. Now GPR’s are: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

It is still possible to access the older register parts as usual. For example, it is possible to access the lower 32-bit part of the RAX register using EAX:

<table>
<thead>
<tr>
<th>Byte number:</th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX&lt;sup&gt;x64&lt;/sup&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The new R8-R15 registers also have their lower parts: R8D-R15D (lower 32-bit parts), R8W-R15W (lower 16-bit parts), R8L-R15L (lower 8-bit parts).

The number of SIMD registers was doubled from 8 to 16: XMM0-XMM15.

- In Win64, the function calling convention is slightly different, somewhat resembling fastcall ([6.1.3 on page 726](#)). The first 4 arguments are stored in the RCX, RDX, R8, R9 registers, the rest—in the stack. The caller function must also allocate 32 bytes so the callee may save there 4 first arguments and use these registers for its own needs. Short functions may use arguments just from registers, but larger ones may save their values on the stack.

System V AMD64 ABI (Linux, *BSD, Mac OS X) [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, System V Application Binary Interface. AMD64 Architecture Processor Supplement, (2013)]<sup>177</sup> also somewhat resembles fastcall, it uses 6 registers RDI, RSI, RDX, RCX, R8, R9 for the first 6 arguments. All the rest are passed via the stack.

See also the section on calling conventions ([6.1 on page 725](#)).

- The C/C++ `int` type is still 32-bit for compatibility.

- All pointers are 64-bit now.

Since now the number of registers is doubled, the compilers have more space for maneuvering called register allocation. For us this implies that the emitted code containing less number of local variables.

For example, the function that calculates the first S-box of the DES encryption algorithm processes 32/64/128/256 values at once (depending on DES_type type (uint32, uint64, SSE2 or AVX)) using the bitslice DES method (read more about this technique here ([1.36 on page 405](#))):

```c
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for any purpose,
 * so long as its origin is acknowledged.
 */
```

<sup>177</sup>Also available as [https://software.intel.com/sites/default/files/article/482129/mpx-linux64-abi.pdf](https://software.intel.com/sites/default/files/article/482129/mpx-linux64-abi.pdf)
ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (  
  DES_type a1,  
  DES_type a2,  
  DES_type a3,  
  DES_type a4,  
  DES_type a5,  
  DES_type a6,  
  DES_type *out1,  
  DES_type *out2,  
  DES_type *out3,  
  DES_type *out4  
) {
  DES_type x1, x2, x3, x4, x5, x6, x7, x8;
  DES_type x9, x10, x11, x12, x13, x14, x15, x16;
  DES_type x17, x18, x19, x20, x21, x22, x23, x24;
  DES_type x25, x26, x27, x28, x29, x30, x31, x32;
  DES_type x33, x34, x35, x36, x37, x38, x39, x40;
  DES_type x41, x42, x43, x44, x45, x46, x47, x48;
  DES_type x49, x50, x51, x52, x53, x54, x55, x56;

  x1 = a3 & ~a5;
  x2 = x1 ^ a4;
  x3 = a3 & ~a4;
  x4 = x3 | a5;
  x5 = a6 & x4;
  x6 = x2 ^ x5;
  x7 = a4 & ~a5;
  x8 = a3 ^ a4;
  x9 = a6 & ~x8;
  x10 = x7 ^ x9;
  x11 = a2 | x10;
  x12 = x6 ^ x11;
  x13 = a5 ^ x5;
  x14 = x13 & x8;
  x15 = a5 & ~a4;
  x16 = x3 ^ x14;
  x17 = a6 | x16;
  x18 = x15 ^ x17;
  x19 = a2 | x18;
  x20 = x14 ^ x19;
  x21 = a1 & x20;
  x22 = x12 ^ ~x21;
  *out2 ^= x22;
  x23 = x1 | x5;
  x24 = x23 ^ x8;
  x25 = x18 & ~x2;
  x26 = a2 & ~x25;
  x27 = x24 ^ x26;
  x28 = x6 | x7;
  x29 = x28 ^ x25;
  x30 = x9 ^ x24;
  x31 = x18 & ~x30;
  x32 = a2 & x31;
  x33 = x29 ^ x32;
  x34 = a1 & x33;
  x35 = x27 ^ x34;
  *out4 ^= x35;
  x36 = a3 & x28;
  x37 = x18 & ~x36;
  x38 = a2 | x3;
There are a lot of local variables. Of course, not all those going into the local stack. Let’s compile it with MSVC 2008 with /Ox option:

Listing 1.399: Optimizing MSVC 2008

PUBLIC _s1
; Function compile flags: /Ogtpy
_TEXT SEGMENT
   _x6$ = -20 ; size = 4
   _x3$ = -16 ; size = 4
   _x1$ = -12 ; size = 4
   _x8$ = -8 ; size = 4
   _x4$ = -4 ; size = 4
   _a1$ = 8 ; size = 4
   _a2$ = 12 ; size = 4
   _a3$ = 16 ; size = 4
   _x33$ = 20 ; size = 4
   _x7$ = 20 ; size = 4
   _a4$ = 20 ; size = 4
   _a5$ = 24 ; size = 4
tv326 = 28 ; size = 4
   _x36$ = 28 ; size = 4
   _x28$ = 28 ; size = 4
   _a6$ = 28 ; size = 4
   _out1$ = 32 ; size = 4
   _x24$ = 36 ; size = 4
   _out2$ = 36 ; size = 4
   _out3$ = 40 ; size = 4
   _out4$ = 44 ; size = 4
   _s1 PROC
   sub esp, 20 ; 00000014H
   mov edx, DWORD PTR _a5$[esp+16]
   push ebx
   mov ebx, DWORD PTR _a4$[esp+20]
   push ebp
   push esi
   mov esi, DWORD PTR _a3$[esp+28]
   push edi
   mov edi, ebx
   not edi
   mov ebp, edi
   and edi, DWORD PTR _a5$[esp+32]
   mov ecx, edx
   not ecx
   and ebp, esi
   mov eax, ecx
   and eax, esi
   and ecx, ebx
   _s1 ENDP
mov DWORD PTR _x1$[esp+36], eax
xor eax, ebx
mov esi, ebp
or esi, edx
mov DWORD PTR _x4$[esp+36], esi
and esi, DWORD PTR _a6$[esp+32]
mov DWORD PTR _x7$[esp+32], ecx
mov edx, esi
xor eax, ebx
mov esi, ebp
or esi, edx
mov DWORD PTR _x6$[esp+36], edx
mov edx, DWORD PTR _a3$[esp+32]
xor edx, esi
mov DWORD PTR _x4$[esp+36], esi
and edx, DWORD PTR _a6$[esp+32]
mov DWORD PTR _x7$[esp+32], eax
mov ecx, edx
mov edx, ebx
xor edx, ebp
or edx, DWORD PTR _a6$[esp+32]
not ecx
and ecx, DWORD PTR _a6$[esp+32]
xor edx, edi
mov edx, edx
or edx, DWORD PTR _a2$[esp+32]
mov DWORD PTR _x3$[esp+36], ebp
mov ebp, DWORD PTR _a2$[esp+32]
xor edi, ebx
and edi, DWORD PTR _a1$[esp+32]
mov ebx, ecx
xor ebx, DWORD PTR _x7$[esp+32]
not edi
or ebx, ebp
xor edi, ebx
mov ebx, edi
mov edi, DWORD PTR _out2$[esp+32]
xor ebx, DWORD PTR [edi]
not eax
xor ebx, DWORD PTR _x6$[esp+36]
and eax, edx
mov DWORD PTR [edi], ebx
mov ebx, DWORD PTR _x7$[esp+32]
or ebx, DWORD PTR _x6$[esp+36]
mov edi, esi
or edi, DWORD PTR _x1$[esp+36]
mov DWORD PTR _x28$[esp+32], ebx
xor edi, DWORD PTR _x8$[esp+36]
mov DWORD PTR _x24$[esp+32], edi
xor edi, ecx
not edi
and edi, edx
mov ebx, edi
and ebx, ebp
xor ebx, DWORD PTR _x28$[esp+32]
xor ebx, eax
not eax
mov DWORD PTR _x33$[esp+32], ebx
and ebx, DWORD PTR _a1$[esp+32]
and eax, ebp
xor eax, ebx
mov ebx, DWORD PTR _out4$[esp+32]
xor eax, DWORD PTR [ebx]
xor eax, DWORD PTR _x24$[esp+32]
mov DWORD PTR [ebx], eax
mov eax, DWORD PTR _x28$[esp+32]
and eax, DWORD PTR _a3$[esp+32]
mov ebx, DWORD PTR _x3$[esp+36]
or edi, DWORD PTR _a3$[esp+32]
mov DWORD PTR _x36$[esp+32], eax
not eax
and eax, edx
or ebx, ebp
xor ebx, eax
not eax
and eax, DWORD PTR _x24$[esp+32]
not ebp
or eax, DWORD PTR _x3$[esp+36]
not esi
and ebp, eax
or eax, edx
xor eax, DWORD PTR _a5$[esp+32]
mov edx, DWORD PTR _x36$[esp+32]
ox edx, DWORD PTR _x4$[esp+36]
xor ebp, edi
mov edi, DWORD PTR _out1$[esp+32]
not eax
and eax, DWORD PTR _a2$[esp+32]
not ebp
and ebp, DWORD PTR _a1$[esp+32]
and edx, esi
xor eax, edx
or eax, DWORD PTR _a1$[esp+32]
not ebp
xor ebp, DWORD PTR [edi]
not ecx
and ecx, DWORD PTR _x33$[esp+32]
ox ebp, ebx
not eax
mov DWORD PTR [edi], ebp
xor eax, ecx
mov ecx, DWORD PTR _out3$[esp+32]
xor eax, DWORD PTR [ecx]
pop edi
pop esi
xor eax, ebx
pop ebp
mov DWORD PTR [ecx], eax
pop ebx
add esp, 20
ret 0
_s1  ENDP

5 variables were allocated in the local stack by the compiler.

Now let's try the same thing in the 64-bit version of MSVC 2008:

Listing 1.400: Optimizing MSVC 2008

a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1  PROC
$LN3:
  mov  QWORD PTR [rsp+24], rbx
  mov  QWORD PTR [rsp+32], rbp
  mov  QWORD PTR [rsp+16], rdx
  mov  QWORD PTR [rsp+8], rcx
  push rsi
  push rdi
  push r12
  push r13
  push r14
  push r15
_s1  ENDP
mov r15, QWORD PTR a5$[rsp]
mov rcx, QWORD PTR a6$[rsp]
mov rbp, r8
mov r10, r9
mov rax, r15
mov rdx, rbp
not rax
xor rdx, r9
not r10
mov r11, rax
and rax, r9
mov rsi, r10
mov QWORD PTR x36$1$[rsp], rax
and r11, r8
and rsi, r8
and r10, r15
mov r13, rdx
mov rbx, r11
xor rbx, r9
mov r9, QWORD PTR a2$[rsp]
mov r12, rsi
or r12, r15
not r13
and r13, rcx
mov r14, r12
and r14, rcx
mov rax, r14
mov r8, r14
xor r8, rbx
xor rax, r15
not rbx
and rax, rdx
mov rdi, rax
or rdi, rsi
or rdi, rcx
xor rdi, r10
and rbx, rdi
mov rcx, rdi
or rcx, r9
xor rcx, rax
mov rax, r13
xor rax, QWORD PTR x36$1$[rsp]
and rcx, QWORD PTR a1$[rsp]
or rax, r9
not rcx
xor rcx, rax
mov rax, QWORD PTR out2$[rsp]
xor rcx, QWORD PTR [rax]
xor rcx, r8
mov QWORD PTR [rax], rcx
mov rax, QWORD PTR x36$1$[rsp]
mov rcx, r14
or rax, r8
or rcx, r11
mov r11, r9
xor rcx, rdx
mov QWORD PTR x36$1$[rsp], rax
mov r8, rsi
mov rdx, rcx
xor rdx, r13
not rdx
and rdx, rdi
mov r10, rdx
and r10, r9
xor r10, rax
xor r10, rbx
not rbx
and rbx, r9
mov rax, r10
and rax, QWORD PTR a1$[rsp]
Nothing was allocated in the local stack by the compiler, x36 is synonym for a5.
By the way, there are CPUs with much more GPR's, e.g. Itanium (128 registers).

## 1.37.2 ARM

64-bit instructions appeared in ARMv8.

## 1.37.3 Float point numbers

How floating point numbers are processed in x86-64 is explained here: 1.38 on the next page.
1.37.4 64-bit architecture criticism

Some people has irritation sometimes: now one needs twice as much memory for storing pointers, including cache memory, despite the fact that x64 CPUs can address only 48 bits of external RAM.

Pointers have gone out of favor to the point now where I had to flame about it because on my 64-bit computer that I have here, if I really care about using the capability of my machine I find that I’d better not use pointers because I have a machine that has 64-bit registers but it only has 2 gigabytes of RAM. So a pointer never has more than 32 significant bits to it. But every time I use a pointer it’s costing me 64 bits and that doubles the size of my data structure. Worse, it goes into the cache and half of my cache is gone and that costs cash—cache is expensive.

So if I’m really trying to push the envelope now, I have to use arrays instead of pointers. I make complicated macros so that it looks like I’m using pointers, but I’m not really.

( Donald Knuth in “Coders at Work: Reflections on the Craft of Programming “. )

Some people make their own memory allocators. It’s interesting to know about CryptoMiniSat case. This program rarely uses more than 4GiB of RAM, but it uses pointers heavily. So it requires less memory on 32-bit architecture than on 64-bit one. To mitigate this problem, author made his own allocator (in clauseallocator.(h|cpp) files), which allows to have access to allocated memory using 32-bit identifiers instead of 64-bit pointers.

1.38 Working with floating point numbers using SIMD

Of course, the FPU has remained in x86-compatible processors when the SIMD extensions were added. The SIMD extensions (SSE2) offer an easier way to work with floating-point numbers. The number format remains the same (IEEE 754).

So, modern compilers (including those generating for x86-64) usually use SIMD instructions instead of FPU ones.

It can be said that it’s good news, because it’s easier to work with them.

We are going to reuse the examples from the FPU section here: 1.25 on page 217.

1.38.1 Simple example

```c
#include <stdio.h>

double f (double a, double b)
{
    return a/3.14 + b*4.1;
};

int main()
{
    printf ("%f\n", f(1.2, 3.4));
};
```

x64

Listing 1.401: Optimizing MSVC 2012 x64

```
__real@4010666666666666 DQ 040106666666666666r ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr ; 3.14

a$ = 8
b$ = 16
f
  PROC
  divsd xmm0, QWORD PTR __real@40091eb851eb851f
  mulsd xmm1, QWORD PTR __real@4010666666666666
  addsd xmm0, xmm1
d3eb6d0c
```

\(^{178}\)https://github.com/msoos/cryptominisat/
The input floating point values are passed in the XMM0-XMM3 registers, all the rest—via the stack. \(^{179}\)

\(a\) is passed in XMM0, \(b\)—via XMM1.

The XMM-registers are 128-bit (as we know from the section about \textit{SIMD: 1.36 on page 404}), but the \textit{double} values are 64 bit, so only lower register half is used.

\textsc{DIVSD} is an SSE-instruction that stands for “Divide Scalar Double-Precision Floating-Point Values”, it just divides one value of type \textit{double} by another, stored in the lower halves of operands.

The constants are encoded by compiler in IEEE 754 format.

\textsc{MULSD} and \textsc{ADDSD} work just as the same, but do multiplication and addition.

The result of the function’s execution in type \textit{double} is left in the in XMM0 register.

That is how non-optimizing MSVC works:

\begin{lstlisting}[language=assembly]
Listing 1.402: MSVC 2012 x64

__real@40106666666666666 DQ 04010666666666666
__real@40091eb851eb851f DQ 040091eb851eb851fr

a$ = 8
b$ = 16

PROC

movsdx QWORD PTR [rsp+16], xmm1
movsdx QWORD PTR [rsp+8], xmm0
movsdx xmm0, QWORD PTR a$[rsp]
divsd xmm0, QWORD PTR __real@40091eb851eb851f
movsdx xmm1, QWORD PTR b$[rsp]
mulsd xmm1, QWORD PTR __real@4010666666666666
addsd xmm0, xmm1
ret

f

ENDP
\end{lstlisting}

Slightly redundant. The input arguments are saved in the “shadow space” (1.14.2 on page 100), but only their lower register halves, i.e., only 64-bit values of type \textit{double}. GCC produces the same code.

\textbf{x86}

Let’s also compile this example for x86. Despite the fact it’s generating for x86, MSVC 2012 uses SSE2 instructions:

\begin{lstlisting}[language=assembly]
Listing 1.403: Non-optimizing MSVC 2012 x86

tv70 = -8 ; size = 8
_a$ = 8 ; size = 8
_b$ = 16 ; size = 8

PROC

push ebp
mov ebp, esp
sub esp, 8
movsdx xmm0, QWORD PTR _a$[ebp]
divsd xmm0, QWORD PTR __real@40091eb851eb851f
movsdx xmm1, QWORD PTR _b$[ebp]
mulsd xmm1, QWORD PTR __real@4010666666666666
addsd xmm0, xmm1
fld QWORD PTR tv70[ebp], xmm0
mov QWORD PTR tv70[ebp]
ret

f

ENDP
\end{lstlisting}

\(^{179}\)\textit{MSDN: Parameter Passing}
It's almost the same code, however, there are some differences related to calling conventions: 1) the arguments are passed not in XMM registers, but in the stack, like in the FPU examples (1.25 on page 217); 2) the result of the function is returned in ST(0) — in order to do so, it's copied (through local variable tv) from one of the XMM registers to ST(0).
Let's try the optimized example in OllyDbg:

Figure 1.114: OllyDbg: MOVSD loads the value of $a$ into XMM1
Figure 1.115: OllyDbg: DIVSD calculated quotient and stored it in XMM1
Figure 1.116: OllyDbg: `MULSD` calculated **product** and stored it in XMM0
Figure 1.117: OllyDbg: ADDSD adds value in XMM0 to XMM1
We see that OllyDbg shows the XMM registers as pairs of double numbers, but only the lower part is used. Apparently, OllyDbg shows them in that format because the SSE2 instructions (suffixed with -SD) are executed right now.

But of course, it’s possible to switch the register format and to see their contents as 4 float-numbers or just as 16 bytes.
1.38.2 Passing floating point number via arguments

```c
#include <math.h>
#include <stdio.h>

int main ()
{
    printf ("32.01 ^ 1.54 = %lf\n", pow (32.01, 1.54));
    return 0;
}
```

They are passed in the lower halves of the XMM0-XMM3 registers.

Listing 1.405: Optimizing MSVC 2012 x64

```
SG1354 DB  '32.01 ^ 1.54 = %lf', 0AH, 00H
__real@40400147ae147ae1 DQ 040400147ae147ae1r ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r ; 1.54

main PROC
sub rsp, 40
    movsd xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
movsd xmm0, QWORD PTR __real@40400147ae147ae1
    call pow
    lea rcx, OFFSET FLAT:SG1354
movaps xmm1, xmm0
movd rdx, xmm1
call printf
xor eax, eax
    add rsp, 40
ret 0

main ENDP
```

There is no MOVSDX instruction in Intel and AMD manuals (12.1.4 on page 999), there it is called just MOVSD. So there are two instructions sharing the same name in x86 (about the other see: 1.6 on page 1015). Apparently, Microsoft developers wanted to get rid of the mess, so they renamed it to MOVSDX. It just loads a value into the lower half of a XMM register.

`pow()` takes arguments from XMM0 and XMM1, and returns result in XMM0. It is then moved to RDX for `printf()`. Why? Maybe because `printf()`—is a variable arguments function?

Listing 1.406: Optimizing GCC 4.4.6 x64

```
.LC2:
    .string "32.01 ^ 1.54 = %lf\n"

main:
    sub rsp, 8
    movsd xmm1, QWORD PTR .LC0[rip]
movsd xmm0, QWORD PTR .LC1[rip]
call pow
    ; result is now in XMM0
mov edi, OFFSET FLAT:.LC2
mov eax, 1 ; number of vector registers passed
call printf
xor eax, eax
    add rsp, 8
ret

.LC0:
    .long 171798692
    .long 1073259479

.LC1:
    .long 2920577761
    .long 1077936455
```

GCC generates clearer output. The value for `printf()` is passed in XMM0. By the way, here is a case when 1 is written into EAX for `printf()`—this implies that one argument will be passed in vector registers, just as the standard requires [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, System V Application Binary Interface. AMD64 Architecture Processor Supplement, (2013)]

### 1.38.3 Comparison example

```c
#include <stdio.h>

double d_max (double a, double b)
{
    if (a>b)
        return a;
    return b;
};

int main()
{
    printf ("%.1f\n", d_max (1.2, 3.4));
    printf ("%.1f\n", d_max (5.6, -4));
};
```

x64

Listing 1.407: Optimizing MSVC 2012 x64

```assembly
a$ = 8
b$ = 16
d_max PROC
    comisd xmm0, xmm1
    ja SHORT $LN2@d_max
    movaps xmm0, xmm1
$LN2@d_max:
    fatret 0
d_max ENDP
```

Optimizing MSVC generates a code very easy to understand.

COMISD is “Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS”. Essentially, that is what it does.

Non-optimizing MSVC generates more redundant code, but it is still not hard to understand:

Listing 1.408: MSVC 2012 x64

```assembly
a$ = 8
b$ = 16
d_max PROC
    movsd xmm0, QWORD PTR [rsp+16], xmm1
    movsd xmm0, QWORD PTR [rsp+8], xmm0
    movsd xmm0, QWORD PTR a$[rsp]
    comisd xmm0, QWORD PTR b$[rsp]
    jbe SHORT $LN1@d_max
    movsd xmm0, QWORD PTR a$[rsp]
    jmp SHORT $LN2@d_max
$LN1@d_max:
    movsd xmm0, QWORD PTR a$[rsp]
$LN2@d_max:
    fatret 0
d_max ENDP
```

However, GCC 4.4.6 did more optimizations and used the MAXSD (“Return Maximum Scalar Double-Precision Floating-Point Value”) instruction, which just choose the maximum value!

Listing 1.409: Optimizing GCC 4.4.6 x64

```assembly
d_max:
    maxsd xmm0, xmm1
    ret
```

434
Let's compile this example in MSVC 2012 with optimization turned on:

Listing 1.410: Optimizing MSVC 2012 x86

```
;a$ = 8 ; size = 8
_b$ = 16 ; size = 8
_d_max PROC
  movsd xmm0, QWORD PTR _a$[esp-4]
  comisd xmm0, QWORD PTR _b$[esp-4]
  jbe SHORT $LN1@d_max
  fld QWORD PTR _a$[esp-4]
  ret 0
$LN1@d_max:
  fld QWORD PTR _b$[esp-4]
  ret 0
_d_max ENDP
```

Almost the same, but the values of a and b are taken from the stack and the function result is left in ST(0).

If we load this example in OllyDbg, we can see how the COMISD instruction compares values and sets/clears the CF and PF flags:

Figure 1.119: OllyDbg: COMISD changed CF and PF flags

1.38.4 Calculating machine epsilon: x64 and SIMD

Let's revisit the “calculating machine epsilon” example for double listing. 1.32.2.

Now we compile it for x64:

Listing 1.411: Optimizing MSVC 2012 x64
There is no way to add 1 to a value in 128-bit XMM register, so it must be placed into memory.

There is, however, the ADDSD instruction (Add Scalar Double-Precision Floating-Point Values) which can add a value to the lowest 64-bit half of a XMM register while ignoring the higher one, but MSVC 2012 probably is not that good yet. Nevertheless, the value is then reloaded to a XMM register and subtraction occurs. SUBSD is “Subtract Scalar Double-Precision Floating-Point Values”, i.e., it operates on the lower 64-bit part of 128-bit XMM register. The result is returned in the XMM0 register.

### 1.38.5 Pseudo-random number generator example revisited

Let’s revisit “pseudo-random number generator example” example listing.1.32.1.

If we compile this in MSVC 2012, it will use the SIMD instructions for the FPU.

Listing 1.412: Optimizing MSVC 2012

```assembly
__real@3f800000 DD 03f800000r ; 1
tv128 = -.4
_tmp$ = -.4
?float_rand@YAMXZ PROC
    push ecx
    call ?my_rand@YAIXZ
    ; EAX=pseudorandom value
    and eax, 8388607 ; 007ffffffH
    or eax, 1065353216 ; 3f800000H
    ; EAX=pseudorandom value & 0x007ffffff | 0x3f800000
    ; store it into local stack:
    mov DWORD PTR _tmp$[esp+4], eax
    ; reload it as float point number:
    movss xmm0, DWORD PTR _tmp$[esp+4]
    ; subtract 1.0:
    subss xmm0, DWORD PTR __real@3f800000
    ; move value to ST0 by placing it in temporary variable...
    movss DWORD PTR tv128[esp+4], xmm0
    ; ... and reloading it into ST0:
    fld DWORD PTR tv128[esp+4]
    pop ecx
    ret 0
?float_rand@YAMXZ ENDP
```

All instructions have the -SS suffix, which stands for “Scalar Single”.

“Scalar” implies that only one value is stored in the register.

“Single”\(^{182}\) stands for float data type.

### 1.38.6 Summary

Only the lower half of XMM registers is used in all examples here, to store number in IEEE 754 format.

Essentially, all instructions prefixed by -SD (“Scalar Double-Precision”)—are instructions working with floating point numbers in IEEE 754 format, stored in the lower 64-bit half of a XMM register.

And it is easier than in the FPU, probably because the SIMD extensions were evolved in a less chaotic way than the FPU ones in the past. The stack register model is not used.

\(^{181}\)As an exercise, you may try to rework this code to eliminate the usage of the local stack.

\(^{182}\)I.e., single precision.
If you would try to replace `double` with `float` in these examples, the same instructions will be used, but prefixed with `-SS` (“Scalar Single-Precision”), for example, MOVSS, COMISS, ADDSS, etc.

“Scalar” implies that the SIMD register containing only one value instead of several.

Instructions working with several values in a register simultaneously have “Packed” in their name.

Needless to say, the SSE2 instructions work with 64-bit IEEE 754 numbers (`double`), while the internal representation of the floating-point numbers in FPU is 80-bit numbers.

Hence, the FPU may produce less round-off errors and as a consequence, FPU may give more precise calculation results.

1.39 ARM-specific details

1.39.1 Number sign (#) before number

The Keil compiler, IDA and objdump precede all numbers with the “#” number sign, for example: listing.1.22.1.

But when GCC 4.9 generates assembly language output, it doesn’t, for example: listing.3.18.

The ARM listings in this book are somewhat mixed.

It’s hard to say, which method is right. Supposedly, one has to obey the rules accepted in environment he/she works in.

1.39.2 Addressing modes

This instruction is possible in ARM64:

```
ldr x0, [x29,24]
```

This means add 24 to the value in X29 and load the value from this address.

Please note that 24 is inside the brackets. The meaning is different if the number is outside the brackets:

```
ldr w4, [x1],28
```

This means load the value at the address in X1, then add 28 to X1.

ARM allows you to add or subtract a constant to/from the address used for loading.

And it’s possible to do that both before and after loading.

There is no such addressing mode in x86, but it is present in some other processors, even on PDP-11.

There is a legend that the pre-increment, post-increment, pre-decrement and post-decrement modes in PDP-11,

were “guilty” for the appearance of such C language (which developed on PDP-11) constructs as *ptr++, *++ptr, *ptr--, *--ptr.

By the way, this is one of the hard to memorize C features. This is how it is:

<table>
<thead>
<tr>
<th>C term</th>
<th>ARM term</th>
<th>C statement</th>
<th>how it works</th>
</tr>
</thead>
<tbody>
<tr>
<td>Post-increment</td>
<td>post-indexed addressing</td>
<td>*ptr++</td>
<td>use *ptr value, then increment ptr pointer</td>
</tr>
<tr>
<td>Post-decrement</td>
<td>post-indexed addressing</td>
<td>*ptr--</td>
<td>use *ptr value, then decrement ptr pointer</td>
</tr>
<tr>
<td>Pre-increment</td>
<td>pre-indexed addressing</td>
<td>*++ptr</td>
<td>increment ptr pointer, then use *ptr value</td>
</tr>
<tr>
<td>Pre-decrement</td>
<td>pre-indexed addressing</td>
<td>*--ptr</td>
<td>decrement ptr pointer, then use *ptr value</td>
</tr>
</tbody>
</table>
Pre-indexing is marked with an exclamation mark in the ARM assembly language. For example, see line 2 in listing 1.29.

Dennis Ritchie (one of the creators of the C language) mentioned that it presumably was invented by Ken Thompson (another C creator) because this processor feature was present in PDP-7. [Dennis M. Ritchie, *The development of the C language*, (1993)]

Thus, C language compilers may use it, if it is present on the target processor.

That’s very convenient for array processing.

### 1.39.3 Loading a constant into a register

#### 32-bit ARM

As we already know, all instructions have a length of 4 bytes in ARM mode and 2 bytes in Thumb mode. Then how can we load a 32-bit value into a register, if it’s not possible to encode it in one instruction?

Let’s try:

```c
unsigned int f()
{
    return 0x12345678;
};
```

Listing 1.413: GCC 4.6.3 -O3 ARM mode

```assembly
f:
    ldr    r0, .L2
    bx     lr
.L2:
    .word  305419896 ; 0x12345678
```

So, the 0x12345678 value is just stored aside in memory and loaded if needed.

But it’s possible to get rid of the additional memory access.

Listing 1.414: GCC 4.6.3 -O3 -march=armv7-a (ARM mode)

```assembly
movw   r0, #22136 ; 0x5678
movt   r0, #4660 ; 0x1234
bx     lr
```

We see that the value is loaded into the register by parts, the lower part first (using MOVW), then the higher (using MOVT).

This implies that 2 instructions are necessary in ARM mode for loading a 32-bit value into a register. It’s not a real problem, because in fact there are not many constants in real code (except of 0 and 1).

Does it mean that the two-instruction version is slower than one-instruction version?

Doubtfully. Most likely, modern ARM processors are able to detect such sequences and execute them fast.

On the other hand, IDA is able to detect such patterns in the code and disassembles this function as:

```assembly
MOV   R0, 0x12345678
BX    LR
```

#### ARM64

```c
uint64_t f()
{
    return 0x12345678ABCDEF01;
};
```

[183](http://yurichev.com/mirrors/C/c_dmr_postincrement.txt) Also available as [http://go.yurichev.com/17264](http://go.yurichev.com/17264)
MOVK stands for “MOV Keep”, i.e., it writes a 16-bit value into the register, not touching the rest of the bits. The LSL suffix shifts left the value by 16, 32 and 48 bits at each step. The shifting is done before loading. This implies that 4 instructions are necessary to load a 64-bit value into a register.

**Storing floating-point number into register**

It’s possible to store a floating-point number into a D-register using only one instruction.

For example:

```c
double a()
{
    return 1.5;
}
```

The number 1.5 was indeed encoded in a 32-bit instruction. But how?

In ARM64, there are 8 bits in the FMOV instruction for encoding some floating-point numbers. The algorithm is called VFPExpandImm() in [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)]185. This is also called minifloat186.

We can try different values: the compiler is able to encode 30.0 and 31.0, but it couldn’t encode 32.0, as 8 bytes have to be allocated for this number in the IEEE 754 format:

```c
double a()
{
    return 32;
}
```

**1.39.4 Relocs in ARM64**

As we know, there are 4-byte instructions in ARM64, so it is impossible to write a large number into a register using a single instruction.

Nevertheless, an executable image can be loaded at any random address in memory, so that’s why relocations exist. Read more about them (in relation to Win32 PE): 6.5.2 on page 751.

The address is formed using the ADRP and ADD instruction pair in ARM64.

The first loads a 4KiB-page address and the second one adds the remainder. Let's compile the example from “Hello, world!” (listing.1.11) in GCC (Linaro) 4.9 under win32:

```c
a:
    ldr     d0, .LC0
    ret
.LC0:
    .word  0
    .word  1077936128
```


186[wikipedia](https://en.wikipedia.org/wiki/Minifloat)
0000000000000000 <main>:
  0: a9bf7bfd stp x29, x30, [sp,#-16]!
  4: 910003fd mov x29, sp
  8: 90000000 adrp x0, 0 <main>
 c: 91000000 add x0, x0, #0x0
10: 94000000 bl 0 <printf>
14: 52800000 mov w0, #0x0 // #0
18: a8c17bfd ldp x29, x30, [sp],#16
1c: d65f03c0 ret

So there are 3 relocs in this object file.

- The first one takes the page address, cuts the lowest 12 bits and writes the remaining high 21 bits to the ADRP instruction’s bit fields. This is because we don’t need to encode the low 12 bits, and the ADRP instruction has space only for 21 bits.

- The second one puts the 12 bits of the address relative to the page start into the ADD instruction’s bit fields.

- The last, 26-bit one, is applied to the instruction at address 0x10 where the jump to the printf() function is.

All ARM64 (and in ARM in ARM mode) instruction addresses have zeros in the two lowest bits (because all instructions have a size of 4 bytes), so one have to encode only the highest 26 bits of 28-bit address space (∓128MB).

There are no such relocs in the executable file: because it’s known where the “Hello!” string is located, in which page, and the address of puts() is also known.

So there are values set already in the ADRP, ADD and BL instructions (the linker has written them while linking):

As an example, let’s try to disassemble the BL instruction manually.

0x97ffffa0 is 0b1001011111111111111111110100000. According to [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)C5.6.26], imm26 is the last 26 bits:

\[imm26 = 0b11111111111111111110100000\]. It is 0x3FFFFA0, but the MSB is 1, so the number is negative, and we can convert it manually to convenient form for us. By the rules of negation (2.2 on page 449), just invert...
all bits: (it is \(0b1011111=0x5F\)), and add 1 (\(0x5F+1=0x60\)). So the number in signed form is \(-0x60\). Let’s multiply \(-0x60\) by 4 (because address stored in opcode is divided by 4): it is \(-0x180\). Now let’s calculate destination address: \(0x4005a0 + (-0x180) = 0x400420\) (please note: we consider the address of the BL instruction, not the current value of PC, which may be different!). So the destination address is \(0x400420\).


## 1.40 MIPS-specific details

### 1.40.1 Loading a 32-bit constant into register

```c
unsigned int f()
{
    return 0x12345678;
};
```

All instructions in MIPS, just like ARM, have a size of 32-bit, so it’s not possible to embed a 32-bit constant into one instruction.

So one have to use at least two instructions: the first loads the high part of the 32-bit number and the second one applies an OR operation, which effectively sets the low 16-bit part of the target register:

```
Listing 1.420: GCC 4.4.5 -O3 (assembly output)

li $2,305397760  # 0x12340000
j $31
ori $2,$2,0x5678 ; branch delay slot
```

IDA is fully aware of such frequently encountered code patterns, so, for convenience it shows the last ORI instruction as the LI pseudo instruction, which allegedly loads a full 32-bit number into the $V0 register.

```
Listing 1.421: GCC 4.4.5 -O3 (IDA)

lui $v0, 0x1234
jr $ra
li $v0, 0x12345678 ; branch delay slot
```

The GCC assembly output has the LI pseudo instruction, but in fact, LUI (“Load Upper Immediate”) is there, which stores a 16-bit value into the high part of the register.

Let’s see in objdump output:

```
Listing 1.422: objdump output

00000000 <f>:
  0:  3c021234  lui  v0,0x1234
  4:  03e00000  jr   ra
  8:  34425678  ori  v0,v0,0x5678
```

### Loading a 32-bit global variable into register

```c
unsigned int global_var=0x12345678;

unsigned int f2()
{
    return global_var;
};
```

This is slightly different: LUI loads upper 16-bit from `global_var` into $2 (or $V0) and then LW loads lower 16-bits summing it with the contents of $2:

```
Listing 1.423: GCC 4.4.5 -O3 (assembly output)

f2:
    lui $2,%hi(global_var)
    lw  $2,%lo(global_var)($2)
```

\[^{187}\]Also available as [http://go.yurichev.com/17288](http://go.yurichev.com/17288)
IDA is fully aware of often used LUI/LW instruction pair, so it coalesces both into a single LW instruction:

```
Listing 1.424: GCC 4.4.5 -O3 (IDA)

_f2:
        lw  $v0, global_var
        jr  $ra
        or  $at, $zero ; branch delay slot
...
.data
.globl global_var
global_var: .word 0x12345678     # DATA XREF: _f2
```

objdump’s output is the same as GCC’s assembly output. Let’s also dump relocations of the object file:

```
Listing 1.425: objdump

objdump -D filename.o
...
00000000 <f2>:
  c: 3c020000 lui v0,0x0
  10: 8c420000 lw v0,0(v0)
  14: 03e00000 jr ra
  18: 00200825 move at,at ; branch delay slot
  1c: 00200825 move at,at

Disassembly of section .data:

00000000 <global_var>:
  0: 12345678 beq s1,s4,159e4 <f2+0x159d8>
...

objdump -r filename.o
...

RELOCATION RECORDS FOR [.text]:
OFFSET TYPE VALUE
00000000c R_MIPS_HI16   global_var
00000010 R_MIPS_LO16   global_var
...
```

We can see that address of global_var is to be written right into LUI and LW instructions during executable file loading: high 16-bit part of global_var goes into the first one (LUI), lower 16-bit part goes into the second one (LW).

### 1.40.2 Further reading about MIPS

Chapter 2
Important fundamentals

2.1 Integral datatypes

Integral datatype is a type for a value which can be converted to number. These are numbers, enumerations, booleans.

2.1.1 Bit

Obvious usage for bits are boolean values: 0 for false and 1 for true.

Set of booleans can be packed into word: there will be 32 booleans in 32-bit word, etc. This way is called bitmap or bitfield.

But it has obvious overhead: a bit jiggling, isolating, etc. While using word (or int type) for boolean variable is not economic, but highly efficient.

In C/C++ environment, 0 is for false and any non-zero value is for true. For example:

```c
if (1234)
    printf ("this will always be executed\n");
else
    printf ("this will never\n");
```

This is popular way of enumerating characters in a C-string:

```c
char *input=...;
while(*input) // execute body if *input character is non-zero
{
    // do something with *input
    input++;
}
```

2.1.2 Nibble AKA nybble

AKA half-byte, tetrade. Equals to 4 bits.

All these terms are still in use today.

Binary-coded decimal (BCD\(^1\))

4-bit nibbles were used in 4-bit CPUs like legendary Intel 4004 (used in calculators).

It’s interesting to know that there was binary-coded decimal (BCD) way of representing decimal digit using 4 bits. Decimal 0 is represented as 0b0000, decimal 9 as 0b1001 and higher values are not used. Decimal 1234 is represented as 0x1234. Of course, this way is not economical.

Nevertheless, it has one advantage: decimal to BCD-packed number conversion and back is extremely easy. BCD-numbers can be added, subtracted, etc., but an additional correction is needed. x86 CPUs has

\(^1\)Binary-Coded Decimal
rare instructions for that: AAA/DAA (adjust after addition), AAS/DAS (adjust after subtraction), AAM (after multiplication), AAD (after division).

The need for CPUs to support BCD numbers is a reason why half-carry flag (on 8080/Z80) and auxiliary flag (AF on x86) are exist: this is carry-flag generated after proceeding of lower 4 bits. The flag is then used for adjustment instructions.

The fact of easy conversion had led to popularity of [Peter Abel, IBM PC assembly language and programming (1987)] book. But aside of this book, the author of these notes never seen BCD numbers in practice, except for magic numbers (5.6.1 on page 704), like when someone’s birthday is encoded like 0x19791011—this is indeed packed BCD number.

Surprisingly, the author found a use of BCD-encoded numbers in SAP software: https://yurichev.com/blog/SAP/. Some numbers, including prices, are encoded in BCD form in database. Perhaps, they used it to make it compatible with some ancient software/hardware?

BCD instructions in x86 were often used for other purposes, especially in undocumented ways, for example:

```
cmp al,10
sbb al,69h
das
```

This obscure code converts number in 0..15 range into ASCII character ‘0’..'9’, ‘A’..'F’.

### Z80

Z80 was clone of 8-bit Intel 8080 CPU, and because of space constraints, it has 4-bit ALU, i.e., each operation over two 8-bit numbers had to be proceeded in two steps. One side-effect of this was easy and natural generation of half-carry flag.

#### 2.1.3 Byte

Byte is primarily used for character storage. 8-bit bytes were not common as today. Punched tapes for teletypes had 5 and 6 possible holes, this is 5 or 6 bits for byte.

To emphasize the fact the byte has 8 bits, byte is sometimes called octet: at least fetchmail uses this terminology.

9-bit bytes used to exist in 36-bit architectures: 4 9-bit bytes would fit in a single word. Probably because of this fact, C/C++ standard tells that char has to have a room for at least 8 bits, but more bits are allowable.

For example, in the early C language manual2, we can find this:

```
char one byte character (PDP-11, IBM360: 8 bits; H6070: 9 bits)
```

By H6070 they probably meant Honeywell 6070, with 36-bit words.

### Standard ASCII table

7-bit ASCII table is standard, which has only 128 possible characters. Early E-Mail transport software were operating only on 7-bit ASCII codes, so a MIME3 standard needed to encode messages in non-Latin writing systems. 7-bit ASCII code was augmented by parity bit, resulting in 8 bits.

Data Encryption Standard (DES4) has a 56 bits key, this is 8 7-bit bytes, leaving a space to parity bit for each character.

There is no need to memorize whole ASCII table, but rather ranges. [0..0x1F] are control characters (non-printable). [0x20..0x7E] are printable ones. Codes starting at 0x80 are usually used for non-Latin writing systems and/or pseudographics.

Significant codes which will be easily memorized are: 0 (end of C-string, ‘\0’ in C/C++); 0xA or 10 (line feed, ‘\n’ in C/C++); 0xD or 13 (carriage return, ‘\r’ in C/C++).

0x20 (space) is also often memorized.

2https://yurichev.com/mirrors/C/bwk-tutor.html
3Multipurpose Internet Mail Extensions
4Data Encryption Standard
8-bit CPUs

x86 has capability to work with byte(s) on register level (because they are descendants of 8-bit 8080 CPU), RISC CPUs like ARM and MIPS—not.

2.1.4 Widechar

This is an attempt to support multi-lingual environment by extending byte to 16-bit. Most well-known example is Windows NT kernel and win32 functions with W suffix. This is why each Latin character in plain English text string is interleaved with zero byte. This encoding is called UCS-2 or UTF-16

Usually, `wchar_t` is synonym to 16-bit `short` data type.

2.1.5 Signed integer vs unsigned

Some may argue, why unsigned data types exist at first place, since any unsigned number can be represented as signed. Yes, but absence of sign bit in a value extends its range twice. Hence, signed byte has range of -128..127, and unsigned one: 0..255. Another benefit of using unsigned data types is self-documenting: you define a variable which can’t be assigned to negative values.

Unsigned data types are absent in Java, for which it’s criticized. It’s hard to implement cryptographical algorithms using boolean operations over signed data types.

Values like 0xFFFFFFFF (-1) are used often, mostly as error codes.

2.1.6 Word

Word word is somewhat ambiguous term and usually denotes a data type fitting in GPR. Bytes are practical for characters, but impractical for other arithmetical calculations.

Hence, many CPUs have GPRs with width of 16, 32 or 64 bits. Even 8-bit CPUs like 8080 and Z80 offer to work with 8-bit register pairs, each pair forming a 16-bit pseudoregister (BC, DE, HL, etc.). Z80 has some capability to work with register pairs, and this is, in a sense, some kind of 16-bit CPU emulation.

In general, if a CPU marketed as “n-bit CPU”, this usually means it has n-bit GPRs.

There was a time when hard disks and RAM modules were marketed as having n kilo-words instead of b kilobytes/megabytes.

For example, Apollo Guidance Computer has 2048 words of RAM. This was a 16-bit computer, so there was 4096 bytes of RAM.

TX-0 had 64K of 18-bit words of magnetic core memory, i.e., 64 kilo-words.

DECSYSTEM-2060 could have up to 4096 kilowords of solid state memory (i.e., hard disks, tapes, etc). This was 36-bit computer, so this is 18432 kilobytes or 18 megabytes.

Essentially, why do you need bytes if you have words? Mostly for text strings processing. Words can be used in almost any other situations.

---

`int` in C/C++ is almost always mapped to `word`. (Except of AMD64 architecture where `int` is still 32-bit one, perhaps, for the reason of better portability.)

`int` is 16-bit on PDP-11 and old MS-DOS compilers. `int` is 32-bit on VAX, on x86 starting at 80386, etc.

Even more than that, if type declaration for a variable is omitted in C/C++ program, `int` is used silently by default. Perhaps, this is inheritance of B programming language.

---

GPR is usually fastest container for variable, faster than packed bit, and sometimes even faster than byte (because there is no need to isolate a single bit/byte from GPR). Even if you use it as a container for loop counter in 0..99 range.

---

**Word** in assembly language is still 16-bit for x86, because it was so for 16-bit 8086. *Double word* is 32-bit, *quad word* is 64-bit. That’s why 16-bit words are declared using DW in x86 assembly, 32-bit ones using DD and 64-bit ones using DQ.

**Word** is 32-bit for ARM, MIPS, etc., 16-bit data types are called *half-word* there. Hence, *double word* on 32-bit RISC is 64-bit data type.

**GDB** has the following terminology: *halfword* for 16-bit, *word* for 32-bit and *giantword* for 64-bit.

16-bit C/C++ environment on PDP-11 and MS-DOS has *long* data type with width of 32 bits, perhaps, they meant *long word* or *long int*?

32-bit C/C++ environment has *long long* data type with width of 64 bits.

Now you see why the *word* word is ambiguous.

**Should I use int?**

Some people argue that *int* shouldn’t be used at all, because it ambiguity can lead to bugs. For example, well-known *lzhuf* library uses *int* at one point and everything works fine on 16-bit architecture. But if ported to architecture with 32-bit *int*, it can crash: [http://yurichev.com/blog/lzhuf/](http://yurichev.com/blog/lzhuf/).

Less ambiguous types are defined in *stdint.h* file: *uint8_t*, *uint16_t*, *uint32_t*, *uint64_t*, etc.

Some people like Donald E. Knuth proposed more sonorous words for these types: *byte/wyde/tetra-byte/octabyte*. But these names are less popular than clear terms with inclusion of *u* (*unsigned*) character and number right into the type name.

**Word-oriented computers**

Despite the ambiguity of the *word* term, modern computers are still word-oriented: RAM and all levels of cache are still organized by words, not by bytes. However, size in bytes is used in marketing.

Access to RAM/cache by address aligned by word boundary is often cheaper than non-aligned.

During data structures development, which are supposed to be fast and efficient, one should always take into consideration length of the *word* on the CPU to be executed on. Sometimes the compiler will do this for programmer, sometimes not.

**2.1.7 Address register**

For those who fostered on 32-bit and/or 64-bit x86, and/or RISC of 90s like ARM, MIPS, PowerPC, it’s natural that address bus has the same width as *GPR* or *word*. Nevertheless, width of address bus can be different on other architectures.

8-bit Z80 can address $2^{16}$ bytes, using 8-bit registers pairs or dedicated registers (*IX*, *IY*). *SP* and *PC* registers are also 16-bit ones.

Cray-1 supercomputer has 64-bit GPRs, but 24-bit address registers, so it can address $2^{24}$ (16 megawords or 128 megabytes). RAM was very expensive in 1970s, and a typical Cray had 1048576 (0x100000) words of RAM or 8MB. So why to allocate 64-bit register for address or pointer?

8086/8088 CPUs had a really weird addressing scheme: values of two 16-bit registers were summed in a weird manner resulting in a 20-bit address. Perhaps, this was some kind of toy-level virtualization (11.6 on page 986)? 8086 could run several programs (not simultaneously, though).

Early ARM1 has an interesting artifact:

Another interesting thing about the register file is the PC register is missing a few bits. Since the ARM1 uses 26-bit addresses, the top 6 bits are not used. Because all instructions are aligned on a 32-bit boundary, the bottom two address bits in the PC are always zero. These 8 bits are not only unused, they are omitted from the chip entirely.


Hence, it’s physically not possible to push a value with one of two last bits set into PC register. Nor it’s possible to set any bits in high 6 bits of PC.

---

x86-64 architecture has virtual 64-bit pointers/addresses, but internally, width of address bus is 48 bits (seems enough to address 256TB of RAM).

### 2.1.8 Numbers

**What are numbers used for?**

When you see some number(s) altering in a CPU register, you may be interested in what this number means. It’s an important skill for a reverse engineer to determine possible data type from a set of changing numbers.

#### Boolean

If the number is switching from 0 to 1 and back, most chances that this value has boolean data type.

#### Loop counter, array index

Variable increasing from 0, like: 0, 1, 2, 3...—a good chance this is a loop counter and/or array index.

#### Signed numbers

If you see a variable which holds very low numbers and sometimes very high numbers, like 0, 1, 2, 3, and 0xFFFFFFFF, 0xFFFFFFFFF, 0xFFFFFFFFD, there’s a good chance it is a signed variable in two’s complement form (2.2 on the following page), and last 3 numbers are -1, -2, -3.

#### 32-bit numbers

There are numbers so large, that there is even a special notation which exists to represent them (Knuth’s up-arrow notation). These numbers are so large so these are not practical for engineering, science and mathematics.

Almost all engineers and scientists are happy with IEEE 754 double precision floating point, which has maximal value around $1.8 \cdot 10^{308}$. (As a comparison, the number of atoms in the observable universe, is estimated to be between $4 \cdot 10^{79}$ and $4 \cdot 10^{81}$.)

In fact, upper bound in practical computing is much, much lower. In MS-DOS era 16-bit `int` was used almost for everything (array indices, loop counters), while 32-bit `long` was used rarely.

During advent of x86-64, it was decided for `int` to stay as 32 bit size integer, because, probably, usage of 64-bit `int` is even rarer.

I would say, 16-bit numbers in range 0..65535 are probably most used numbers in computing.

Given that, if you see unusually large 32-bit value like 0x87654321, this is a good chance this can be:

- this can still be a 16-bit number, but signed, between 0xFFFF8000 (-32768) and 0xFFFFFFFF (-1).
- address of memory cell (can be checked using memory map feature of debugger).
- packed bytes (can be checked visually).
- bit flags.
- something related to (amateur) cryptography.
- magic number (5.6.1 on page 704).
- IEEE 754 floating point number (can also be checked).

Almost same story for 64-bit values.

### ...so 16-bit `int` is enough for almost everything?

It’s interesting to note: in [Michael Abrash, *Graphics Programming Black Book*, 1997 chapter 13] we can find that there are plenty cases in which 16-bit variables are just enough. In a meantime, Michael Abrash has a pity that 80386 and 80486 CPUs has so little available registers, so he offers to put two 16-bit values into one 32-bit register and then to rotate it using ROR reg, 16 (on 80386 and later) (ROL reg, 16 will also work) or BSWAP (on 80486 and later) instruction.
That reminds us Z80 with alternate pack of registers (suffixed with apostrophe), to which CPU can switch (and then switch back) using EXX instruction.

**Size of buffer**

When a programmer needs to declare the size of some buffer, values in form of $2^x$ are usually used (512 bytes, 1024, etc.). Values in $2^x$ form are easily recognizable (1.28.5 on page 320) in decimal, hexadecimal and binary base.

But needless to say, programmers are still humans with their decimal culture. And somehow, in DBMS area, size of textual database fields is often chosen as $10^x$ number, like 100, 200. They just think “Okay, 100 is enough, wait, 200 will be better”. And they are right, of course.

Maximum width of VARCHAR2 data type in Oracle RDBMS is 4000 characters, not 4096.

There is nothing wrong with this, this is just a place where numbers like $10^x$ can be encountered.

**Address**

It’s always a good idea to keep in mind an approximate memory map of the process you currently debug. For example, many win32 executables started at 0x00401000, so an address like 0x00451230 is probably located inside executable section. You’ll see addresses like these in the EIP register.

Stack is usually located somewhere below.

Many debuggers are able to show the memory map of the debuggee, for example: 1.12.3 on page 79.

If a value is increasing by step 4 on 32-bit architecture or by step 8 on 64-bit one, this probably sliding address of some elements of array.

It’s important to know that win32 doesn’t use addresses below 0x10000, so if you see some number below this constant, this cannot be an address (see also: https://msdn.microsoft.com/en-us/library/ms810627.aspx).

Anyway, many debuggers can show you if the value in a register can be an address to something. OllyDbg can also show an ASCII string if the value is an address of it.

**Bit field**

If you see a value where one (or more) bit(s) are flipping from time to time like 0xABCD1234 → 0xABCD1434 and back, this is probably a bit field (or bitmap).

**Packed bytes**

When strcmp() or memcmp() copies a buffer, it loads/stores 4 (or 8) bytes simultaneously, so if a string containing “4321”, and it would be copied to another place, at one point you’ll see 0x31323334 value in some register. This is 4 packed bytes into a 32-bit value.

### 2.2 Signed number representations

There are several methods for representing signed numbers, but “two’s complement” is the most popular one in computers.

Here is a table for some byte values:
The difference between signed and unsigned numbers is that if we represent 0xFFFFFFFFFE and 0x000000002 as unsigned, then the first number (4294967294) is bigger than the second one (2). If we represent them both as signed, the first one becomes \(-2\), and it is smaller than the second (2). That is the reason why conditional jumps (1.18 on page 123) are present both for signed (e.g. JG, JL) and unsigned (JA, JB) operations.

For the sake of simplicity, this is what one needs to know:

- Numbers can be signed or unsigned.
- C/C++ signed types:
  - `int64_t` (-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807) (-9.2..9.2 quintillions) or 0x8000000000000000..0x7FFFFFFFFFFFFFFF),
  - `int` (-2,147,483,648..2,147,483,647 (-2.15..2.15Gb) or 0x80000000..0x7FFFFFFF),
  - `char` (-128..127 or 0x80..0x7F),
  - `ssize_t`.
- Unsigned:
  - `uint64_t` (0..18,446,744,073,709,551,615 (18 quintillions) or 0..0xFFFFFFFFFFFFFFFF),
  - `unsigned int` (0..4,294,967,295 (4.3Gb) or 0..0xFFFFFFFF),
  - `unsigned char` (0..255 or 0..0xFF),
  - `size_t`.
- Signed types have the sign in the **MSB**: 1 means “minus”, 0 means “plus”.
- Promoting to a larger data types is simple: 1.34.5 on page 401.
- Negation is simple: just invert all bits and add 1.

We can keep in mind that a number of inverse sign is located on the opposite side at the same proximity from zero. The addition of one is needed because zero is present in the middle.

- The addition and subtraction operations work well for both signed and unsigned values. But for multiplication and division operations, x86 has different instructions: IDIV/IMUL for signed and DIV/MUL for unsigned.
- Here are some more instructions that work with signed numbers: CBW/CWD/CWDE/CDQ/CDQE (.1.6 on page 1017), MOVsx (1.23.1 on page 200), SAR (.1.6 on page 1021).

A table of some negative and positive values (?? on page ??) looks like thermometer with Celsius scale. This is why addition and subtraction works equally well for both signed and unsigned numbers: if the first addend is represented as mark on thermometer, and one need to add a second addend, and it’s positive,
we just shift mark up on thermometer by the value of second addend. If the second addend is negative, then we shift mark down to absolute value of the second addend.

Addition of two negative numbers works as follows. For example, we need to add -2 and -3 using 16-bit registers. -2 and -3 is 0xfffe and 0xfffdf respectively. If we add these numbers as unsigned, we will get 0xffffe+0xfffffd=0x11fff. But we work on 16-bit registers, so the result is cut off, the first 1 is dropped, 0xffff is left, and this is -5. This works because -2 (or 0xfffe) can be represented using plain English like this: “2 lacks in this value up to maximal value in 16-bit register +1”. -3 can be represented as “...3 lacks in this value up to ...”. Maximal value of 16-bit register +1 is 0x10000. During addition of two numbers and cutting off by $2^{16}$ modulo, $2 + 3 = 5$ will be lacking.

2.2.1 Using IMUL over MUL

Example like listing.3.23.2 where two unsigned values are multiplied compiles into listing.3.23.2 where IMUL is used instead of MUL.

This is important property of both MUL and IMUL instructions. First of all, they both produce 64-bit value if two 32-bit values are multiplied, or 128-bit value if two 64-bit values are multiplied (biggest possible product in 32-bit environment is $0xffffffff*0xffffffff=0x100000001$). But C/C++ standards have no way to access higher half of result, and a product always has the same size as multiplicands. And both MUL and IMUL instructions works in the same way if higher half is ignored, i.e., they both generate the same lower half. This is important property of “two’s complement” way of representing signed numbers.

So C/C++ compiler can use any of these instructions.

But IMUL is more versatile than MUL because it can take any register(s) as source, while MUL requires one of multiplicands stored in AX/EAX/RAX register. Even more than that: MUL stores result in EDX:EAX pair in 32-bit environment, or RDX:RAX in 64-bit one, so it always calculates the whole result. On contrary, it’s possible to set a single destination register while using IMUL instead of pair, and then CPU will calculate only lower half, which works faster [see Torbom Granlund, Instruction latencies and throughput for AMD and Intel x86 processors]

Given than, C/C++ compilers may generate IMUL instruction more often then MUL.

Nevertheless, using compiler intrinsic, it’s still possible to do unsigned multiplication and get full result. This is sometimes called extended multiplication. MSVC has intrinsic for this called _emu8 and another one: _umul128⁹. GCC offer __int128 data type, and if 64-bit multiplicands are first promoted to 128-bit ones, then a product is stored into another __int128 value, then result is shifted by 64 bits right, you’ll get higher half of result.¹⁰

MulDiv() function in Windows

Windows has MulDiv() function ¹¹, fused multiply/divide function, it multiplies two 32-bit integers into intermediate 64-bit value and then divides it by a third 32-bit integer. It is easier than to use two compiler intrinsic, so Microsoft developers made a special function for it. And it seems, this is busy function, judging by its usage.

2.2.2 Couple of additions about two’s complement form

Exercise 2-1. Write a program to determine the ranges of char, short, int, and long variables, both signed and unsigned, by printing appropriate values from standard headers and by direct computation.


---

⁷http://yurichev.com/mirrors/x86-timing.pdf
⁸https://msdn.microsoft.com/en-us/library/d2s81x8t(v=vs.80).aspx
⁹https://msdn.microsoft.com/library/3daytyw%28v=vs.100%29.aspx
¹⁰Example: http://stackoverflow.com/a/13187798
Getting maximum number of some word

Maximum unsigned number is just a number where all bits are set: \texttt{0xFF....FF} (this is -1 if the \texttt{word} is treated as signed integer). So you take a \texttt{word}, set all bits and get the value:

```c
#include <stdio.h>
int main()
{
    unsigned int val=0; // change to "unsigned char" to get maximal value for the unsigned 8-bit byte
    // 0-1 will also work, or just -1
    printf ("%u
", val); // 

    This is 4294967295 for 32-bit integer.
```

Getting minimum number for some signed word

Minimum signed number is encoded as \texttt{0x80....00}, i.e., most significant bit is set, while others are cleared. Maximum signed number is encoded in the same way, but all bits are inverted: \texttt{0x7F....FF}.

Let’s shift a lone bit left until it disappears:

```c
#include <stdio.h>
int main()
{
    signed int val=1; // change to "signed char" to find values for signed byte
    while (val!=0)
    {
        printf ("%d %d
", val, -val);
        val=val<<1;
    }
};
```

Output is:

```plaintext
... 
536870912  -536870913 
1073741824  -1073741825 
-2147483648  2147483647
```

Two last numbers are minimum and maximum signed 32-bit int respectively.

2.2.3 \texttt{-1}

Now you see that \texttt{-1} is when all bits are set. Often, you can find the \texttt{-1} constant in all sorts of code, where a constant with all bits set are needed, for example, a mask.

For example: 3.18.1 on page 526.

2.3 Integer overflow

I intentionally put this section after the section about signed number representation.

First, take a look at this implementation of \texttt{itoa()} function from [Brian W. Kernighan, Dennis M. Ritchie, \textit{The C Programming Language}, 2ed, (1988)]:

```c
void itoa(int n, char s[])
{
    int i, sign;
    if ((sign = n) < 0) /* record sign */
        n = -n; /* make n positive */
    i = 0;
    do { /* generate digits in reverse order */
        s[i++] = n % 10 + '0'; /* get next digit */
    } while ((n /= 10) > 0); /* delete it */
```
if (sign < 0)
    s[i++] = '.';
else
    s[i++] = '0';
strrev(s);

(The full source code: https://beginners.re/current-tree/fundamentals/itoa_KR.c)

It has a subtle bug. Try to find it. You can download source code, compile it, etc. The answer on the next page.

Exercise 3-4. In a two’s complement number representation, our version of \textit{itoa} does not handle the largest negative number, that is, the value of \( n \) equal to \(-2^{\text{wordsize}-1}\). Explain why not. Modify it to print that value correctly, regardless of the machine on which it runs.

The answer is: the function cannot process largest negative number (\texttt{INT\_MIN} or \texttt{0x80000000} or -2147483648) correctly.

How to change sign? Invert all bits and add 1. If to invert all bits in \texttt{INT\_MIN} value (\texttt{0x80000000}), this is \texttt{0x7fffffff}. Add 1 and this is \texttt{0x80000000} again. So changing sign has no effect. This is an important artifact of two’s complement system.

Further reading:

- blexim – Basic Integer Overflows\textsuperscript{12}
- Yannick Moy, Nikolaj Bjørner, and David Sielaff – Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis\textsuperscript{13}

2.4 AND

2.4.1 Checking if a value is on \( 2^n \) boundary

If you need to check if your value is divisible by \( 2^n \) number (like 1024, 4096, etc.) without remainder, you can use a \& operator in C/C++, but there is a simpler way. 4096 is \texttt{0x1000}, so it always has \( 4 \times 3 = 12 \) lower bits cleared.

What you need is just:

```c
if (value&0xFFF)
{
    printf ("value is not divisible by 0x1000 (or 4096)\n");
    printf ("by the way, remainder is %d\n", value&0xFFF);
}
else
    printf ("value is divisible by 0x1000 (or 4096)\n");
```

In other words, this code checks if there are any bit set among lower 12 bits. As a side effect, lower 12 bits is always a remainder from division a value by 4096 (because division by \( 2^n \) is merely a right shift, and shifted (and dropped) bits are bits of remainder).

Same story if you want to check if the number is odd or even:

```c
if (value&1) // odd
else // even
```

This is merely the same as if to divide by 2 and get 1-bit remainder.

2.4.2 KOI-8R Cyrillic encoding

It was a time when 8-bit ASCII table wasn’t supported by some Internet services, including email. Some supported, some others—not.

It was also a time, when non-Latin writing systems used second half of 8-bit ASCII table to accommodate non-Latin characters. There were several popular Cyrillic encodings, but KOI-8R (devised by Andrey "ache" Chernov) is somewhat unique in comparison with others.

\textsuperscript{12}http://phrack.org/issues/60/10.html
\textsuperscript{13}https://yurichev.com/mirrors/SMT/z3prefix.pdf
Someone may notice that Cyrillic characters are allocated almost in the same sequence as Latin ones. This leads to one important property: if all 8th bits in Cyrillic text encoded in KOI-8R are to be reset, a text transforms into transliterated text with Latin characters in place of Cyrillic. For example, Russian sentence:

Мой дядя самых честных правил, Когда не в шутку занемог, Он уважать себя заставил, И лучше выдумать не мог.

...if encoded in KOI-8R and then 8th bit stripped, transforms into:

mOJ DQDQ SAMYH ^ESTNYH PRAWIL, kOGDA NE W [UTKU ZANEMOG, oN UWAVATX SEBQ ZASTAWIL, i LU^[E WYDUMATX NE MOG.

...perhaps this is not very appealing æsthetically, but this text is still readable to Russian language natives. Hence, Cyrillic text encoded in KOI-8R, passed through an old 7-bit service will survive into transliterated, but still readable text.

Stripping 8th bit is automatically transposes any character from the second half of the (any) 8-bit ASCII table to the first one, into the same place (take a look at red arrow right of table). If the character has already been placed in the first half (i.e., it has been in standard 7-bit ASCII table), it’s not transposed.

Perhaps, transliterated text is still recoverable, if you’ll add 8th bit to the characters which were seems transliterated.

Drawback is obvious: Cyrillic characters allocated in KOI-8R table are not in the same sequence as in Russian/Bulgarian/Ukrainian/etc. alphabet, and this isn’t suitable for sorting, for example.

2.5 AND and OR as subtraction and addition

2.5.1 ZX Spectrum ROM text strings

Those who once investigated ZX Spectrum ROM internals, probably noticed that the last symbol of each text string is seemingly absent.
There are present, in fact.

Here is excerpt of ZX Spectrum 128K ROM disassembled:

```c
unsigned char hw[] = {
    'H',
    'e',
    'l',
    'l',
    'o' | 0x80
};

void print_string()
{
    for (int i=0; ; i++)
    {
        if (hw[i] & 0x80) // check MSB
        {
            // clear MSB
            // (in other words, clear all, but leave 7 lower bits intact)
            printf ("%c", hw[i] & 0x7f);
        }
        // stop
    }
}
```

(last character has most significant bit set, which marks string end. Presumably, it was done to save some space? Old 8-bit computers have very tight environment.

Characters of all messages are always in standard 7-bit ASCII table, so it's guaranteed 7th bit is never used for characters.

To print such string, we must check MSB of each byte, and if it's set, we must clear it, then print character, and then stop. Here is a C example:

(http://www.matthew-wilson.net/spectrum/rom/128_ROM0.html)

Figure 2.2: Part of ZX Spectrum ROM
Now what is interesting, since 7th bit is the most significant bit (in byte), we can check it, set it and remove it using arithmetical operations instead of logical.

I can rewrite my C example:

```c
unsigned char hw[] = {
    'H', 'e', 'l', 'l', 'o' + 0x80
};
void print() {
    for (int i = 0; ; i++) {
        // hw[] must have 'unsigned char' type
        if (hw[i] >= 0x80) { // check for MSB
            printf("%c", hw[i]-0x80); // clear MSB
            // stop
            break;
        }
        printf("%c", hw[i]);
    }
}
```

By default, `char` is signed type in C/C++, so to compare it with variable like 0x80 (which is negative (-128) if treated as signed), we must treat each character in text message as unsigned.

Now if 7th bit is set, the number is always larger or equal to 0x80. If 7th bit is clear, the number is always smaller than 0x80.

Even more than that: if 7th bit is set, it can be cleared by subtracting 0x80, nothing else. If it’s not set beforehand, however, subtracting will destruct other bits.

Likewise, if 7th bit is clear, it’s possible to set it by adding 0x80. But if it’s set beforehand, addition operation will destruct some other bits.

In fact, this is valid for any bit. If the 4th bit is clear, you can set it just by adding 0x10: 0x100 + 0x10 = 0x110. If the 4th bit is set, you can clear it by subtracting 0x10: 0x1234 - 0x10 = 0x1224.

It works, because carry isn’t happened during addition/subtraction. It will, however, happen, if the bit is already set there before addition, or absent before subtraction.

Likewise, addition/subtraction can be replaced using OR/AND operation if two conditions are met: 1) you want to add/subtract by a number in form of $2^n$; 2) this bit in source value is clear/set.

For example, addition of 0x20 is the same as ORing value with 0x20 under condition that this bit is clear before: 0x1204 | 0x20 = 0x1204 + 0x20 = 0x1224.

Subtraction of 0x20 is the same as ANDing value with 0x20 (0x....FFDF), but if this bit is set before: 0x1234 & (~0x20) = 0x1234 & 0xFFDF = 0x1234-0x20 = 0x1214.

Again, it works because carry not happened when you add $2^n$ number and this bit isn’t set before.

This property of boolean algebra is important, worth understanding and keeping it in mind.

Another example in this book: 3.19.3 on page 536.

### 2.6 XOR (exclusive OR)

XOR is widely used when one needs just to flip specific bit(s). Indeed, the XOR operation applied with 1 effectively inverts a bit:
And vice-versa, the XOR operation applied with 0 does nothing, i.e., it’s an idle operation. This is a very important property of the XOR operation and it’s highly recommended to memorize it.

### 2.6.1 Logical difference

In Cray-1 supercomputer (1976-1977) manual \(^{14}\), you can find XOR instruction was called *logical difference*. Indeed, XOR(a,b)=1 if a!=b.

### 2.6.2 Everyday speech

XOR operation present in common everyday speech. When someone asks “please buy apples or bananas”, this usually means “buy the first object or the second, but not both”—this is exactly exclusive OR, because logical OR would mean “both objects are also fine”.

Some people suggest “and/or” should be used in everyday speech to make emphasis that logical OR is used instead of exclusive OR: [https://en.wikipedia.org/wiki/And/or](https://en.wikipedia.org/wiki/And/or).

### 2.6.3 Encryption

XOR is heavily used in both amateur (9.1 on page 908) and real encryption (at least in Feistel network). XOR is very useful here because: cipher_text = plain_text ⊕ key and then: (plain_text ⊕ key) ⊕ key = plain_text.

### 2.6.4 RAID4

RAID4 offers a very simple method to protect hard disks. For example, there are several disks (\(D_1, D_2, D_3, \text{ etc.}\)) and one parity disk (\(P\)). Each bit/byte written to parity disk is calculated and written on-fly:

\[ P = D_1 ⊕ D_2 ⊕ D_3 \quad (2.1) \]

If any of disks is failed, for example, \(D_2\), it’s restored using the very same way:

\[ D_2 = D_1 ⊕ P ⊕ D_3 \quad (2.2) \]

If parity disk failed, it is restored using 2.1 way. If two of any disks are failed, then it wouldn’t be possible to restore both.

RAID5 is more advanced, but this XOR property is still exploited there.

That’s why RAID controllers has hardware “XOR accelerators” helping to XOR large chunks of written data on-fly. When computers get faster and faster, it now can be done at software level, using SIMD.

### 2.6.5 XOR swap algorithm

Hard to believe, but this code swaps values in EAX and EBX without aid of any other additional register or memory cell:

```assembly
xor eax, ebx
xor ebx, eax
xor eax, ebx
```

Let’s find out, how it works. First, we will rewrite it to step aside from x86 assembly language:

What X and Y has at each step? Just keep in mind the simple rule: \((X \oplus Y) \oplus Y = X\) for any values of X and Y.

Let’s see, X after 1st step has \(X \oplus Y\); Y after 2nd step has \(Y \oplus (X \oplus Y) = X\); X after 3rd step has \(X \oplus Y\).

Hard to say if anyone should use this trick, but it serves as a good demonstration example of XOR properties.

Wikipedia article (https://en.wikipedia.org/wiki/XOR_swap_algorithm) has also yet another explanation: addition and subtraction operations can be used instead of XOR:

\[
\begin{align*}
X &= X + Y \\
Y &= X - Y \\
X &= X - Y
\end{align*}
\]

Let’s see: X after 1st step has \(X + Y\); Y after 2nd step has \(X + Y - Y = X\); X after 3rd step has \(X + Y - X = Y\).

### 2.6.6 XOR linked list

Doubly linked list is a list in which each element has link to the previous element and to the next one. Hence, it’s very easy to traverse list backwards or forward. std::list in C++ implements doubly linked list which also is examined in this book: 3.21.4 on page 566.

So each element has two pointers. Is it possible, perhaps in environment of small RAM footprint, to preserve all functionality with one pointer instead of two? Yes, if a value of \(\text{prev} \oplus \text{next}\) will be stored in this memory cell, which is usually called “link”.

Maybe, we could say that address to the previous element is “encrypted” using address of next element and otherwise: next element address is “encrypted” using previous element address.

When we traverse this list forward, we always know address of the previous element, so we can “decrypt” this field and get address of the next element. Likewise, it’s possible to traverse this list backwards, “decrypting” this field using next element’s address.

But it’s not possible to find address of previous or next element of some specific element without knowing address of the first one.

Couple of things to complete this solution: first element will have address of next element without any XOR-ing, last element will have address of previous element without any XOR-ing.

Now let’s sum it up. This is example of doubly linked list of 5 elements. \(A_x\) is address of element.

<table>
<thead>
<tr>
<th>address</th>
<th>link field contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_0)</td>
<td>(A_1)</td>
</tr>
<tr>
<td>(A_1)</td>
<td>(A_0 \oplus A_2)</td>
</tr>
<tr>
<td>(A_2)</td>
<td>(A_1 \oplus A_3)</td>
</tr>
<tr>
<td>(A_3)</td>
<td>(A_2 \oplus A_4)</td>
</tr>
<tr>
<td>(A_4)</td>
<td>(A_3)</td>
</tr>
</tbody>
</table>

And again, hard to say if anyone should use this tricky hacks, but this is also a good demonstration of XOR properties. As with XOR swap algorithm, Wikipedia article about it also offers way to use addition or subtraction instead of XOR: https://en.wikipedia.org/wiki/XOR_linked_list.

### 2.6.7 Switching value trick

... found in Jorg Arndt — Matters Computational / Ideas, Algorithms, Source Code 15.

You want a variable to be switching between 123 and 456. You may write something like:

```c
if (a==123) 
  a=456;
else
  a=123;
```

15https://www.jjj.de/fxt/fxtbook.pdf
But this can be done using a single operation:

```c
#include <stdio.h>

int main()
{
    int a=123;
    #define C 123^456
    a=a^C;
    printf ("%d\n", a);
    a=a^C;
    printf ("%d\n", a);
    a=a^C;
    printf ("%d\n", a);
}
```

It works because $123 \oplus 123 \oplus 456 = 0 \oplus 456 = 456$ and $456 \oplus 123 \oplus 456 = 456 \oplus 456 \oplus 123 = 0 \oplus 123 = 123$.

One can argue, worth it using or not, especially keeping in mind code readability. But this is yet another demonstration of XOR properties.

### 2.6.8 Zobrist hashing / tabulation hashing

If you work on a chess engine, you traverse a game tree many times per second, and often, you can encounter the same position, which has already been processed.

So you have to use a method to store already calculated positions somewhere. But chess position can require a lot of memory, and a hash function would be used instead.

Here is a way to compress a chess position into 64-bit value, called Zobrist hashing:

```c
// we have 8*8 board and 12 pieces (6 for white side and 6 for black)
uint64_t table[12][8][8]; // filled with random values

int position[8][8]; // for each square on board. 0 - no piece. 1..12 - piece

uint64_t hash;

for (int row=0; row<8; row++)
    for (int col=0; col<8; col++)
    {
        int piece=position[row][col];
        if (piece!=0)
            hash=hash^table[piece][row][col];
    }

return hash;
```

Now the most interesting part: if the next (modified) chess position differs only by one (moved) piece, you don’t need to recalculate hash for the whole position, all you need is:

```c
hash=...; // (already calculated)

// subtract information about the old piece:
hash=hash^table[old_piece][old_row][old_col];

// add information about the new piece:
hash=hash^table[new_piece][new_row][new_col];
```

### 2.6.9 By the way

The usual OR also sometimes called inclusive OR (or even IOR), as opposed to exclusive OR. One place is operator Python’s library: it’s called operator.iOr here.
2.6.10 AND/OR/XOR as MOV

OR reg, 0xFFFFFFFF sets all bits to 1, hence, no matter what has been in register before, it will be set to −1. OR reg, -1 is shorter than MOV reg, -1, so MSVC uses OR instead the latter, for example: 3.18.1 on page 526.

Likewise, AND reg, 0 always resets all bits, hence, it acts like MOV reg, 0.

XOR reg, reg, no matter what has been in register beforehand, resets all bits, and also acts like MOV reg, 0.

2.7 Population count

POPCNT instruction is population count (AKA Hamming weight). It just counts number of bits set in an input value.

As a side effect, POPCNT instruction (or operation) can be used to determine, if the value has $2^n$ form. Since, $2^n$ number always has just one single bit, POPCNT’s result will always be just 1.

For example, I once wrote a base64 strings scanner for hunting something interesting in binary files\(^\text{16}\). And there is a lot of garbage and false positives, so I add an option to filter out data blocks which has size of $2^n$ bytes (i.e., 256 bytes, 512, 1024, etc.). The size of block is checked just like this:

```c
if (popcnt(size)==1)
    // OK
```

The instruction is also known as “NSA\(^\text{17}\) instruction” due to rumors:

```
This branch of cryptography is fast-paced and very politically charged. Most designs are secret; a majority of military encryptions systems in use today are based on LFSRs. In fact, most Cray computers (Cray 1, Cray X-MP, Cray Y-MP) have a rather curious instruction generally known as “population count.” It counts the 1 bits in a register and can be used both to efficiently calculate the Hamming distance between two binary words and to implement a vectorized version of a LFSR. I’ve heard this called the canonical NSA instruction, demanded by almost all computer contracts.
```

[Bruce Schneier, Applied Cryptography, (John Wiley & Sons, 1994)]

2.8 Endianness

The endianness is a way of representing values in memory.

2.8.1 Big-endian

The 0x12345678 value is represented in memory as:

<table>
<thead>
<tr>
<th>address in memory</th>
<th>byte value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>0x12</td>
</tr>
<tr>
<td>+1</td>
<td>0x34</td>
</tr>
<tr>
<td>+2</td>
<td>0x56</td>
</tr>
<tr>
<td>+3</td>
<td>0x78</td>
</tr>
</tbody>
</table>

Big-endian CPUs include Motorola 68k, IBM POWER.

2.8.2 Little-endian

The 0x12345678 value is represented in memory as:

\(^\text{16}\)https://github.com/DennisYurichev/base64scanner

\(^\text{17}\)National Security Agency

460
Little-endian CPUs include Intel x86. One important example of little-endian using in this book is: ?? on page ??.

2.8.3 Example

Let’s take big-endian MIPS Linux installed and ready in QEMU. And let’s compile this simple example:

```c
#include <stdio.h>

int main()
{
    int v;
    v=123;
    printf ("%02X %02X %02X %02X\n",
            (*((char*)&v),
            *((char*)&v)+1),
            *((char*)&v)+2),
            *((char*)&v)+3));
}
```

After running it we get:

```bash
root@debian-mips:~# ./a.out
00 00 00 7B
```

That is it. 0x7B is 123 in decimal. In little-endian architectures, 7B is the first byte (you can check on x86 or x86-64), but here it is the last one, because the highest byte goes first.

That’s why there are separate Linux distributions for MIPS (“mips” (big-endian) and “mipsel” (little-endian)). It is impossible for a binary compiled for one endianness to work on an OS with different endianness.

There is another example of MIPS big-endiannes in this book: 1.30.4 on page 362.

2.8.4 Bi-endian

CPUs that may switch between endianness are ARM, PowerPC, SPARC, MIPS, IA64, etc.

2.8.5 Converting data

The BSWAP instruction can be used for conversion.

TCP/IP network data packets use the big-endian conventions, so that is why a program working on a little-endian architecture has to convert the values. The htonl() and htons() functions are usually used.

In TCP/IP, big-endian is also called “network byte order”, while byte order on the computer “host byte order”. “host byte order” is little-endian on Intel x86 and other little-endian architectures, but it is big-endian on IBM POWER, so htonl() and htons() don’t shuffle any bytes on the latter.

2.9 Memory

There are 3 main types of memory:

18Available for download here: http://go.yurichev.com/17008
19Intel Architecture 64 (Itanium)
• Global memory AKA “static memory allocation”. No need to allocate explicitly, the allocation is performed just by declaring variables/arrays globally. These are global variables, residing in the data or constant segments. They are available globally (hence, considered as an anti-pattern). Not convenient for buffers/arrays, because they must have a fixed size. Buffer overflows that occur here usually overwrite variables or buffers residing next to them in memory. There’s an example in this book: 1.12.3 on page 76.

• Stack AKA “allocate on stack”. The allocation is performed just by declaring variables/arrays locally in the function. These are usually local variables for the function. Sometimes these local variable are also available to descending functions (to callee functions, if caller passes a pointer to a variable to the callee to be executed). Allocation and deallocation are very fast, it just SP needs to be shifted. But they’re also not convenient for buffers/arrays, because the buffer size has to be fixed, unless alloca() (1.9.2 on page 34) (or a variable-length array) is used. Buffer overflows usually overwrite important stack structures: 1.26.2 on page 271.

• Heap AKA “dynamic memory allocation”. Allocation/deallocation is performed by calling malloc()/free() or new/delete in C++. This is the most convenient method: the block size may be set at runtime. Resizing is possible (using realloc()), but can be slow. This is the slowest way to allocate memory: the memory allocator must support and update all control structures while allocating and deallocating. Buffer overflows usually overwrite these structures. Heap allocations are also source of memory leak problems: each memory block has to be deallocated explicitly, but one may forget about it, or do it incorrectly.

Another problem is the “use after free”—using a memory block after free() has been called on it, which is very dangerous.

Example in this book: 1.30.2 on page 344.

2.10 CPU

2.10.1 Branch predictors

Some latest compilers try to get rid of conditional jump instructions. Examples in this book are: 1.18.1 on page 134, 1.18.3 on page 142, 1.28.5 on page 328.

This is because the branch predictor is not always perfect, so the compilers try to do without conditional jumps, if possible.

Conditional instructions in ARM (like ADRcc) are one way, another one is the CMOVcc x86 instruction.

2.10.2 Data dependencies

Modern CPUs are able to execute instructions simultaneously (OOE20), but in order to do so, the results of one instruction in a group must not influence the execution of others. Hence, the compiler endeavors to use instructions with minimal influence on the CPU state.

That’s why the LEA instruction is so popular, because it does not modify CPU flags, while other arithmetic instructions does.

2.11 Hash functions

A very simple example is CRC32, an algorithm that provides “stronger” checksum for integrity checking purposes. It is impossible to restore the original text from the hash value, it has much less information: But CRC32 is not cryptographically secure: it is known how to alter a text in a way that the resulting CRC32 hash value will be the one we need. Cryptographic hash functions are protected from this.

MD5, SHA1, etc. are such functions and they are widely used to hash user passwords in order to store them in a database. Indeed: an Internet forum database may not contain user passwords (a stolen database can compromise all users’ passwords) but only hashes (so a cracker can’t reveal the passwords). Besides, an Internet forum engine does not need to know your password exactly, it needs only to check if its hash is the same as the one in the database, and give you access if they match. One of the simplest

---

20Out-of-Order Execution
password cracking methods is just to try hashing all possible passwords in order to see which matches the resulting value that we need. Other methods are much more complex.

2.11.1 How do one-way functions work?

A one-way function is a function which is able to transform one value into another, while it is impossible (or very hard) to reverse it. Some people have difficulties while understanding how this is possible at all. Here is a simple demonstration.

We have a vector of 10 numbers in range 0..9, each is present only once, for example:

| 4 | 6 | 0 | 1 | 3 | 5 | 7 | 8 | 9 | 2 |

The algorithm for the simplest possible one-way function is:

- take the number at zeroth position (4 in our case);
- take the number at first position (6 in our case);
- swap numbers at positions of 4 and 6.

Let's mark the numbers at positions 4 and 6:

<table>
<thead>
<tr>
<th>4</th>
<th>6</th>
<th>0</th>
<th>1</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td>6</td>
<td></td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Let's swap them and we get this result:

| 4 | 6 | 0 | 1 | 7 | 5 | 3 | 8 | 9 | 2 |

While looking at the result, and even if we know the algorithm, we can’t know unambiguously the initial state, because the first two numbers could be 0 and/or 1, and then they could participate in the swapping procedure.

This is an utterly simplified example for demonstration. Real one-way functions are much more complex.
Chapter 3

Slightly more advanced examples

3.1 Zero register

x86 architecture lacks zero register, unlike MIPS and ARM. However, it’s often a case, when a compiler assigns a zero to a register, that will hold it upon the function’s exit.

This is a case of Mahjong game from Windows 7 x86. Zeroed EBX (at 0x010281BF) is used to initialize local variables, to pass zero argument into other functions and to compare values against it.

Listing 3.1: Mahjong.exe from Windows 7 x86

```
.text:010281AE sub_10281AE proc near ; CODE XREF: sub_1028790+4FFp
.text:010281AE ; sub_102909A+357p ...
.text:010281AE var_34 = dword ptr -34h
.text:010281AE var_30 = dword ptr -30h
.text:010281AE var_2C = dword ptr -2Ch
.text:010281AE var_28 = dword ptr -28h
.text:010281AE var_24 = dword ptr -24h
.text:010281AE var_20 = dword ptr -20h
.text:010281AE var_1C = dword ptr -1Ch
.text:010281AE var_18 = dword ptr -18h
.text:010281AE var_14 = dword ptr -14h
.text:010281AE var_10 = dword ptr -10h
.text:010281AE var_4  = dword ptr -4
.text:010281AE arg_0  = dword ptr 8
.text:010281AE arg_4  = byte ptr 0Ch
.text:010281AE
.text:010281AE push 28h
.text:010281AE mov eax, offset __ehandler$
 \_enable_segment@_Helper@_Concurrent_vector_base_v4@detaile
.text:010281AE call __EH_prolog3
.text:010281BE mov edi, ecx
.text:010281BF mov esi, [ebp+arg_0] ; *
.text:010281C0 mov ebx, ebx          ; *
.text:010281C4 cmp [esi], ebx      ; *
.text:010281C6 jbe short loc_10281E8
.text:010281C8 loc_10281C8:       ; CODE XREF: sub_10281AE+38j
.text:010281C8 mov eax, [esi+0Ch]
.text:010281C8 mov ecx, [ebp+var_10]
.text:010281CE push dword ptr [eax+ecx*4]
.text:010281D0 call sub_10506C9
.text:010281D3 mov eax, [ebp+var_10]
.text:010281D9 pop ecx
.text:010281DA mov ecx, [esi+0Ch]
.text:010281DD mov [ecx+eax*4], ebx ; *
.text:010281E0 inc eax
.text:010281E1 mov [ebp+var_10], eax
.text:010281E4 cmp eax, [esi]
.text:010281E6 jb short loc_10281C8
```
.text:0102829B loc_102829B: ; CODE XREF: sub_10281AE+81j
.text:0102829B lea ecx, [ebp-var_24]
.text:0102829E mov byte ptr [ebp-var_4], bl
.text:010282A1 call sub_10349DB
.text:010282A6 loc_10282A6: ; CODE XREF: sub_10281AE+72j
.text:010282A6 ; sub_10281AE+83j
.text:010282A6 push [ebp+arg_0]
.text:010282A9 lea ecx, [ebp-var_34]
.text:010282AC call sub_1049228
.text:010282B1 inc [ebp+arg_0]
.text:010282B4 mov eax, [ebp+arg_0]
.text:010282B7 cmp eax, [edi+0B0h]
.text:010282BD jb loc_102820F
.text:010282C3 loc_10282C3: ; CODE XREF: sub_10281AE+5Bj
.text:010282C3 cmp [ebp+arg_4], bl
.text:010282C6 jz short loc_1028337
.text:010282C8 mov eax, dword_10B8AD8
.text:010282CD mov esi, ds:EnableMenuItem
.text:010282D3 mov edi, 40002
.text:010282D8 cmp [eax+8], ebx ; *
.text:010282DB jnz short loc_10282EC
.text:010282DD push 3 ; uEnable
.text:010282DF push edi ; uIDEnableItem
.text:010282E0 push hMenu ; hMenu
.text:010282E6 call esi ; EnableMenuItem
.text:010282E8 push 3
.text:010282EA jmp short loc_10282F7
.text:010282EC ;
.text:010282EC ;
.text:010282EE loc_10282EC: ; CODE XREF: sub_10281AE+12Dj
.text:010282EE push ebx ; *
.text:010282ED push edi ; uIDEnableItem
.text:010282EE push hMenu ; hMenu
.text:010282F4 call esi ; EnableMenuItem
.text:010282F6 push ebx ; *
.text:010282F7 loc_10282F7: ; CODE XREF: sub_10281AE+13Cj
.text:010282F7 push edi ; uIDEnableItem
.text:010282FE push hmenu ; hMenu
.text:01028300 call esi ; EnableMenuItem
.text:01028303 mov ecx, dword_10B8AD8
.text:01028306 call sub_1020402
.text:0102830B mov edi, 40001
.text:01028310 test al, al
.text:01028312 jz short loc_1028321
.text:01028314 push ebx ; *
.text:01028315 push edi ; uIDEnableItem
.text:01028316 push hMenu ; hMenu
.text:0102831C call esi ; EnableMenuItem
.text:0102831E push ebx ; *
.text:0102831F jmp short loc_102832E
.text:01028321 ;
.text:01028321 ;
.text:01028321 loc_1028321: ; CODE XREF: sub_10281AE+164j
.text:01028321 push 3 ; uEnable
.text:01028323 push edi ; uIDEnableItem
.text:01028324 push hMenu ; hMenu
.text:0102832A call esi ; EnableMenuItem
.text:0102832C push 3 ; uEnable
.text:0102832E loc_102832E: ; CODE XREF: sub_10281AE+171j
.text:0102832E push edi ; uIDEnableItem
.text:0102832F push hmenu ; hMenu
.text:01028335 call esi ; EnableMenuItem
.text:01028337 loc_1028337: ; CODE XREF: sub_10281AE+118j
.text:01028337 lea ecx, [ebp-var_34]
.text:0102833A call sub_10349DB
By the way, IDA can rename a register inside a function (press n):

Listing 3.2: Mahjong.exe from Windows 7 x86

```
.text:010281AE sub_10281AE    proc near
...
.text:010281AE zero = ebx
...
.text:010281BF xor zero, zero    ; *
.text:010281C1 mov [ebp+var_10], zero    ; *
.text:010281C4 cmp [esi], zero    ; *
.text:010281C6 jbe short loc_10281E8
...
```

### 3.2 Double negation

A popular way\(^1\) to convert non-zero value into 1 (or boolean true) and zero value into 0 (or boolean false) is `!!statement`:

```c
int convert_to_bool(int a)
{
    return !!a;
}
```

Optimizing GCC 5.4 x86:

```
convert_to_bool:
    mov edx, DWORD PTR [esp+4]
    xor eax, eax
    test edx, edx
    setne al
    ret
```

XOR always clears return value in EAX, even in case if SETNE will not trigger. I.e., XOR sets default return value to zero.

If the input value is not equal to zero (-NE suffix in SET instruction), 1 is set to AL, otherwise AL isn’t touched.

Why SETNE operates on low 8-bit part of EAX register? Because the matter is just in the last bit (0 or 1), while other bits are cleared by XOR.

Therefore, that C/C++ code could be rewritten like this:

```c
int convert_to_bool(int a)
{
    if (a!=0)
        return 1;
    else
        return 0;
}
```

...or even:

```c
int convert_to_bool(int a)
{
    if (a)
        return 1;
}
```

---

\(^1\)This way is also controversial, because it leads to hard-to-read code

467
else
    return 0;
};

Compilers targeting CPUs lacking instruction similar to SET, in this case, generates branching instructions, etc.

### 3.3 const correctness

This is undeservedly underused feature of many programming languages. Read here about its importance: [1](#), [2](#).

Ideally, everything you don’t modify should have `const` modifier.

Interestingly, how `const correctness` is implemented at low level. There are no runtime checks of local `const` variables and function arguments (only compile-time checks). But global variables of such a type are to be allocated in read-only data segments.

This is example is to be crashed, because if compiled by MSVC for win32, the `a` global variable is allocated in `.rdata` read-only segment:

```c
const a=123;
void f(int *i)
{
    *i=11; // crash
};
int main()
{
    f(&a);
    return a;
};
```

`Anonymous` (not linked to a variable name) C strings also have `const char*` type. You can’t modify them:

```c
#include <string.h>
#include <stdio.h>

void alter_string(char *s)
{
    strcpy(s, "Goodbye!");
    printf("Result: \%s\n", s);
};
int main()
{
    alter_string("Hello, world!\n");
};
```

This code will crash on Linux ("segmentation fault") and on Windows if compiled by MinGW.

GCC for Linux places all text strings info `.rodata` data segment, which is explicitly read-only ("read only data"):

```
$ objdump -s 1
...
Contents of section .rodata:
400600 01000200 52657375 6c743a20 25730a00 ....Result: %s..
400610 48656c6c 6f2c2077 6f726c64 210a00 Hello, world!..
```

When the `alter_string()` function tries to write there, exception occurred.

Things are different in the code generated by MSVC, strings are located in `.data` segment, which has no `READONLY` flag. MSVC’s developers misstep?
C:\...>objdump -s 1.exe
...

Contents of section .data:
40b000 476f6f64 62796521 00000000 52657375 Goodbye!....Resu
40b010 6c743a20 25730a00 48656c6c 6f2c20lt: %s..Hello, w
40b020 6f726c64 210a0000 00000000 00000000 orld!...........
40b030 01000000 00000000 00000000 00000000 ..........@.....
...
C:\...>objdump -x 1.exe
...

Sections:
Idx Name Size VMA LMA File off Algn
0 .text 00006d2a 00401000 00401000 00000400 2**2
CONTENTS, ALLOC, LOAD, READONLY, CODE
1 .rdata 00002262 00408000 00408000 00007200 2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA
2 .data 00000e00 0040b000 0040b000 00009600 2**2
CONTENTS, ALLOC, LOAD, DATA
3 .reloc 00000b98 0040e000 0040e000 0000a400 2**2
CONTENTS, ALLOC, LOAD, READONLY, DATA

However, MinGW hasn't this fault and allocates text strings in .rdata segment.

### 3.3.1 Overlapping const strings

The fact that an anonymous C-string has `const` type (1.5.1 on page 9), and that C-strings allocated in constants segment are guaranteed to be immutable, has an interesting consequence: the compiler may use a specific part of the string.

Let's try this example:

```c
#include <stdio.h>

int f1()
{
    printf ("world
");
}

int f2()
{
    printf ("hello world
");
}

int main()
{
    f1();
    f2();
}
```

Common C/C++-compilers (including MSVC) allocate two strings, but let's see what GCC 4.8.1 does:

```
Listing 3.3: GCC 4.8.1 + IDA listing

f1 proc near
s     = dword ptr -1Ch
sub    esp, 1Ch
mov    [esp+1Ch+s], offset s ; "world\n"
call   _puts
add    esp, 1Ch
retn
f1 endp
```

469
Indeed: when we print the “hello world” string these two words are positioned in memory adjacently and puts() called from f2() function is not aware that this string is divided. In fact, it’s not divided; it’s divided only virtually, in this listing.

When puts() is called from f1(), it uses the “world” string plus a zero byte. puts() is not aware that there is something before this string!

This clever trick is often used by at least GCC and can save some memory. This is close to string interning. Another related example is here: 3.4.

### 3.4 strstr() example

Let’s back to the fact that GCC sometimes can use part of string: 3.3.1 on the previous page.

The strstr() C/C++ standard library function is used to find any occurrence in a string. This is what we will do:

```c
#include <string.h>
#include <stdio.h>

int main()
{
    char *s="Hello, world!";
    char *w=strstr(s, "world");

    printf ("%p, [%s]\n", s, s);
    printf ("%p, [%s]\n", w, w);
}
```

The output is:

```
0x8048530, [Hello, world!]
0x8048537, [world!]
```

The difference between the address of the original string and the address of the substring that strstr() has returned is 7. Indeed, “Hello,” string has length of 7 characters.

The printf() function during second call has no idea there are some other characters before the passed string and it prints characters from the middle of original string till the end (marked by zero byte).

### 3.5 qsort() revisited

(Let’s back to the fact that CMP instruction works like SUB: 1.12.4 on page 86.)

Now that you already familiar with qsort() function (1.33 on page 381), here is a nice example where comparison operation (CMP) can be replaced by subtraction operation (SUB).

```c
/* qsort int comparison function */
int int_cmp(const void *a, const void *b)
{
    const int *ia = (const int *)a; // casting pointer types
    const int *ib = (const int *)b;
```
3.6 Temperature converting

Another very popular example in programming books for beginners is a small program that converts Fahrenheit temperature to Celsius or back.

\[
C = \frac{5 \cdot (F - 32)}{9}
\]

We can also add simple error handling: 1) we must check if the user has entered a correct number; 2) we must check if the Celsius temperature is not below \(-273\) (which is below absolute zero, as we may recall from school physics lessons).

The exit() function terminates the program instantly, without returning to the caller function.

3.6.1 Integer values

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int celsius, fahr;
    printf("Enter temperature in Fahrenheit:\n");
    if (scanf("%d", &fahr)!=1)
    {
        printf("Error while parsing your input\n");
        exit(0);
    }
    celsius = 5 * (fahr-32) / 9;
    if (celsius<-273)
    {
        printf("Error: incorrect temperature!\n");
        exit(0);
    }
    printf("Celsius: %d\n", celsius);
    return 0;
}
```

Optimizing MSVC 2012 x86

| $SG4228$ DB | 'Enter temperature in Fahrenheit:', 0AH, 00H |
| $SG4230$ DB | '%'D', 00H |
| $SG4231$ DB | 'Error while parsing your input', 0AH, 00H |
| $SG4233$ DB | 'Error: incorrect temperature!', 0AH, 00H |
What we can say about it:

- The address of printf() is first loaded in the ESI register, so the subsequent printf() calls are done just by the CALL ESI instruction. It’s a very popular compiler technique, possible if several consequent calls to the same function are present in the code, and/or if there is a free register which can be used for this.

- We see the ADD EAX, -32 instruction at the place where 32 has to be subtracted from the value. EAX = EAX + (-32) is equivalent to EAX = EAX – 32 and somehow, the compiler decided to use ADD instead of SUB. Maybe it’s worth it, it’s hard to be sure.

- The LEA instruction is used when the value is to be multiplied by 5: lea ecx, DWORD PTR [eax+eax*4]. Yes, \( i + i \times 4 \) is equivalent to \( i \times 5 \) and LEA works faster than IMUL. By the way, the SHL EAX, 2 / ADD EAX, EAX instruction pair could be also used here instead—some compilers do it like.

- The division by multiplication trick (3.12 on page 496) is also used here.
main() returns 0 if we don't have return 0 at its end. The C99 standard tells us [ISO/IEC 9899:TC3 (C C99 standard), (2007)]5.1.2.2.3 that main() will return 0 in case the return statement is missing.

This rule works only for the main() function.

Though, MSVC doesn't officially support C99, but maybe it support it partially?

**Optimizing MSVC 2012 x64**

The code is almost the same, but we can find INT 3 instructions after each exit() call.

```
xor ecx, ecx
    call QWORD PTR__imp_exit
    int 3
```

INT 3 is a debugger breakpoint.

It is known that exit() is one of the functions which can never return 2, so if it does, something really odd has happened and it's time to load the debugger.

### 3.6.2 Floating-point values

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    double celsius, fahr;
    printf("Enter temperature in Fahrenheit:\n");
    if (scanf("%lf", &fahr)!=1)
    {
        printf("Error while parsing your input\n");
        exit(0);
    }
    celsius = 5 * (fahr-32) / 9;
    if (celsius<-273)
    {
        printf("Error: incorrect temperature!\n");
        exit(0);
    }
    printf("Celsius: %lf\n", celsius);
}
```

MSVC 2010 x86 uses FPU instructions...

Listing 3.5: Optimizing MSVC 2010 x86

```
SG4038 DB 'Enter temperature in Fahrenheit:', 0AH, 00H
SG4040 DB '%lf', 00H
SG4041 DB 'Error while parsing your input', 0AH, 00H
SG4043 DB 'Error: incorrect temperature!', 0AH, 00H
SG4044 DB 'Celsius: %lf', 0AH, 00H

__real@c0711000000000000 DQ 0c0711000000000000r ; -273
__real@40220000000000000000 DQ 04022000000000000000r ; 9
__real@40140000000000000000 DQ 04014000000000000000r ; 5
__real@40400000000000000000 DQ 04040000000000000000r ; 32

_fahr$ = -8 ; size = 8
main PROC
    sub esp, 8
    push esi
    mov esi, DWORD PTR __imp__printf
    push OFFSET $SG4038 ; 'Enter temperature in Fahrenheit:'
    call esi ; call printf()
    lea eax, DWORD PTR _fahr$[esp+16]
    push eax
    ... another popular one is longjmp()
```
...but MSVC 2012 uses SIMD instructions instead:

Listing 3.6: Optimizing MSVC 2010 x86

```assembly
...but MSVC 2012 uses SIMD instructions instead:

Listing 3.6: Optimizing MSVC 2010 x86

```
Of course, SIMD instructions are available in x86 mode, including those working with floating point numbers.

It's somewhat easier to use them for calculations, so the new Microsoft compiler uses them.

We can also see that the \(-273\) value is loaded into XMM0 register too early. And that's OK, because the compiler may emit instructions not in the order they are in the source code.

### 3.7 Fibonacci numbers

Another widespread example used in programming textbooks is a recursive function that generates the Fibonacci numbers\(^3\). The sequence is very simple: each consecutive number is the sum of the previous two. The first two numbers are 0 and 1, or 1 and 1.

The sequence starts like this:

\[
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, ...
\]

#### 3.7.1 Example #1

The implementation is simple. This program generates the sequence until 21.

```c
#include <stdio.h>

void fib (int a, int b, int limit)
{
    printf ("%d\n", a+b);
    if (a+b > limit)
        return;
    fib (b, a+b, limit);
};

int main()
{
    printf ("0\n1\n1\n");  
    fib (1, 1, 20);
}
```

\(^3\)http://go.yurichev.com/17332
Listing 3.7: MSVC 2010 x86

```assembly
.a$ = 8 ; size = 4
.b$ = 12 ; size = 4
._limit$ = 16 ; size = 4
_fib PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    add eax, DWORD PTR _b$[ebp]
    push eax
    push OFFSET $SG2643
    call DWORD PTR __imp__printf
    add esp, 8
    mov ecx, DWORD PTR _a$[ebp]
    add ecx, DWORD PTR _b$[ebp]
    cmp ecx, DWORD PTR _limit$[ebp]
    jle SHORT $LN1@fib
    jmp SHORT $LN2@fib
$LN1@fib:
    mov edx, DWORD PTR _limit$[ebp]
    push edx
    mov eax, DWORD PTR _a$[ebp]
    add eax, DWORD PTR _b$[ebp]
    push eax
    mov ecx, DWORD PTR _b$[ebp]
    push ecx
    call _fib
    add esp, 12
$LN2@fib:
    pop ebp
    ret 0
_fib ENDP

_main PROC
    push ebp
    mov ebp, esp
    push OFFSET $SG2647 ; "0
1
1"
    call DWORD PTR __imp__printf
    add esp, 4
    push 20
    push 1
    push 1
    call _fib
    add esp, 12
    xor eax, eax
    pop ebp
    ret 0
_main ENDP
```

We will illustrate the stack frames with this.
Let's load the example in OllyDbg and trace to the last call of f():

![OllyDbg screenshot showing the last call of f()](image-url)

Figure 3.1: OllyDbg: last call of f()
Let’s investigate the stack more closely. Comments were added by the author of this book 4:

The function is recursive 5, hence stack looks like a “sandwich”.

We can also say that each function incarnation must not access stack elements beyond the boundaries of its frame (excluding function arguments), although it’s technically possible.

It’s usually true, unless the function has bugs.

Each saved EBP value is the address of the previous stack frame: this is the reason why some debuggers can easily divide the stack in frames and dump each function’s arguments.

As we see here, each function incarnation prepares the arguments for the next function call.

At the end we see the 3 arguments for main(). argc is 1 (yes, indeed, we have ran the program without command-line arguments).

This easily to lead to a stack overflow: just remove (or comment out) the limit check and it will crash with exception 0xC00000FD (stack overflow.)

3.7.2 Example #2

My function has some redundancy, so let’s add a new local variable next and replace all “a+b” with it:

```c
#include <stdio.h>
```

4By the way, it’s possible to select several entries in OllyDbg and copy them to the clipboard (Ctrl-C). That’s what was done by author for this example.

5i.e., it calls itself
void fib (int a, int b, int limit) {
    int next=a+b;
    printf ("%d\n", next);
    if (next > limit)
        return;
    fib (b, next, limit);
};

int main() {
    printf ("0\n1\n1\n");
    fib (1, 1, 20);
};

This is the output of non-optimizing MSVC, so the next variable is actually allocated in the local stack:

Listing 3.8: MSVC 2010 x86

```assembly
_fib PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _a$[ebp]
    add     eax, DWORD PTR _b$[ebp]
    mov     DWORD PTR _next$[ebp], eax
    mov     ecx, DWORD PTR _next$[ebp]
    push    ecx
    push    OFFSET $SG2751 ;'\d'
    call    DWORD PTR __imp__printf
    add     esp, 8
    mov     edx, DWORD PTR _next$[ebp]
    cmp     edx, DWORD PTR _limit$[ebp]
    jle     SHORT $LN1@fib
    jmp     SHORT $LN2@fib
$LN1@fib:
    mov     eax, DWORD PTR _limit$[ebp]
    push    eax
    mov     ecx, DWORD PTR _next$[ebp]
    push    ecx
    mov     edx, DWORD PTR _b$[ebp]
    push    edx
    call    _fib
    add     esp, 12
$LN2@fib:
    mov     esp, ebp
    pop     ebp
    ret     0
_fib ENDP

_main PROC
    push    ebp
    mov     ebp, esp
    push    OFFSET $SG2753 ; "0\n1\n1\n"
    call    DWORD PTR __imp__printf
    add     esp, 4
    push    20
    push    1
    push    1
    call    _fib
    add     esp, 12
    xor     eax, eax
    pop     ebp
    ret     0
_main ENDP
```
Let's load it in OllyDbg once again:

Figure 3.2: OllyDbg: last call of f()

Now the next variable is present in each frame.
Let's investigate the stack more closely. The author has again added his comments:

```
0029FC14 00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC18 00000008 1st argument: a
0029FC1C 0000000D 2nd argument: b
0029FC20 00000014 3rd argument: limit
0029FC24 0000000D "next" variable
0029FC28 /0029FC40 saved EBP register
0029FC2C |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC30 |00000005 1st argument: a
0029FC34 |00000008 2nd argument: b
0029FC38 |00000014 3rd argument: limit
0029FC3C |00000008 "next" variable
0029FC40 |0029FC58 saved EBP register
0029FC44 |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC48 |00000003 1st argument: a
0029FC4C |00000005 2nd argument: b
0029FC50 |00000014 3rd argument: limit
0029FC54 |00000005 "next" variable
0029FC58 |0029FC70 saved EBP register
0029FC5C |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC60 |00000002 1st argument: a
0029FC64 |00000003 2nd argument: b
0029FC68 |00000014 3rd argument: limit
0029FC6C |00000003 "next" variable
0029FC70 |0029FC88 saved EBP register
0029FC74 |00E0103A RETURN to fib2.00E0103A from fib2.00E01000
0029FC78 |00000001 1st argument: a \
0029FC7C |00000002 2nd argument: b | prepared in f1() for next f1() call
0029FC80 |00000014 3rd argument: limit /
0029FC84 |00000002 "next" variable
0029FC88 |0029FC9C saved EBP register
0029FC8C |00E0106C RETURN to fib2.00E0106C from fib2.00E01000
0029FC90 |00000000 1st argument: a \
0029FC94 |00000001 2nd argument: b | prepared in main() for f1()
0029FC98 |00000014 3rd argument: limit /
0029FC9C |0029FCE0 saved EBP register
0029FCA0 |00E011E0 RETURN to fib2.00E011E0 from fib2.00E01050
0029FCA4 |00000001 main() 1st argument: argc \
0029FCA8 |00812C8 main() 2nd argument: argv | prepared in CRT for main()
0029FCAC |00082940 main() 3rd argument: envp /
```

Here we see it: the `next` value is calculated in each function incarnation, then passed as argument `b` to the next incarnation.

### 3.7.3 Summary

Recursive functions are æsthetically nice, but technically may degrade performance because of their heavy stack usage. Everyone who writes performance critical code probably should avoid recursion.

For example, the author of this book once wrote a function to seek a particular node in a binary tree. As a recursive function it looked quite stylish but since additional time was spent at each function call for the prologue/epilogue, it was working a couple of times slower than an iterative (recursion-free) implementation.

By the way, that is the reason that some functional PL\(^6\) compilers (where recursion is used heavily) use tail call. We talk about tail call when a function has only one single call to itself located at the end of it, like:

```
;; factorial : number -> number
;; to calculate the product of all positive
;; integers less than or equal to n.
(define (factorial n)
  (if (= n 1)
    1
    (* n (factorial (- n 1)))))
```

\(^6\)LISP, Python, Lua, etc.
Tail call is important because compiler can rework this code easily into iterative one, to get rid of recursion.

### 3.8 CRC32 calculation example

This is a very popular table-based CRC32 hash calculation technique.

```c
/* By Bob Jenkins, (c) 2006, Public Domain */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef unsigned long ub4;
typedef unsigned char ub1;

static const ub4 crcstab[256] = {
    0x00000000, 0x77873096, 0xee0e612c, 0x990951ba, 0x0706c419,
    0x7023b48f, 0xe0e64559, 0x910135b4, 0x0b08c59f, 0x7b7bb57e,
    0x0d0f5f57, 0xd0d88f43, 0x9ad97f4e, 0x04026f53, 0x74715f38,
    0x0f0f4f2d, 0xd9287e0b, 0x962f6ef2, 0x08055e67, 0x78744e5c,
    0x00000000, 0x77873096, 0xee0e612c, 0x990951ba, 0x0706c419,
    0x7023b48f, 0xe0e64559, 0x910135b4, 0x0b08c59f, 0x7b7bb57e,
    0x0d0f5f57, 0xd0d88f43, 0x9ad97f4e, 0x04026f53, 0x74715f38,
    0x0f0f4f2d, 0xd9287e0b, 0x962f6ef2, 0x08055e67, 0x78744e5c,
```

1The source code has been taken from: [http://go.yurichev.com/17327](http://go.yurichev.com/17327)
/* how to derive the values in crctab[] from polynomial 0xedb88320 */

void build_table()
{
    ub4 i, j;
    for (i=0; i<256; ++i) {
        j = i;
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
        printf("0x%.8lx, ", j);
        if (i%6 == 5) printf("\n");
    }
}

/* the hash function */

ub4 crc(const void *key, ub4 len, ub4 hash)
{
    ub4 i;
    const ub1 *k = key;
    for (hash=len, i=0; i<len; ++i)
        hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k[i]];
    return hash;
}

/* To use, try "gcc -O crc.c -o crc; crc < crc.c */

int main()
{
    char s[1000];
    while (gets(s)) printf("%.8lx\n", crc(s, strlen(s), 0));
    return 0;
}

We are interested in the crc() function only. By the way, pay attention to the two loop initializers in the for() statement: hash=len, i=0. The C/C++ standard allows this, of course. The emitted code will contain two operations in the loop initialization part instead of one.

Let's compile it in MSVC with optimization (/Ox). For the sake of brevity, only the crc() function is listed here, with my comments.

```asm
理想位置","text":null,"layout":null,"type":null,"is_image":false,"is_diagram":false,"natural_text":null}
but other bits (8-31) are cleared all time, so it is OK
these are cleared because, as for EDI, it was done by MOVZX instruction above
high bits of EBX was cleared by AND EBX, 255 instruction above (255 = 0xff)

```
xor edi, ebx
```

EAX=EAX>>8; bits 24-31 taken from nowhere will be cleared

```
shr eax, 8
```

EAX=EAX^crctab[EDI*4] - choose EDI-th element from crctab[] table

```
xor eax, DWORD PTR _crctab[edi*4]
inc ecx ; i++
cmp ecx, edx ; i<len ?
jb SHORT $LL3@crc ; yes
pop edi
pop esi
pop ebx
$LN1@crc:
ret 0
```

ENDORP

Let’s try the same in GCC 4.4.1 with -O3 option:

```
public crc
proc near
key = dword ptr 8
hash = dword ptr 0Ch
push ebp
xor edx, edx
mov ebp, esp
push esi
mov esi, [ebp+key]
push ebx
mov ebx, [ebp+hash]
test ebx, ebx
mov eax, ebx
jz short loc_80484D3
nop ; padding
lea esi, [esi+0] ; padding; works as NOP (ESI does not change here)
loc_80484B8:
mov ecx, eax ; save previous state of hash to ECX
xor al, [esi+edx] ; AL=*(key+i)
add edx, 1 ; i++
shr ecx, 8 ; ECX=hash>>8
movzx eax, al ; EAX=*(key+i)
mov eax, dword ptr ds:crctab[eax*4] ; EAX=crctab[EAX]
xor eax, ecx ; hash=EAX^ECX
cmp ebx, edx
ja short loc_80484B8
loc_80484D3:
pop ebx
pop esi
pop ebp
retn
```

crc endp

GCC has aligned the loop start on a 8-byte boundary by adding NOP and lea esi, [esi+0] (that is an idle operation too). Read more about it in npad section (.1.7 on page 1024).

### 3.9 Network address calculation example

As we know, a TCP/IP address (IPv4) consists of four numbers in the 0…255 range, i.e., four bytes.
Four bytes can be fit in a 32-bit variable easily, so an IPv4 host address, network mask or network address can all be 32-bit integers.

From the user's point of view, the network mask is defined as four numbers and is formatted like 255.255.255.0 or so, but network engineers (sysadmins) use a more compact notation (CIDR\textsuperscript{8}), like “/8”, “/16”, etc.

This notation just defines the number of bits the mask has, starting at the MSB.

<table>
<thead>
<tr>
<th>Mask</th>
<th>Hosts</th>
<th>Usable</th>
<th>Netmask Hex mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>/30</td>
<td>4</td>
<td>2</td>
<td>255.255.255.252</td>
</tr>
<tr>
<td>/29</td>
<td>8</td>
<td>6</td>
<td>255.255.255.248</td>
</tr>
<tr>
<td>/28</td>
<td>16</td>
<td>14</td>
<td>255.255.255.240</td>
</tr>
<tr>
<td>/27</td>
<td>32</td>
<td>30</td>
<td>255.255.255.224</td>
</tr>
<tr>
<td>/26</td>
<td>64</td>
<td>62</td>
<td>255.255.255.192</td>
</tr>
<tr>
<td>/24</td>
<td>256</td>
<td>254</td>
<td>255.255.255.0</td>
</tr>
<tr>
<td>/23</td>
<td>512</td>
<td>510</td>
<td>255.255.254.0</td>
</tr>
<tr>
<td>/22</td>
<td>1024</td>
<td>1022</td>
<td>255.255.252.0</td>
</tr>
<tr>
<td>/21</td>
<td>2048</td>
<td>2046</td>
<td>255.255.248.0</td>
</tr>
<tr>
<td>/20</td>
<td>4096</td>
<td>4094</td>
<td>255.255.240.0</td>
</tr>
<tr>
<td>/19</td>
<td>8192</td>
<td>8190</td>
<td>255.255.224.0</td>
</tr>
<tr>
<td>/18</td>
<td>16384</td>
<td>16382</td>
<td>255.255.192.0</td>
</tr>
<tr>
<td>/17</td>
<td>32768</td>
<td>32766</td>
<td>255.255.128.0</td>
</tr>
<tr>
<td>/16</td>
<td>65536</td>
<td>65534</td>
<td>255.255.0.0</td>
</tr>
<tr>
<td>/8</td>
<td>16777216</td>
<td>16777214</td>
<td>255.255.0.0</td>
</tr>
</tbody>
</table>

Here is a small example, which calculates the network address by applying the network mask to the host address.

```c
#include <stdio.h>
#include <stdint.h>

uint32_t form_IP (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4)
{
    return (ip1<<24) | (ip2<<16) | (ip3<<8) | ip4;
}

void print_as_IP (uint32_t a)
{
    printf (“%d.%d.%d.%d\n”,
            (a>>24) &0xFF,
            (a>>16) &0xFF,
            (a>>8) &0xFF,
            a &0xFF);
}

// bit=31..0
uint32_t set_bit (uint32_t input, int bit)
{
    return input=input|(1<<bit);
}

uint32_t form_netmask (uint8_t netmask_bits)
{
    uint32_t netmask=0;
    uint8_t i;
    for (i=0; i<netmask_bits; i++)
        netmask=set_bit(netmask, 31-i);
    return netmask;
}

void calc_network_address (uint8_t ip1, uint8_t ip2, uint8_t ip3, uint8_t ip4, uint8_t netmask_bits)
{
    uint32_t netmask=form_netmask(netmask_bits);
    uint32_t ip=form_IP(ip1, ip2, ip3, ip4);
    uint32_t netw adr;
```

\textsuperscript{8}Classless Inter-Domain Routing
## 3.9.1 calc_network_address()

The `calc_network_address()` function is the simplest one: it just ANDs the host address with the network mask, resulting in the network address.

```assembly
; ip1$ = 8 ; size = 1
_ipl$ = 8
; size = 1
_ipl$ = 12
; size = 1
_ipl$ = 16
; size = 1
_ipl$ = 20
; size = 1
_netmask_bits$ = 24 ; size = 1
 CALC_NETWORK_ADDRESS PROC
 push edi
 push DWORD PTR _netmask_bits$[esp]
call _form_netmask
 push OFFSET $Sg3045 ; 'netmask='
 mov edi, eax
call DWORD PTR __imp__printf
 push edi
 call _print_as_IP
 push OFFSET $Sg3046 ; 'network address='
call DWORD PTR __imp__printf
 push DWORD PTR _ipl$[esp+16]
push DWORD PTR _ipl$[esp+20]
push DWORD PTR _ipl$[esp+24]
push DWORD PTR _ipl$[esp+28]
call _form_IP
 and eax, edi ; network address = host address & netmask
 push eax
call _print_as_IP
 add esp, 36
 pop edi
 ret 0
 CALC_NETWORK_ADDRESS ENDP
```

At line 22 we see the most important AND—here the network address is calculated.

### 3.9.2 form_IP()

The `form_IP()` function just puts all 4 bytes into a 32-bit value.

Here is how it is usually done:

- Allocate a variable for the return value. Set it to 0.
- Take the fourth (lowest) byte, apply OR operation to this byte and return the value. The return value contain the 4th byte now.
- Take the third byte, shift it left by 8 bits. You’ll get a value like 0x0000bb00 where bb is your third byte. Apply the OR operation to the resulting value and returning value. The return value has contained 0x000000aa so far, so ORing the values will produce a value like 0x0000bbaa.
• Take the second byte, shift it left by 16 bits. You’ll get a value like 0x00cc0000, where cc is your second byte. Apply the OR operation to the resulting value and returning value. The return value has contained 0x0000bbbaa so far, so ORing the values will produce a value like 0x00ccbbbaa.

• Take the first byte, shift it left by 24 bits. You’ll get a value like 0xdd000000, where dd is your first byte. Apply the OR operation to the resulting value and returning value. The return value has contained 0x00ccbbaa so far, so ORing the values will produce a value like 0xddccbbaa.

And this is how it’s done by non-optimizing MSVC 2012:

### Listing 3.11: Non-optimizing MSVC 2012

```assembly
; denote ip1 as "dd", ip2 as "cc", ip3 as "bb", ip4 as "aa".
_ip1$ = 8    ; size = 1
_ip2$ = 12   ; size = 1
_ip3$ = 16   ; size = 1
_ip4$ = 20   ; size = 1
_form_IP PROC
    push    ebp
    mov     ebp, esp
    movzx   eax, BYTE PTR _ip1$[ebp] ; EAX=000000dd
    shl     eax, 24 ; EAX=dd000000
    movzx   ecx, BYTE PTR _ip2$[ebp] ; ECX=000000cc
    shl     ecx, 16 ; ECX=00cc0000
    or      eax, ecx ; EAX=ddcc0000
    movzx   edx, BYTE PTR _ip3$[ebp] ; EDX=000000bb
    shl     edx, 8 ; EDX=0000bb00
    or      eax, edx ; EAX=ddccbb00
    movzx   ecx, BYTE PTR _ip4$[ebp] ; ECX=000000aa
    or      eax, ecx ; EAX=ddccbbaa
    pop     ebp
    ret     0
_form_IP ENDP
```

Well, the order is different, but, of course, the order of the operations doesn’t matter.

Optimizing MSVC 2012 does essentially the same, but in a different way:

### Listing 3.12: Optimizing MSVC 2012 /Ob0

```assembly
; denote ip1 as "dd", ip2 as "cc", ip3 as "bb", ip4 as "aa".
_ip1$ = 8    ; size = 1
_ip2$ = 12   ; size = 1
_ip3$ = 16   ; size = 1
_ip4$ = 20   ; size = 1
_form_IP PROC
    movzx   eax, BYTE PTR _ip1$[esp-4] ; EAX=000000dd
    movzx   ecx, BYTE PTR _ip2$[esp-4] ; ECX=000000cc
    shl     eax, 8 ; EAX=0000dd00
    or      eax, ecx ; EAX=0000ddcc
    movzx   ecx, BYTE PTR _ip3$[esp-4] ; ECX=000000bb
    shl     eax, 8 ; EAX=0000ddcc00
    or      eax, ecx ; EAX=0000ddccbb
    movzx   ecx, BYTE PTR _ip4$[esp-4] ; ECX=000000aa
```

487
We could say that each byte is written to the lowest 8 bits of the return value, and then the return value is shifted left by one byte at each step.

Repeat 4 times for each input byte.

That’s it! Unfortunately, there are probably no other ways to do it.

There are no popular CPUs or ISAs which has instruction for composing a value from bits or bytes. It’s all usually done by bit shifting and ORing.

3.9.3 print_as_IP()

print_as_IP() does the inverse: splitting a 32-bit value into 4 bytes.

Slicing works somewhat simpler: just shift input value by 24, 16, 8 or 0 bits, take the bits from zeroth to seventh (lowest byte), and that’s it:

Listing 3.13: Non-optimizing MSVC 2012

```assembly
;a$ = 8 ; size = 4
_print_as_IP PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
        ; EAX=ddccbb00
    or eax, ecx
        ; EAX=ddccbbbaa
    ret 0
_print_as_IP ENDP
```

Optimizing MSVC 2012 does almost the same, but without unnecessary reloading of the input value:

Listing 3.14: Optimizing MSVC 2012 /Ob0

```assembly
;a$ = 8 ; size = 4
_print_as_IP PROC
    mov ecx, DWORD PTR _a$[esp-4]
        ; ECX=ddccbbbaa
```
3.9.4 form_netmask() and set_bit()

form_netmask() makes a network mask value from CIDR notation. Of course, it would be much effective to use here some kind of a precalculated table, but we consider it in this way intentionally, to demonstrate bit shifts.

We will also write a separate function set_bit(). It’s a not very good idea to create a function for such primitive operation, but it would be easy to understand how it all works.

Listing 3.15: Optimizing MSVC 2012 /Ob0

```assembly
_input$ = 8          ; size = 4
_bit$ = 12           ; size = 4
_set_bit PROC
    mov ecx, DWORD PTR _bit$[esp-4]
    mov eax, 1
    shl eax, cl
    or eax, DWORD PTR _input$[esp-4]
    ret 0
_set_bit ENDP

_netmask_bits$ = 8   ; size = 1
_form_netmask PROC
    push ebx
    push esi
    movzx esi, BYTE PTR _netmask_bits$[esp+4]
    xor ecx, ecx
    xor bl, bl
    test esi, esi
    jle SHORT $LN9@form_netma
    xor edx, edx
$LL3@form_netma:
    mov eax, 31
    sub eax, edx
    push eax
    push ecx
    call _set_bit
    inc bl
    movzx edx, bl
    add esp, 8
    mov ecx, eax
    cmp edx, esi
```
set_bit() is primitive: it just shift left 1 to number of bits we need and then ORs it with the “input” value.
form_netmask() has a loop: it will set as many bits (starting from the MSB) as passed in the netmask_bits argument.

3.9.5 Summary

That’s it! We run it and getting:

netmask=255.255.255.0
network address=10.1.2.0
netmask=255.0.0.0
network address=10.0.0.0
netmask=255.255.255.128
network address=10.1.2.0
netmask=255.255.255.192
network address=10.1.2.64

3.10 Loops: several iterators

In most cases loops have only one iterator, but there could be several in the resulting code.

Here is a very simple example:

```c
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;
    // copy from one array to another in some weird scheme
    for (i=0; i<cnt; i++)
        a1[i*3]=a2[i*7];
}
```

There are two multiplications at each iteration and they are costly operations. Can we optimize it somehow?

Yes, if we notice that both array indices are jumping on values that we can easily calculate without multiplication.

3.10.1 Three iterators

```c
f PROC
; RCX=a1
; RDX=a2
; R8=cnt
    test r8, r8          ; cnt==0? exit then
    je SHORT $LN1@f
    npad 11
$LL3@f:
    mov eax, DWORD PTR [rdx]
    lea rcx, QWORD PTR [rcx+12]
    lea rdx, QWORD PTR [rdx+28]
    mov DWORD PTR [rcx-12], eax
    dec r8
    jne SHORT $LL3@f
$LN1@f:
```
Now there are 3 iterators: the \textit{cnt} variable and two indices, which are increased by 12 and 28 at each iteration. We can rewrite this code in C/C++:

```c
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t i;
    size_t idx1=0; idx2=0;

    // copy from one array to another in some weird scheme
    for (i=0; i<cnt; i++)
    {
        a1[idx1]=a2[idx2];
        idx1+=3;
        idx2+=7;
    }
}
```

So, at the cost of updating 3 iterators at each iteration instead of one, we can remove two multiplication operations.

### 3.10.2 Two iterators

GCC 4.9 does even more, leaving only 2 iterators:

```c
Listing 3.17: Optimizing GCC 4.9 x64

; RDI=a1
; RSI=a2
; RDX=cnt
f:
    test    rdx, rdx ; cnt==0? exit then
je .L1
    ; calculate last element address in "a2" and leave it in RDX
    lea    rax, [0+rdx*4]
    ; RAX=RDX*4=cnt*4
    sal    rdx, 5
    ; RDX=RDX<<5=cnt*32
    sub    rdx, rax
    ; RDX=RDX-RAX=cnt*32-cnt*4=cnt*28
    add    rdx, rsi
    ; RDX=RDX+RSI=a2+cnt*28
.L3:
    mov    eax, DWORD PTR [rsi]
    add    rsi, 28
    add    rdi, 12
    mov    DWORD PTR [rdi-12], eax
    cmp    rsi, rdx
    jne    .L3
.L1:
    rep ret
```

There is no \textit{counter} variable any more: GCC concluded that it is not needed.

The last element of the \textit{a2} array is calculated before the loop begins (which is easy: \textit{cnt*7}) and that's how the loop is to be stopped: just iterate until the second index reaches this precalculated value.

You can read more about multiplication using shifts/additions/subtractions here: \textbf{1.24.1 on page 212}. This code can be rewritten into C/C++ like that:

```c
#include <stdio.h>

void f(int *a1, int *a2, size_t cnt)
{
    size_t idx1=0; idx2=0;
```

491
size_t last_idx2 = cnt * 7;

// copy from one array to another in some weird scheme
for (;;) {
    a1[idx1] = a2[idx2];
    idx1 += 3;
    idx2 += 7;
    if (idx2 == last_idx2)
        break;
}

GCC (Linaro) 4.9 for ARM64 does the same, but it precalculates the last index of a1 instead of a2, which, of course has the same effect:

Listing 3.18: Optimizing GCC (Linaro) 4.9 ARM64

GCC 4.4.5 for MIPS does the same:

Listing 3.19: Optimizing GCC 4.4.5 for MIPS (IDA)
3.10.3 Intel C++ 2011 case

Compiler optimizations can also be weird, but nevertheless, still correct. Here is what the Intel C++ compiler 2011 does:

Listing 3.20: Optimizing Intel C++ 2011 x64

```assembly
f PROC
; parameter 1: rcx = a1
; parameter 2: rdx = a2
; parameter 3: r8 = cnt
.B1.1::
    test r8, r8
    jbe exit

.B1.2::
    cmp r8, 6
    jbe just_copy

.B1.3::
    cmp rcx, rdx
    jbe .B1.5

.B1.4::
    mov r10, r8
    mov r9, rcx
    shl r10, 5
    lea rax, QWORD PTR [r8*4]
    sub r9, rdx
    sub r10, rax
    cmp r9, r10
    jge just_copy2

.B1.5::
    cmp rdx, rcx
    jbe just_copy

.B1.6::
    mov r9, rdx
    lea rax, QWORD PTR [r8*8]
    sub r9, rcx
    lea r10, QWORD PTR [rax+r8*4]
    cmp r9, r10
    jl just_copy2

just_copy2::
    ; R8 = cnt
    ; RDX = a2
    ; RCX = a1
    xor r10d, r10d
    xor r9d, r9d
    xor eax, eax

.B1.8::
    mov r11d, DWORD PTR [rax+rdx]
    inc r10
    mov DWORD PTR [r9+rcx], r11d
    add r9, 12
    add rax, 28
    cmp r10, r8
    jb .B1.8
    jmp exit

just_copy::
    ; R8 = cnt
    ; RDX = a2
    ; RCX = a1
    xor r10d, r10d
    xor r9d, r9d
    xor eax, eax
```
First, there are some decisions taken, then one of the routines is executed. Looks like it is a check if arrays intersect.

This is very well known way of optimizing memory block copy routines. But copy routines are the same! This is has to be an error of the Intel C++ optimizer, which still produces workable code, though.

We intentionally considering such example code in this book so the reader would understand that compiler output is weird at times, but still correct, because when the compiler was tested, it passed the tests.

### 3.11 Duff’s device

Duff’s device is an unrolled loop with the possibility to jump to the middle of it. The unrolled loop is implemented using a fallthrough switch() statement. We would use here a slightly simplified version of Tom Duff’s original code. Let’s say, we have to write a function that clears a region in memory. One can come with a simple loop, clearing byte by byte. It’s obviously slow, since all modern computers have much wider memory bus. So the better way is to clear the memory region using 4 or 8 bytes blocks. Since we are going to work with a 64-bit example here, we are going to clear the memory in 8 bytes blocks. So far so good. But what about the tail? Memory clearing routine can also be called for regions of size that’s not a multiple of 8. So here is the algorithm:

- calculate the number of 8-bytes blocks, clear them using 8-bytes (64-bit) memory accesses;
- calculate the size of the tail, clear it using 1-byte memory accesses.

The second step can be implemented using a simple loop. But let’s implement it as an unrolled loop:

```c
#include <stdint.h>
#include <stdio.h>

void bzero(uint8_t* dst, size_t count)
{
    int i;

    if (count&(~7)) // work out 8-byte blocks
        for (i=0; i<count>>3; i++)
        {
            *(uint64_t*)dst=0;
            dst=dst+8;
        }

    // work out the tail
    switch(count & 7)
    {
        case 7: *dst++ = 0;
        case 6: *dst++ = 0;
        case 5: *dst++ = 0;
        case 4: *dst++ = 0;
        case 3: *dst++ = 0;
        case 2: *dst++ = 0;
        case 1: *dst++ = 0;
        case 0: // do nothing
            break;
    }
}
```
Let’s first understand how the calculation is performed. The memory region size comes as a 64-bit value. And this value can be divided in two parts:

```
7 6 5 4 3 2 1 0
```

( “B” is number of 8-byte blocks and “S” is length of the tail in bytes ).

When we divide the input memory region size by 8, the value is just shifted right by 3 bits. But to calculate the remainder, we can just to isolate the lowest 3 bits! So the number of 8-byte blocks is calculated as \( \text{count} \gg 3 \) and remainder as \( \text{count} \& 7 \). We also have to find out if we are going to execute the 8-byte procedure at all, so we need to check if the value of \( \text{count} \) is greater than 7. We do this by clearing the 3 lowest bits and comparing the resulting number with zero, because all we need here is to answer the question, is the high part of \( \text{count} \) non-zero. Of course, this works because 8 is \( 2^3 \) and division by numbers that are \( 2^n \) is easy. It’s not possible for other numbers. It’s actually hard to say if these hacks are worth using, because they lead to hard-to-read code. However, these tricks are very popular and a practicing programmer, even if he/she is not using them, nevertheless has to understand them.

So the first part is simple: get the number of 8-byte blocks and write 64-bit zero values to memory. The second part is an unrolled loop implemented as fallthrough switch() statement.

First, let’s express in plain English what we have to do here.

We have to “write as many zero bytes in memory, as \( \text{count} \& 7 \) value tells us”. If it’s 0, jump to the end, there is no work to do. If it’s 1, jump to the place inside switch() statement where only one storage operation is to be executed. If it’s 2, jump to another place, where two storage operation are to be executed, etc. 7 as input value leads to the execution of all 7 operations. There is no 8, because a memory region of 8 bytes is to be processed by the first part of our function. So we wrote an unrolled loop. It was definitely faster on older computers than normal loops (and conversely, latest CPUs works better for short loops than for unrolled ones). Maybe this is still meaningful on modern low-cost embedded MCU’s.

Let’s see what the optimizing MSVC 2012 does:

```assembly
dst$ = 8
count$ = 16
bzero PROC
    test rdx, -8
    je SHORT $LN11@bzero
; work out 8-byte blocks
    xor r10d, r10d
    mov r9, rdx
    shr r9, 3
    mov r8d, r10d
    test r9, r9
    je SHORT $LN11@bzero
    npad 5
$LL19@bzero:
    inc r8d
    mov QWORD PTR [rcx], r10
    add rcx, 8
    movsx dax, r8d
    cmp rax, r9
    jb SHORT $LL19@bzero
$LN11@bzero:
; work out the tail
    and edx, 7
    dec rdx
    cmp rdx, 6
    ja SHORT $LN9@bzero
    lea r8, OFFSET FLAT: ImageBase
    mov eax, DWORD PTR $LN22@bzero[r8+rdx*4]
    add rax, r8
    jmp rax
$LN8@bzero:
    mov BYTE PTR [rcx], 0
    inc rcx
$LN7@bzero:
    mov BYTE PTR [rcx], 0
    inc rcx
```

Microcontroller Unit
The first part of the function is predictable. The second part is just an unrolled loop and a jump passing control flow to the correct instruction inside it. There is no other code between the MOV/INC instruction pairs, so the execution is to fall until the very end, executing as many pairs as needed. By the way, we can observe that the MOV/INC pair consumes a fixed number of bytes (3+3). So the pair consumes 6 bytes. Knowing that, we can get rid of the switch() jumptable, we can just multiple the input value by 6 and jump to current RIP + input_value * 6.

This can also be faster because we are not in need to fetch a value from the jumptable.

It’s possible that 6 probably is not a very good constant for fast multiplication and maybe it’s not worth it, but you get the idea.

That is what old-school demomakers did in the past with unrolled loops.

### 3.11.1 Should one use unrolled loops?

Unrolled loops can have benefits if there is no fast cache memory between RAM and CPU, and the CPU, in order to get the code of the next instruction, must load it from RAM each time. This is a case of modern low-cost MCU and old CPUs.

Unrolled loops are slower than short loops if there is a fast cache between RAM and CPU and the body of loop can fit into cache, and CPU will load the code from it not touching the RAM. Fast loops are the loops which body’s size can fit into L1 cache, but even faster loops are those small ones which can fit into micro-operation cache.

### 3.12 Division using multiplication

A very simple function:

```c
int f(int a) {
    return a/9;
};
```

#### 3.12.1 x86

...is compiled in a very predictable way:

---

10 As an exercise, you can try to rework the code to get rid of the jumptable. The instruction pair can be rewritten in a way that it will consume 4 bytes or maybe 8. 1 byte is also possible (using STOSB instruction).
Listing 3.21: MSVC

```assembly
; a$ = 8 ; size = 4
.f PROC
    push ebp
    mov ebp, esp
    mov eax, DWORD PTR _a$[ebp]
    cdq ; sign extend EAX to EDX:EAX
    mov ecx, 9
    idiv ecx
    pop ebp
    ret 0
.f ENDP
```

IDIV divides the 64-bit number stored in the EDX:EAX register pair by the value in the ECX. As a result, EAX will contain the quotient, and EDX—the remainder. The result is returned from the f() function in the EAX register, so the value is not moved after the division operation, it is in right place already.

Since IDIV uses the value in the EDX:EAX register pair, the CDQ instruction (before IDIV) extends the value in EAX to a 64-bit value taking its sign into account, just as MOVSD does.

If we turn optimization on (/Ox), we get:

Listing 3.22: Optimizing MSVC

```assembly
; a$ = 8 ; size = 4
._f PROC
    mov ecx, DWORD PTR _a$[esp-4]
    mov eax, 954437177 ; 38E38E39H
    imul ecx
    sar edx, 1
    mov eax, edx
    shr eax, 31 ; 0000001fH
    add eax, edx
    ret 0
._f ENDP
```

This is division by multiplication. Multiplication operations work much faster. And it is possible to use this trick\(^\text{11}\) to produce code which is effectively equivalent and faster.

This is also called “strength reduction” in compiler optimizations.

GCC 4.4.1 generates almost the same code even without additional optimization flags, just like MSVC with optimization turned on:

Listing 3.23: Non-optimizing GCC 4.4.1

```assembly
public f
f proc near
arg_0 = dword ptr 8
    push ebp
    mov ebp, esp
    mov ecx, [ebp+arg_0]
    mov edx, 954437177 ; 38E38E39h
    mov eax, ecx
    imul edx
    sar edx, 1
    mov eax, ecx
    sar eax, 1fh
    mov ecx, edx
    sub ecx, eax
    mov eax, ecx
    pop ebp
    ret
f endp
```

\(^\text{11}\)Read more about division by multiplication in [Henry S. Warren, Hacker’s Delight, (2002)10-3]
3.12.2 How it works

From school-level mathematics, we can remember that division by 9 can be replaced by multiplication by \( \frac{1}{9} \). In fact, sometimes compilers do so for floating-point arithmetics, for example, FDIV instruction in x86 code can be replaced by FMUL. At least MSVC 6.0 will replace division by 9 by multiplication by 0.111111... and sometimes it’s hard to be sure, what operation was in the original source code.

But when we operate over integer values and integer CPU registers, we can’t use fractions. However, we can rework fraction like that:

\[
\text{result} = \frac{x}{9} = x \cdot \frac{1}{9} = x \cdot \frac{1}{9} \cdot \text{MagicNumber}
\]

Given the fact that division by \( 2^n \) is very fast (using shifts), we now should find that MagicNumber, for which the following equation will be true:

\[
2^n = 9 \cdot \text{MagicNumber}
\]

Division by \( 2^{32} \) is somewhat hidden: lower 32-bit of product in EAX is not used (dropped), only higher 32-bit of product (in EDX) is used and then shifted by additional 1 bit.

In other words, the assembly code we have just seen multiplicates by \( \frac{954437177}{2^{32+1}} \), or divides by \( 954437177 \cdot 2^{32+1} \).

To find a divisor we just have to divide numerator by denominator. Using Wolfram Alpha, we can get 8.99999999.... as result (which is close to 9).

Read more about it in [Henry S. Warren, Hacker’s Delight, (2002)10-3].

Many people miss “hidden” division by \( 2^{32} \) or \( 2^{64} \), when lower 32-bit part (or 64-bit part) of product is not used. This is why division by multiplication is difficult to understand at the beginning.

Mathematics for Programmers\(^{12}\) has yet another explanation.

3.12.3 ARM

The ARM processor, just like in any other “pure” RISC processor lacks an instruction for division. It also lacks a single instruction for multiplication by a 32-bit constant (recall that a 32-bit constant cannot fit into a 32-bit opcode).

By taking advantage of this clever trick (or hack), it is possible to do division using only three instructions: addition, subtraction and bit shifts (1.28 on page 303).

Here is an example that divides a 32-bit number by 10, from [Advanced RISC Machines Ltd, The ARM Cookbook, (1994)3.3 Division by a Constant]. The output consists of the quotient and the remainder.

```asm
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
SUB  a2, a1, #10 ; keep (x-10) for later
SUB  a1, a1, a1, lsr #2
ADD  a1, a1, a1, lsr #4
ADD  a1, a1, a1, lsr #8
ADD  a1, a1, a1, lsr #16
MOV  a1, a1, lsr #3
ADD  a3, a1, a1, asl #2
SUBS a2, a2, a3, asl #1 ; calc (x-10) - (x/10)*10
ADDP L a1, a1, #1 ; fix-up quotient
ADDMI a2, a2, #10 ; fix-up remainder
MOV  pc, lr
```

Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

```
__text:00002C58 39 1E 08 E3 E3 18 43 E3 MOV  R1, 0x38E38E39
__text:00002C60 10 F1 50 E7 SMMUL R0, R0, R1
__text:00002C64 C0 10 A0 E1 MOV  R1, R0,ASR#1
__text:00002C68 A0 0F 81 E0 ADD  R0, R1, R0,LSR#31
__text:00002C6C 1E FF 2F E1 BX  LR
```

This code is almost the same as the one generated by the optimizing MSVC and GCC.

Apparently, LLVM uses the same algorithm for generating constants.

\(^{12}\)https://yurichev.com/writings/Math-for-programmers.pdf
The observant reader may ask, how does MOV writes a 32-bit value in a register, when this is not possible in ARM mode.

it is impossible indeed, but, as we see, there are 8 bytes per instruction instead of the standard 4, in fact, there are two instructions.

The first instruction loads 0x8E39 into the low 16 bits of register and the second instruction is MOV, it loads 0x383E into the high 16 bits of the register. IDA is fully aware of such sequences, and for the sake of compactness reduces them to one single “pseudo-instruction”.

The SMMUL (Signed Most Significant Word Multiply) instruction two multiplies numbers, treating them as signed numbers and leaving the high 32-bit part of result in the R0 register, dropping the low 32-bit part of the result.

The “MOV R1, R0,ASR#1” instruction is an arithmetic shift right by one bit.

“ADD R0, R1, R0,LSR#31” is $R0 = R1 + R0 >> 31$

There is no separate shifting instruction in ARM mode. Instead, an instructions like (MOV, ADD, SUB, RSB)\(^{13}\) can have a suffix added, that says if the second operand must be shifted, and if yes, by what value and how. ASR stands for Arithmetic Shift Right, LSR—Logical Shift Right.

### Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>R1, 0x38E38E39</td>
</tr>
<tr>
<td>SMMUL.W</td>
<td>R0, R0, R1</td>
</tr>
<tr>
<td>ASRS</td>
<td>R1, R0, #1</td>
</tr>
<tr>
<td>ADD.W</td>
<td>R0, R1, R0,LSR#31</td>
</tr>
<tr>
<td>BX</td>
<td>LR</td>
</tr>
</tbody>
</table>

There are separate instructions for shifting in Thumb mode, and one of them is used here—ASRS (arithmetic shift right).

### Non-optimizing Xcode 4.6.3 (LLVM) and Keil 6/2013

Non-optimizing LLVM does not generate the code we saw before in this section, but instead inserts a call to the library function \_divsi3\.

What about Keil: it inserts a call to the library function \_aeabi_idivmod\ in all cases.

### 3.12.4 MIPS

For some reason, optimizing GCC 4.4.5 generate just a division instruction:

### Listing 3.24: Optimizing GCC 4.4.5 (IDA)

```assembly
f:
    li $v0, 9
    bnez $v0, loc_10
    div $a0, $v0; branch delay slot
    break 0x1C00; "break 7" in assembly output and objdump

loc_10:
    mflo $v0
    jr $ra
    or $at, $zero; branch delay slot, NOP
```

Here we see here a new instruction: BREAK. It just raises an exception.

In this case, an exception is raised if the divisor is zero (it’s not possible to divide by zero in conventional math).

But GCC probably did not do very well the optimization job and did not see that $V0 is never zero.

So the check is left here. So if $V0 is zero somehow, BREAK is to be executed, signaling to the OS about the exception.

Otherwise, MFLO executes, which takes the result of the division from the LO register and copies it in $V0.

\(^{13}\)These instructions are also called “data processing instructions”
By the way, as we may know, the MUL instruction leaves the high 32 bits of the result in register HI and the low 32 bits in register LO.

DIV leaves the result in the LO register, and remainder in the HI register.

If we alter the statement to “a % 9”, the MFHI instruction is to be used here instead of MFLO.

### 3.12.5 Exercise

- [http://challenges.re/27](http://challenges.re/27)

### 3.13 String to number conversion (atoi())

Let’s try to reimplement the standard atoi() C function.

#### 3.13.1 Simple example

Here is the simplest possible way to read a number represented in ASCII encoding.

It’s not error-prone: a character other than a digit leads to incorrect result.

```c
#include <stdio.h>

int my_atoi (char *s)
{
    int rt=0;
    while (*s)
    {
        rt=rt*10 + (*s-'0');
        s++;
    }
    return rt;
}

int main()
{
    printf "%d\n", my_atoi ("1234");
    printf "%d\n", my_atoi ("1234567890");
}
```

So what the algorithm does is just reading digits from left to right.

The zero ASCII character is subtracted from each digit.

The digits from “0” to “9” are consecutive in the ASCII table, so we do not even need to know the exact value of the “0” character.

All we have to know is that “0” minus “0” is 0, “9” minus “0” is 9 and so on.

Subtracting “0” from each character results in a number from 0 to 9 inclusive.

Any other character leads to an incorrect result, of course!

Each digit has to be added to the final result (in variable “rt”), but the final result is also multiplied by 10 at each digit.

In other words, the result is shifted left by one position in decimal form on each iteration.

The last digit is added, but there is no shift.

### Optimizing MSVC 2013 x64

Listing 3.25: Optimizing MSVC 2013 x64

```
s$ = 8
my_atoi PROC
; load first character
    movzx r8d, BYTE PTR [rcx]
```
EAX is allocated for "rt" variable
its 0 at start
xor eax, eax
; first character is zero-byte, i.e., string terminator?
; exit then.
test r8b, r8b
je SHORT $LN9@my_atoi

$LL2@my_atoi:
  lea edx, DWORD PTR [rax+rax*4]
  ; EDX=RAX+RAX*4=rt+rt*4=rt*5
  movsx eax, r8b
  ; EAX=input character
  ; load next character to R8D
  movzx r8d, BYTE PTR [rcx+1]
  ; shift pointer in RCX to the next character:
  lea rcx, QWORD PTR [rcx+1]
  lea eax, DWORD PTR [rax+rdx*2]
  ; EAX=RAX+RDX*2=input character + rt*5*2=input character + rt*10
  ; correct digit by subtracting 48 (0x30 or '0')
  add eax, -48
  ; was the last character zero?
  test dl, dl
  jne SHORT $LL2@my_atoi

$LN9@my_atoi:
  ret 0
my_atoi ENDP

A character can be loaded in two places: the first character and all subsequent characters. This is arranged so for loop regrouping.

There is no instruction for multiplication by 10, two LEA instruction do this instead.

MSVC sometimes uses the ADD instruction with a negative constant instead of SUB. This is the case.

It’s very hard to say why this is better then SUB. But MSVC does this often.

Optimizing GCC 4.9.1 x64

Optimizing GCC 4.9.1 is more concise, but there is one redundant RET instruction at the end. One would be enough.

Listing 3.26: Optimizing GCC 4.9.1 x64

my_atoi:
  ; load input character into EDX
  movsx edx, BYTE PTR [rdi]
  ; EAX is allocated for "rt" variable
  xor eax, eax
  ; exit, if loaded character is null byte
  test dl, dl
  je .L4
  .L3:
  lea eax, [rax+rax*4]
  ; EAX=RAX*5=rt*5
  ; shift pointer to the next character:
  add rdi, 1
  lea eax, [rdx-48+rax*2]
  ; EAX=input character - 48 + RAX*2 = input character - '0' + rt*10
  ; load next character:
  movsx edx, BYTE PTR [rdi]
  ; goto loop begin, if loaded character is not null byte
  test dl, dl
  jne .L3
  .L4:
  ret
  rep ret

Optimizing Keil 6/2013 (ARM mode)
my_atoi PROC
 ; R1 will contain pointer to character
 MOV  r1,r0
 ; R0 will contain "rt" variable
 MOV  r0,#0
 B    |L0.12|
|L0.12|
 ADD  r0,r0,r0,LSL #2
 R0=R0+R0<<2=rt*5
 ADD  r0,r2,r0,LSL #1
 R0=input character + rt*5<<1 = input character + rt*10
 ; correct whole thing by subtracting '0' from rt:
 SUB  r0,r0,#0x30
 ; shift pointer to the next character:
 ADD  r1,r1,#1
 |L0.28|
 ; load input character to R2
 LDRB  r2,[r1,#0]
 ; is it null byte? if no, jump to loop body.
 CMP  r2,#0
 BNE  |L0.12|
 ; exit if null byte.
 ; "rt" variable is still in R0 register, ready to be used in caller function
 BX   lr
 ENDP

Interestingly, from school mathematics we may recall that the order of addition and subtraction operations doesn't matter.
That's our case: first, the rt∗10−'0' expression is computed, then the input character value is added to it.
Indeed, the result is the same, but the compiler did some regrouping.

Optimizing GCC 4.9.1 ARM64
The ARM64 compiler can use the pre-increment instruction suffix:
Listing 3.29: Optimizing GCC 4.9.1 ARM64

my_atoi:
; load input character into W1
ldrb w1, [x0]
mov x2, x0
; X2=address of input string
; is loaded character zero?
; jump to exit if its so
; W1 will contain 0 in this case.
; it will be reloaded into W0 at L4.
cbz w1, .L4
; W0 will contain "rt" variable
; initialize it at zero:
mov w0, 0
.L3:
; subtract 48 or '0' from input variable and put result into W3:
sub w3, w1, #48
; load next character at address X2+1 into W1 with pre-increment:
ldrb w1, [x2,1]!
add w0, w0, w0, lsl 2
; W0=W0+W0<<2=W0+W0*4=rt*5
add w0, w3, w0, lsl 1
; W0=input digit + W0<<1 = input digit + rt*5*2 = input digit + rt*10
; if the character we just loaded is not null byte, jump to the loop begin
cbnz w1, .L3
; variable to be returned (rt) is in W0, ready to be used in caller function
ret
.L4:
mov w0, w1
ret

3.13.2 A slightly advanced example

My new code snippet is more advanced, now it checks for the “minus” sign at the first character and reports an error if a non-digit has been found in the input string:

#include <stdio.h>

int my_atoi (char *s)
{
    int negative=0;
    int rt=0;

    if (*s=='-')
    {
        negative=1;
        s++;
    };

    while (*s)
    {
        if (*s<='0' || *s>'9')
        {
            printf ("Error! Unexpected char: '%c'\n", *s);
            exit(0);
        }
        rt=rt*10 + (*s-'0');
        s++;
    }

    if (negative)
        return -rt;
    return rt;
}

int main()
{
    printf ("%d\n", my_atoi ("1234"));
    printf ("%d\n", my_atoi ("1234567890"));
printf ("%d\n", my_atoi ("-1234"));
printf ("%d\n", my_atoi ("-1234567890"));
printf ("%d\n", my_atoi ("-a1234567890")); /* error */

**Optimizing GCC 4.9.1 x64**

Listing 3.30: Optimizing GCC 4.9.1 x64

```assembly
.LC0:
    .string "Error! Unexpected char: '\%c'\n"

my_atoi:
    sub    rsp, 8
    movsx  edx, BYTE PTR [rdi]
    ; check for minus sign
    cmp    dl, 45 ; '\-
    je .L22
    xor    esi, esi
    test   dl, dl
    je .L20

.L10:
    ; ESI=0 here if there was no minus sign and 1 if it was
    lea    eax, [rdx-48]
    ; any character other than digit will result in unsigned number greater than 9 after subtraction
    ; so if it is not digit, jump to L4, where error will be reported:
    cmp    al, 9
    ja .L4
    xor    eax, eax
    jmp .L6

.L7:
    lea    ecx, [rdx-48]
    cmp    cl, 9
    ja .L4

.L6:
    lea    eax, [rax+rax*4]
    add    rdi, 1
    lea    eax, [rdx-48+rax*2]
    movsx  edx, BYTE PTR [rdi]
    test   dl, dl
    jne .L7
    ; if there was no minus sign, skip NEG instruction
    ; if it was, execute it.
    test   esi, esi
    je .L18
    neg    eax

.L18:
    add    rsp, 8
    ret

.L22:
    movsx  edx, BYTE PTR [rdi+1]
    lea    rax, [rdi+1]
    test   dl, dl
    je .L20
    mov    rdi, rax
    mov    esi, 1
    jmp .L10

.L20:
    xor    eax, eax
    jmp .L18

.L4:
    ; report error. character is in EDX
    mov    edi, 1
    mov    esi, OFFSET FLAT:.LC0 ; "Error! Unexpected char: '\%c'\n"
    xor    eax, eax
    call __printf_chk
    xor    edi, edi
    call exit
```
If the “minus” sign has been encountered at the string start, the `NEG` instruction is to be executed at the end. It just negates the number.

There is one more thing that needs mentioning.

How would a common programmer check if the character is not a digit? Just how we have it in the source code:

```c
if (*s<='0' || *s>'9')
    ...
```

There are two comparison operations.

What is interesting is that we can replace both operations by single one: just subtract “0” from character value, treat result as unsigned value (this is important) and check if it’s greater than 9.

For example, let’s say that the user input contains the dot character (”.") which has ASCII code 46.

\[46 - 48 = -2\] if we treat the result as a signed number.

Indeed, the dot character is located two places earlier than the “0” character in the ASCII table. But it is \(0xFFFFFFFE\) (4294967294) if we treat the result as an unsigned value, and that’s definitely bigger than 9!

The compilers do this often, so it’s important to recognize these tricks.

Another example of it in this book: 3.19.1 on page 534.

Optimizing MSVC 2013 x64 does the same tricks.

**Optimizing Keil 6/2013 (ARM mode)**

```
my_atoi PROC
    PUSH {r4-r6,lr}
    MOV r4,r0
    LDRB r0,[r0,#0]
    MOV r6,#0
    MOV r5,r6
    CMP r0,#0x2d '
'; R6 will contain 1 if minus was encountered, 0 if otherwise
    MOVEQ r6,#1
    ADDEQ r4,r4,#1
    B |L0.80|
    |L0.36|
    SUB r0,r1,#0x30
    CMP r0,#0xa
    BCC |L0.64|
    ADR r0,|L0.220|
    BL __2printf
    MOV r0,#0
    BL exit
    |L0.64|
    LDRB r0,[r4],#1
    ADD r1,r5,r5,LSL #2
    ADD r0,r0,r1,LSL #1
    SUB r5,r0,#0x30
    |L0.80|
    LDRB r1,[r4,#0]
    CMP r1,#0
    BNE |L0.36|
    CMP r6,#0
    ; negate result
    RSBNE r0,r5,#0
    MOVEQ r0,r5
    POP {r4-r6,pc}
ENDP
```

DCB "Error! Unexpected char: ‘%c’n",0
There is no NEG instruction in 32-bit ARM, so the “Reverse Subtraction” operation (line 31) is used here. It is triggered if the result of the CMP instruction (at line 29) has been “Not Equal” (hence -NE suffix). So what RSBNE does is to subtract the resulting value from 0. It works just like the regular subtraction operation, but swaps operands. Subtracting any number from 0 results in negation: 0 − x = −x.

Thumb mode code is mostly the same.

GCC 4.9 for ARM64 can use the NEG instruction, which is available in ARM64.

3.13.3 Exercise

Oh, by the way, security researchers deals often with unpredictable behavior of program while handling of incorrect data.

For example, while fuzzing. As an exercise, you may try to enter non-digit characters and see what happens.

Try to explain, what happened and why.

3.14 Inline functions

Inlined code is when the compiler, instead of placing a call instruction to a small or tiny function, just places its body right in-place.

Listing 3.32: A simple example

```
#include <stdio.h>

int celsius_to_fahrenheit (int celsius)
{
    return celsius * 9 / 5 + 32;
};

int main(int argc, char *argv[])
{
    int celsius=atol(argv[1]);
    printf (“%d
”, celsius_to_fahrenheit (celsius));
};
```

...is compiled in very predictable way, however, if we turn on GCC optimizations (-O3), we’ll see:

Listing 3.33: Optimizing GCC 4.8.1

```
_main:
    push   ebp
    mov    ebp, esp
    and    esp, -16
    sub    esp, 16
    call   __main
    mov    eax, DWORD PTR [ebp+12]
    mov    eax, DWORD PTR [eax+4]
    mov    DWORD PTR [esp], eax
    call   _atol
    mov    edx, 1717986919
    mov    DWORD PTR [esp], OFFSET FLAT:LC2 ; "%d
"
    lea    ecx, [eax+eax*8]
    mov    eax, ecx
    imul   edx
    sar    ecx, 31
    sar    edx
    sub    edx, ecx
    add    edx, 32
    mov    DWORD PTR [esp+4], edx
    call   _printf
    leave
    ret
```
Yes, our small function `celsius_to_fahrenheit()` has just been placed before the `printf()` call. Why? It can be faster than executing this function's code plus the overhead of calling/returning.

Modern optimizing compilers are choosing small functions for inlining automatically. But it's possible to force compiler additionally to inline some function, if to mark it with the “inline” keyword in its declaration.

### 3.14.1 Strings and memory functions

Another very common automatic optimization tactic is the inlining of string functions like `strcpy()`, `strcmp()`, `strlen()`, `memset()`, `memcmp()`, `memcpy()`, etc.

Sometimes it's faster than to call a separate function.

These are very frequent patterns and it is highly advisable for reverse engineers to learn to detect automatically.

#### `strcmp()`

```c
bool is_bool (char *s)
{
    if (strcmp (s, "true") == 0)
        return true;
    if (strcmp (s, "false") == 0)
        return false;
    assert(0);
}
```

Listing 3.34: `strcmp()` example

Listing 3.35: Optimizing GCC 4.8.1

```asm
.LC0:
.string "true"
.LC1:
.string "false"
is_bool:
.LFB0:
    push    edi
    mov     ecx, 5
    push    esi
    mov     edi, OFFSET FLAT:.LC0
    sub     esp, 20
    mov     esi, DWORD PTR [esp+32]
    repz cmpsb
    je     .L3
    mov     esi, DWORD PTR [esp+32]
    mov     ecx, 6
    mov     edi, OFFSET FLAT:.LC1
    repz cmpsb
    seta    cl
    setb    dl
    xor     eax, eax
    cmp     cl, dl
    jne     .L8
    add     esp, 20
    pop     esi
    pop     edi
    ret
.L8:
    mov     DWORD PTR [esp], 0
    call    assert
    add     esp, 20
    pop     esi
    pop     edi
    ret
.L3:
    add     esp, 20
```

(Here the division is performed by multiplication (3.12 on page 496).)
Listing 3.36: Optimizing MSVC 2010

SG3454 DB 'true', 00H
SG3456 DB 'false', 00H

_s$ = 8 ; size = 4

?is_bool@YA NPADZ PROC ; is bool
push esi
mov esi, DWORD PTR _s$[esp]
mov ecx, OFFSET $SG3454 ; 'true'
mov eax, esi
npad 4 ; align next label

$LL6@is_bool:
mov dl, BYTE PTR [eax]
cmp dl, BYTE PTR [ecx]
jne SHORT $LN7@is_bool
test dl, dl
je SHORT $LN8@is_bool
mov dl, BYTE PTR [eax+1]
cmp dl, BYTE PTR [ecx+1]
jne SHORT $LN7@is_bool
add eax, 2
add ecx, 2
test dl, dl
jne SHORT $LL6@is_bool

$LN8@is_bool:
xor eax, eax
jmp SHORT $LN9@is_bool

$LN7@is_bool:
sbb eax, eax
sbb eax, -1

$LN9@is_bool:
test eax, eax
jne SHORT $LN2@is_bool

mov al, 1
pop esi

ret 0

$LN2@is_bool:
mov ecx, OFFSET $SG3456 ; 'false'
mov eax, esi

$LL10@is_bool:
mov dl, BYTE PTR [eax]
cmp dl, BYTE PTR [ecx]
jne SHORT $LN11@is_bool
test dl, dl
je SHORT $LN12@is_bool
mov dl, BYTE PTR [eax+1]
cmp dl, BYTE PTR [ecx+1]
jne SHORT $LN11@is_bool
add eax, 2
add ecx, 2
test dl, dl
jne SHORT $LL10@is_bool

$LN12@is_bool:
xor eax, eax
jmp SHORT $LN13@is_bool

$LN11@is_bool:
sbb eax, eax
sbb eax, -1

$LN13@is_bool:
test eax, eax
jne SHORT $LN1@is_bool
xor     al, al
pop     esi
ret     0

$LN1@is_bool:
push    11
push    OFFSET $SG3458
push    OFFSET $SG3459
call    DWORD PTR __imp___wassert
add     esp, 12
pop     esi
ret     0

?is_bool@@YA_NPAD@Z ENDP ; is_bool

strlen()

Listing 3.37: strlen() example

int strlen_test(char *s1)
{
    return strlen(s1);
};

Listing 3.38: Optimizing MSVC 2010

_s1$ = 8 ; size = 4
_strlen_test PROC
    mov    eax, DWORD PTR _s1$[esp-4]
    lea    edx, DWORD PTR [eax+1]
$LL3@strlen_tes:
    mov    cl, BYTE PTR [eax]
    inc    eax
    test   cl, cl
    jne SHORT $LL3@strlen_tes
    sub    eax, edx
    ret    0
_strlen_test ENDP

strcpy()

Listing 3.39: strcpy() example

void strcpy_test(char *s1, char *outbuf)
{
    strcpy(outbuf, s1);
};

Listing 3.40: Optimizing MSVC 2010

_s1$ = 8 ; size = 4
_outbuf$ = 12 ; size = 4
strcpy_test PROC
    mov    eax, DWORD PTR _s1$[esp-4]
    mov    edx, DWORD PTR _outbuf$[esp-4]
    sub    edx, eax
    npad   6 ; align next label
$LL3@strcpy_tes:
    mov    cl, BYTE PTR [eax]
    inc    edx, eax
    test   cl, cl
    jne SHORT $LL3@strcpy_tes
    ret    0
strcpy_test ENDP
memset()

Example#1

Listing 3.41: 32 bytes

```c
#include <stdio.h>

void f(char *out)
{
    memset(out, 0, 32);
}
```

Many compilers don’t generate a call to memset() for short blocks, but rather insert a pack of MOVs:

Listing 3.42: Optimizing GCC 4.9.1 x64

```
f:
  mov QWORD PTR [rdi], 0
  mov QWORD PTR [rdi+8], 0
  mov QWORD PTR [rdi+16], 0
  mov QWORD PTR [rdi+24], 0
  ret
```

By the way, that remind us of unrolled loops: 1.22.1 on page 191.

Example#2

Listing 3.43: 67 bytes

```c
#include <stdio.h>

void f(char *out)
{
    memset(out, 0, 67);
}
```

When the block size is not a multiple of 4 or 8, the compilers can behave differently.

For instance, MSVC 2012 continues to insert MOVs:

Listing 3.44: Optimizing MSVC 2012 x64

```
out$ = 8
f PROC
  xor eax, eax
  mov QWORD PTR [rcx], rax
  mov QWORD PTR [rcx+8], rax
  mov QWORD PTR [rcx+16], rax
  mov QWORD PTR [rcx+24], rax
  mov QWORD PTR [rcx+32], rax
  mov QWORD PTR [rcx+40], rax
  mov QWORD PTR [rcx+48], rax
  mov QWORD PTR [rcx+56], rax
  mov WORD PTR [rcx+64], ax
  mov BYTE PTR [rcx+66], al
  ret 0
f ENDP
```

...while GCC uses REP STOSQ, concluding that this would be shorter than a pack of MOVs:

Listing 3.45: Optimizing GCC 4.9.1 x64

```
f:
  mov QWORD PTR [rdi], 0
  mov QWORD PTR [rdi+59], 0
  mov rcx, rdi
  lea rdi, [rdi+8]
  xor eax, eax
  and rdi, -8
```
memcpy()

Short blocks

The routine to copy short blocks is often implemented as a sequence of MOV instructions.

Listing 3.46: memcpy() example

```c
void memcpy_7(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 7);
}
```

Listing 3.47: Optimizing MSVC 2010

```c
_inbuf$ = 8  ; size = 4
_outbuf$ = 12; size = 4
_memcpy_7 PROC
    mov    ecx, DWORD PTR _inbuf$[esp-4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR _outbuf$[esp-4]
    mov    DWORD PTR [eax+10], edx
    mov    dx, WORD PTR [ecx+4]
    mov    WORD PTR [eax+14], dx
    mov    cl, BYTE PTR [ecx+6]
    mov    BYTE PTR [eax+16], cl
    ret 0
_memcpy_7 ENDP
```

Listing 3.48: Optimizing GCC 4.8.1

```c
memcpy_7:
    push  ebx
    mov   eax, DWORD PTR [esp+8]
    mov   ecx, DWORD PTR [esp+12]
    mov   ebx, DWORD PTR [eax]
    lea   edx, [ecx+10]
    mov   DWORD PTR [ecx+10], ebx
    movzx  ecx, WORD PTR [eax+4]
    mov   WORD PTR [edx+4], cx
    movzx  eax, BYTE PTR [eax+6]
    mov   BYTE PTR [edx+6], al
    pop   ebx
    ret
```

That's usually done as follows: 4-byte blocks are copied first, then a 16-bit word (if needed), then the last byte (if needed).

Structures are also copied using MOV: 1.30.4 on page 358.

Long blocks

The compilers behave differently in this case.

Listing 3.49: memcpy() example

```c
void memcpy_128(char *inbuf, char *outbuf)
{
    memcpy(outbuf+10, inbuf, 128);
}
```

void memcpy_123(char *inbuf, char *outbuf)
For copying 128 bytes, MSVC uses a single MOVSD instruction (because 128 divides evenly by 4):

Listing 3.50: Optimizing MSVC 2010

```
memcpy(outbuf+10, inbuf, 123);
```

When copying 123 bytes, 30 32-bit words are copied first using MOVSD (that’s 120 bytes), then 2 bytes are copied using MOVSW, then one more byte using MOVSB.

Listing 3.51: Optimizing MSVC 2010

```
inbuf$ = 8 ; size = 4
outbuf$ = 12; size = 4
_memcpy_128 PROC
    push esi
    mov esi, DWORD PTR _inbuf$[esp]
    push edi
    mov edi, DWORD PTR _outbuf$[esp+4]
    add edi, 10
    mov ecx, 32
    rep movsd
    pop edi
    pop esi
    ret 0
_memcpy_128 ENDP
```

GCC uses one big universal functions, that works for any block size:

Listing 3.52: Optimizing GCC 4.8.1

```
memcpy_123:
.LFB3:
push edi
mov eax, 123
push esi
mov edx, DWORD PTR [esp+16]
mov esi, DWORD PTR [esp+12]
lea edi, [edx+10]
test edi, 1
jne .L24
test edi, 2
jne .L25
.L7:
mov ecx, eax
xor edx, edx
shr ecx, 2
test al, 2
rep movsd
je .L8
movzx edx, WORD PTR [esi]
mov WORD PTR [edi], dx
mov edx, 2
.L8:
```
Universal memory copy functions usually work as follows: calculate how many 32-bit words can be copied, then copy them using MOVSD, then copy the remaining bytes.

More advanced and complex copy functions use SIMD instructions and also take the memory alignment in consideration.

As an example of SIMD strlen() function: 1.36.2 on page 414.

**memcmp()**

Listing 3.53: memcmp() example

```c
int memcmp_1235(char *buf1, char *buf2)
{
    return memcmp(buf1, buf2, 1235);
}
```

For any block size, MSVC 2013 inserts the same universal function:

Listing 3.54: Optimizing MSVC 2010

```c
_buf1$ = 8 ; size = 4
_buf2$ = 12 ; size = 4
_memcmp_1235 PROC
    mov  ecx, DWORD PTR _buf1$[esp-4]
    mov  edx, DWORD PTR _buf2$[esp-4]
    push esi
    mov  esi, 1231
    npad 2
$LL5@memcmp_1235:
    mov  eax, DWORD PTR [ecx]
    cmp  eax, DWORD PTR [edx]
    jne  SHORT $LN4@memcmp_123
    add  edx, 4
    add  esi, 4
    jae  SHORT $LL5@memcmp_123
$LN4@memcmp_1235:
    mov  al, BYTE PTR [ecx]
    cmp  al, BYTE PTR [edx]
    jne  SHORT $LN6@memcmp_123
    mov  al, BYTE PTR [ecx+1]
    cmp  al, BYTE PTR [edx+1]
    jne  SHORT $LN6@memcmp_123

```
mov al, BYTE PTR [ecx+2]
cmp al, BYTE PTR [edx+2]
jne SHORT $LN6@memcmp_123
cmp esi, -1
je SHORT $LN3@memcmp_123
mov al, BYTE PTR [ecx+3]
cmp al, BYTE PTR [edx+3]
jne SHORT $LN6@memcmp_123

$LN3@memcmp_123:
xor eax, eax
pop esi
ret 0

$LN6@memcmp_123:
sbb eax, eax
or eax, 1
pop esi
ret 0

_memcmp_1235 ENDP

strcat()

This is inlined strcat() as it has been generated by MSVC 6.0. There are 3 parts visible: 1) getting source string length (first scasb); 2) getting destination string length (second scasb); 3) copying source string into the end of destination string (movsd/movsb pair).

Listing 3.55: strcat()

lea edi, [src]
or ecx, 0FFFFFFFFh
repne scasb
not ecx
sub edi, ecx
mov esi, edi
mov edi, [dst]
mov edx, ecx
or ecx, 0FFFFFFFFh
repne scasb
mov ecx, edx
dec edi
shr ecx, 2
rep movsd
mov ecx, edx
and ecx, 3
rep movsb

IDA script

There is also a small IDA script for searching and folding such very frequently seen pieces of inline code: GitHub.

### 3.15 C99 restrict

Here is a reason why Fortran programs, in some cases, work faster than C/C++ ones.

```c
void f1 (int* x, int* y, int* sum, int* product, int* sum_product, int* update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    }
};
```
That's very simple example with one specific thing in it: the pointer to the `update_me` array could be a pointer to the `sum` array, product array, or even the `sum_product` array—nothing forbids that, right?

The compiler is fully aware of this, so it generates code with four stages in the loop body:

- calculate next `sum[i]`
- calculate next `product[i]`
- calculate next `update_me[i]`
- calculate next `sum_product[i]`—on this stage, we need to load from memory the already calculated `sum[i]` and `product[i]`

Is it possible to optimize the last stage? Since we have already calculated `sum[i]` and `product[i]` it is not necessary to load them again from memory.

Yes, but compiler is not sure that nothing has been overwritten at the 3rd stage! This is called “pointer aliasing”, a situation when the compiler cannot be sure that a memory to which a pointer is pointing hasn’t been changed.

_re restrict_ in the C99 standard [ISO/IEC 9899:TC3 (C C99 standard), (2007) 6.7.3/1] is a promise, given by programmer to the compiler that the function arguments marked by this keyword always points to different memory locations and never intersects.

To be more precise and describe this formally, _restrict_ shows that only this pointer is to be used to access an object, and no other pointer will be used for it.

It can be even said the object will be accessed only via one single pointer, if it is marked as _restrict_.

Let's add this keyword to each pointer argument:

```c
void f2 (int* restrict x, int* restrict y, int* restrict sum, int* restrict product, int* restrict sum_product, int* restrict update_me, size_t s)
{
    for (int i=0; i<s; i++)
    {
        sum[i]=x[i]+y[i];
        product[i]=x[i]*y[i];
        update_me[i]=i*123; // some dummy value
        sum_product[i]=sum[i]+product[i];
    }
}
```

Let's see results:

<table>
<thead>
<tr>
<th>Listing 3.56: GCC x64: f1()</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1:</td>
</tr>
<tr>
<td>push r15 r14 r13 r12 rbp rdi rsi rbx</td>
</tr>
<tr>
<td>mov r13, QWORD PTR 120[rsp]</td>
</tr>
<tr>
<td>mov rbp, QWORD PTR 104[rsp]</td>
</tr>
<tr>
<td>mov r12, QWORD PTR 112[rsp]</td>
</tr>
<tr>
<td>test r13, r13</td>
</tr>
<tr>
<td>je .L1</td>
</tr>
<tr>
<td>add r13, 1</td>
</tr>
<tr>
<td>xor ebx, ebx</td>
</tr>
<tr>
<td>mov edi, 1</td>
</tr>
<tr>
<td>xor r11d, r11d</td>
</tr>
<tr>
<td>jmp .L4</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>.L6:</td>
</tr>
<tr>
<td>mov r11, rdi</td>
</tr>
<tr>
<td>mov rdi, rax</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>.L4:</td>
</tr>
<tr>
<td>lea rax, 0[0+r11*4]</td>
</tr>
<tr>
<td>lea r10, [rcx+rax]</td>
</tr>
<tr>
<td>lea r14, [rdx+rax]</td>
</tr>
<tr>
<td>lea rsi, [r8+rax]</td>
</tr>
<tr>
<td>add rax, r9</td>
</tr>
<tr>
<td>mov r15d, DWORD PTR [r10]</td>
</tr>
<tr>
<td>add r15d, DWORD PTR [r14]</td>
</tr>
<tr>
<td>mov DWORD PTR [rsi], r15d</td>
</tr>
<tr>
<td>mov r16d, DWORD PTR [r10]</td>
</tr>
</tbody>
</table>
```

515
The difference between the compiled \texttt{f1()} and \texttt{f2()} functions is as follows: in \texttt{f1()}, \texttt{sum[i]} and \texttt{product[i]} are reloaded in the middle of the loop, and in \texttt{f2()} there is no such thing, the already calculated values are used, since we “promised” the compiler that no one and nothing will change the values in \texttt{sum[i]} and \texttt{product[i]} during the execution of the loop’s body, so it is “sure” that there is no need to load the value from memory again.

Obviously, the second example works faster.

But what if the pointers in the function’s arguments intersect somehow?

This is on the programmer’s conscience, and the results will be incorrect.

Let’s go back to Fortran.

Compilers of this programming language treats all pointers as such, so when it was not possible to set \textit{restrict} in C, Fortran could generate faster code in these cases.

How practical is it?

In the cases when the function works with several big blocks in memory.

There are a lot of such in linear algebra, for instance.
Supercomputers/HPC\textsuperscript{14} are very busy with linear algebra, so probably that is why, traditionally, Fortran is still used there [Eugene Loh, \textit{The Ideal HPC Programming Language}, (2010)].

But when the number of iterations is not very big, certainly, the speed boost may not to be significant.

### 3.16 Branchless \texttt{abs()} function

Let’s revisit an example we considered earlier \texttt{1.18.2} on page 140 and ask ourselves, is it possible to make a branchless version of the function in x86 code?

```c
int my_abs (int i)
{
    if (i<0)
        return -i;
    else
        return i;
}
```

And the answer is yes.

#### 3.16.1 Optimizing GCC 4.9.1 x64

We could see it if we compile it using optimizing GCC 4.9:

```c
my_abs:
    mov    edx, edi
    mov    eax, edi
    sar    edx, 31
    ; EDX is 0xFFFFFFFF here if sign of input value is minus
    ; EDX is 0 if sign of input value is plus (including 0)
    ; the following two instructions have effect only if EDX is 0xFFFFFFFF
    ; or idle if EDX is 0
    xor    eax, edx
    sub    eax, edx
    ret
```

This is how it works:

Arithmetically shift the input value right by 31.
Arithmetical shift implies sign extension, so if the \texttt{MSB} is 1, all 32 bits are to be filled with 1, or with 0 if otherwise.

In other words, the \texttt{SAR REG, 31} instruction makes 0xFFFFFFFF if the sign has been negative or 0 if positive.

After the execution of \texttt{SAR}, we have this value in \texttt{EDX}.

Then, if the value is 0xFFFFFFFF (i.e., the sign is negative), the input value is inverted (because \texttt{XOR REG, 0xFFFFFFFF} is effectively an inverse all bits operation).

Then, again, if the value is 0xFFFFFFFF (i.e., the sign is negative), 1 is added to the final result (because subtracting \(-1\) from some value resulting in incrementing it).

Inversion of all bits and incrementing is exactly how two’s complement value is negated: \texttt{2.2} on page 448.

We may observe that the last two instruction do something if the sign of the input value is negative.
Otherwise (if the sign is positive) they do nothing at all, leaving the input value untouched.


It’s hard to say, how GCC did it, deduced it by itself or found a suitable pattern among known ones?

---

\textsuperscript{14}High-Performance Computing
3.16.2 Optimizing GCC 4.9 ARM64

GCC 4.9 for ARM64 generates mostly the same, just decides to use the full 64-bit registers.

There are less instructions, because the input value can be shifted using a suffixed instruction (“asr”) instead of using a separate instruction.

Listing 3.59: Optimizing GCC 4.9 ARM64

my_abs:
; sign-extend input 32-bit value to X0 64-bit register:
 sxtw x0, w0
 eor x1, x0, x0, asr 63
 ; X1=X0^(X0>>63) (shift is arithmetical)
 sub x0, x1, x0, asr 63
 ; X0=X1-(X0>>63)=X0^(X0>>63)-(X0>>63) (all shifts are arithmetical)
 ret

3.17 Variadic functions

Functions like printf() and scanf() can have a variable number of arguments. How are these arguments accessed?

3.17.1 Computing arithmetic mean

Let's imagine that we want to calculate arithmetic mean, and for some weird reason we want to specify all the values as function arguments.

But it's impossible to get the number of arguments in a variadic function in C/C++, so let's denote the value of −1 as a terminator.

Using va_arg macro

There is the standard stdarg.h header file which define macros for dealing with such arguments.

The printf() and scanf() functions use them as well.

```
#include <stdio.h>
#include <stdarg.h>

int arith_mean(int v, ...)
{
    va_list args;
    int sum=v, count=1, i;
    va_start(args, v);

    while(1)
    {
        i=va_arg(args, int);
        if (i==-1) // terminator
            break;
        sum=sum+i;
        count++;
    }

    va_end(args);
    return sum/count;
}

int main()
{
    printf("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* terminator */));
}
```

The first argument has to be treated just like a normal argument.

All other arguments are loaded using the va_arg macro and then summed.

So what is inside?
cdecl calling conventions

Listing 3.60: Optimizing MSVC 6.0

_v$ = 8
_arith_mean PROC NEAR
  mov  eax, DWORD PTR _v$[esp-4] ; load 1st argument into sum
  push esi ; count=1
  mov  esi, 1
  lea  edx, DWORD PTR _v$[esp] ; address of the 1st argument
$L838:
  mov  ecx, DWORD PTR [edx+4] ; load next argument
  add  edx, 4 ; shift pointer to the next argument
  cmp  ecx, -1 ; is it -1?
  je   SHORT $L856 ; exit if so
  add  eax, ecx ; sum = sum + loaded argument
  inc  esi ; count++
  jmp  SHORT $L838
$L856:
; calculate quotient
  cdq
  idiv  esi
  pop  esi
  ret 0
_arith_mean ENDP

$SG851 DB '%d', 0aH, 00H

_main PROC NEAR
  push  -1
  push  15
  push  10
  push  7
  push  2
  push  1
  call  _arith_mean
  push  eax
  push  OFFSET FLAT:$SG851 ; '%d'
  call  _printf
  add  esp, 32
  ret  0
_main ENDP

The arguments, as we may see, are passed to main() one-by-one.
The first argument is pushed into the local stack as first.
The terminating value (−1) is pushed last.
The arith_mean() function takes the value of the first argument and stores it in the sum variable.
Then, it sets the EDX register to the address of the second argument, takes the value from it, adds it to sum, and does this in an infinite loop, until −1 is found.
When it’s found, the sum is divided by the number of all values (excluding −1) and the quotient is returned.
So, in other words, the function treats the stack fragment as an array of integer values of infinite length.
Now we can understand why the cdecl calling convention forces us to push the first argument into the stack as last.
Because otherwise, it would not be possible to find the first argument, or, for printf-like functions, it would not be possible to find the address of the format-string.

Register-based calling conventions

The observant reader may ask, what about calling conventions where the first few arguments are passed in registers? Let’s see:
We see that the first 4 arguments are passed in the registers and two more—in the stack.

The arith_mean() function first places these 4 arguments into the Shadow Space and then treats the Shadow Space and stack behind it as a single continuous array!

What about GCC? Things are slightly clumsier here, because now the function is divided in two parts: the first part saves the registers into the “red zone”, processes that space, and the second part of the function processes the stack:

```
Listing 3.62: Optimizing GCC 4.9.1 x64

%SG3013 DB 'd', 0AH, 00H

v$ = 8
arith_mean PROC
    lea rax, [rsp+8] ; save 6 input registers in
    ; red zone in the local stack
    mov QWORD PTR [rsp-40], rsi
    mov QWORD PTR [rsp-32], rdx
    mov QWORD PTR [rsp-16], r8
    mov QWORD PTR [rsp-24], rcx
    mov esi, 8
    mov QWORD PTR [rsp-64], rax
    lea rax, [rsp-48]
    mov QWORD PTR [rsp-8], r9
    mov DWORD PTR [rsp-72], 8
    lea rdx, [rsp+8]

arith_mean ENDP
```
mov r8d, 1
mov QWORD PTR [rsp-56], rax
jmp .L5

.L7:
; work out saved arguments
lea rax, [rsp-48]
mov ecx, esi
add esi, 8
add rcx, rax
mov ecx, DWORD PTR [rcx]
cmp ecx, -1
je .L4

.L8:
add edi, ecx
add r8d, 1

.L5:
; decide, which part we will work out now.
; is current argument number less or equal 6?
cmp esi, 47
jbe .L7 ; no, process saved arguments then
; work out arguments from stack
mov rcx, rdx
add rdx, 8
mov ecx, DWORD PTR [rcx]
cmp ecx, -1
jne .L8

.L4:
mov eax, edi
cdq
idiv r8d
ret

.LC1:
.string "\%d\n"
main:
sub rsp, 8
mov edx, 7
mov esi, 2
mov edi, 1
mov r9d, -1
mov r8d, 15
mov ecx, 10
xor eax, eax
call arith_mean
mov esi, OFFSET FLAT:.LC1
mov edx, eax
mov edi, 1
xor eax, eax
add rsp, 8
jmp __printf_chk

By the way, a similar usage of the Shadow Space is also considered here: 6.1.8 on page 731.

Using pointer to the first function argument
The example can be rewritten without va_arg macro:

#include <stdio.h>

int arith_mean(int v, ...)
{
    int *i=&v;
    int sum=*i, count=1;
    i++;
    while(1)
    {
        if ((*i)==-1) // terminator
            break;
        }
```c
sum = sum + (*i);
count++;  
i++;  
}

return sum / count;
};
int main()
{
    printf("%d\n", arith_mean (1, 2, 7, 10, 15, -1 /* terminator */));
    // test: https://www.wolframalpha.com/input/?i=mean(1,2,7,10,15)
}
}
```

In other words, if an argument set is array of words (32-bit or 64-bit), we just enumerate array elements starting at first one.

### 3.17.2 `vprintf()` function case

Many programmers define their own logging functions which take a printf-like format string + a variable number of arguments.

Another popular example is the `die()` function, which prints some message and exits.

We need some way to pack input arguments of unknown number and pass them to the `printf()` function. But how?

That's why there are functions with "v" in name.

One of them is `vprintf()`: it takes a format-string and a pointer to a variable of type `va_list`:

```c
#include <stdlib.h>
#include <stdarg.h>

void die (const char * fmt, ...)
{
    va_list va;
    va_start (va, fmt);
    vprintf (fmt, va);
    exit(0);
};
```

By closer examination, we can see that `va_list` is a pointer to an array. Let's compile:

#### Listing 3.63: Optimizing MSVC 2010

```assembly
_fmt$ = 8
_die PROC
    ; load 1st argument (format-string)
    mov ecx, DWORD PTR _fmt$[esp-4]
    ; get pointer to the 2nd argument
    lea eax, DWORD PTR _fmt$[esp]
    push eax
    push ecx
    call _vprintf
    add esp, 8
    push 0
    call _exit
$LN3@die:
    int 3
_die ENDP
```

We see that all our function does is just taking a pointer to the arguments and passing it to `vprintf()`, and that function is treating it like an infinite array of arguments!

#### Listing 3.64: Optimizing MSVC 2012 x64

```assembly
_fmt$ = 48
_die PROC
    ; save first 4 arguments in Shadow Space
```
3.17.3 Pin case

It's interesting to note how some functions from Pin DBI framework takes number of arguments:

```
INS_InsertPredicatedCall(
    ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
    IARG_INST_PTR,
    IARG_MEMORYOP_EA, memOp,
    IARG_END);
```

(pinatrace.cpp)

And this is how INS_InsertPredicatedCall() function is declared:

```
extern VOID INS_InsertPredicatedCall(INS ins, IPOINT ipoint, AFUNPTR funptr, ...);
```

(pin_client.PH)

Hence, constants with names starting with IARG_ are some kinds of arguments to the function, which are handled inside of INS_InsertPredicatedCall(). You can pass as many arguments, as you need. Some commands has additional argument(s), some are not. Full list of arguments: [https://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/group__INST__ARGS.html](https://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/group__INST__ARGS.html). And it has to be a way to detect an end of arguments list, so the list must be terminated with IARG_END constant, without which, the function will (try to) handle random noise in the local stack, treating it as additional arguments.

Also, in [Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)] we can find a nice example of C/C++ routines very similar to **pack/unpack** in Python.

3.17.4 Format string exploit

It's a popular mistake, to write `printf(string)` instead of `puts(string)` or `printf("%s", string)`. If the attacker can put his/her own text into `string`, he/she can crash process, or get insight into variables in the local stack.

Take a look at this:

```
#include <stdio.h>

int main()
{
    char *s1="hello";
    char *s2="world";
    char buf[128];

    // do something mundane here
    strcpy (buf, s1);
    strcpy (buf, " ");
    strcpy (buf, s2);
    printf ("%s");
}
```

---

15 Dynamic Binary Instrumentation
16 [https://docs.python.org/3/library/struct.html](https://docs.python.org/3/library/struct.html)
Please note, that printf() has no additional arguments besides single format string.

Now let’s imagine, that was the attacker who put %s string into the last printf() first arguments. I compile this example using GCC 5.4.0 on x86 Ubuntu, and the resulting executable prints “world” string if it gets executed!

If I turn optimization on, printf() outputs some garbage, though—probably, strcpy() calls has been optimized and/or local variables as well. Also, result will be different for x64 code, different compiler, OS, etc.

Now, let’s say, attacker could pass the following string to printf() call: %x %x %x %x %x. In may case, output is: “80485c6 b7751b48 1 0 80485c0” (these are just values from local stack). You see, there are 1 and 0 values, and some pointers (first is probably pointer to “world” string). So if the attacker passes %s %s %s %s %s string, the process will crash, because printf() treats 1 and/or 0 as pointer to string, tries to read characters from there and fails.

Even worse, there could be sprintf (buf, string) in code, where buf is a buffer in the local stack with size of 1024 bytes or so, attacker can craft string in such a way that buf will be overflown, maybe even in a way that would lead to code execution.

Many popular and well-known software was (or even still) vulnerable:

```
QuakeWorld went up, got to around 4000 users, then the master server exploded.
Disrupter and cohorts are working on more robust code now.
If anyone did it on purpose, how about letting us know... (It wasn’t all the people that tried %s as a name)
```

( John Carmack’s .plan file, 17-Dec-1996\(^{17}\) )

Nowadays, almost all decent compilers warn about this.

Another problem is the lesser known %n printf() argument: whenever printf() reaches it in a format string, it writes the number of characters printed so far into the corresponding argument: [http://stackoverflow.com/questions/3401156/what-is-the-use-of-the-n-format-specifier-in-c](http://stackoverflow.com/questions/3401156/what-is-the-use-of-the-n-format-specifier-in-c). Thus, an attacker could zap local variables by passing many %n commands in format string.

### 3.18 Strings trimming

A very common string processing task is to remove some characters at the start and/or at the end.

In this example, we are going to work with a function which removes all newline characters (CR\(^{18}\)/LF\(^{19}\)) from the end of the input string:

```
#include <stdio.h>
#include <string.h>

char* str_trim (char *s)
{
    char c;
    size_t str_len;
    
    // work as long as \r or \n is at the end of string
    // stop if some other character there or its an empty string
    // (at start or due to our operation)
    for (str_len=strlen(s); str_len>0 && (c=s[str_len-1]); str_len--)
    {
        if (c==\'\r\' || c==\'\n\')
            s[str_len-1]=0;
        else
            break;
    }
    return s;
}
```

\(^{17}\)https://github.com/ESWAT/john-carmack-plan-archive/blob/33ae52fdba46aa0d1abfed6fc7598233748541c08/by_day/johnnc_plan_19961217.txt

\(^{18}\)Carriage Return (13 or ‘\r’ in C/C++)

\(^{19}\)Line Feed (10 or ‘\n’ in C/C++)
```c
int main()
{
    // test
    // strdup() is used to copy text string into data segment,
    // because it will crash on Linux otherwise,
    // where text strings are allocated in constant data segment,
    // and not modifiable.
    printf ("[\%s]\n", str_trim (strdup("")));
    printf ("[\%s]\n", str_trim (strdup("\n")));
    printf ("[\%s]\n", str_trim (strdup("\r")));
    printf ("[\%s]\n", str_trim (strdup("\n\r")));
    printf ("[\%s]\n", str_trim (strdup("test1\n\r")));
    printf ("[\%s]\n", str_trim (strdup("test2\n\r")));
    printf ("[\%s]\n", str_trim (strdup("test3\n\r\n")));
    printf ("[\%s]\n", str_trim (strdup("test4\n")));
    printf ("[\%s]\n", str_trim (strdup("test5\r")));
    printf ("[\%s]\n", str_trim (strdup("test6\r\r\r")));
}
```

The input argument is always returned on exit, this is convenient when you want to chain string processing functions, like it has done here in the `main()` function.

The second part of for() (str_len>0 & (c=s[str_len-1])) is the so called “short-circuit” in C/C++ and is very convenient [Dennis Yurichev, C/C++ programming language notes1.3.8].

The C/C++ compilers guarantee an evaluation sequence from left to right.

So if the first clause is false after evaluation, the second one is never to be evaluated.

### 3.18.1 x64: Optimizing MSVC 2013

Listing 3.65: Optimizing MSVC 2013 x64

```assembly
s$ = 8
str_trim PROC
    ; RCX is the first function argument and it always holds pointer to the string
    mov    rdx, rcx
    ; this is strlen() function inlined right here:
    ; set RAX to 0xFFFFFFFFFFFFFFFF (-1)
    or    rax, -1
$LL14@str_trim:
    inc    rax
    cmp    BYTE PTR [rcx+rax], 0
    jne    SHORT $LL14@str_trim
    ; is the input string length zero? exit then:
    test   rax, rax
    je     SHORT $LN15@str_trim
    ; RAX holds string length
    dec    rcx
    ; RCX = s-1
    mov    r8d, 1
    add    rcx, rax
    ; RCX = s-1+strlen(s), i.e., this is the address of the last character in the string
    sub    r8, rdx
    ; R8 = 1-s
$LL6@str_trim:
    ; load the last character of the string:
    ; jump, if its code is 13 or 10:
    movzx   eax, BYTE PTR [rcx]
    cmp    al, 13
    je     SHORT $LN2@str_trim
    cmp    al, 10
    jne    SHORT $LN15@str_trim
$LN2@str_trim:
    ; the last character has a 13 or 10 code
    ; write zero at this place:
```
First, MSVC inlined the `strlen()` function code, because it concluded this is to be faster than the usual `strlen()` work + the cost of calling it and returning from it. This is called inlining: 3.14 on page 506.

The first instruction of the inlined `strlen()` is

OR RAX, 0xFFFFFFFFFFFFFFFF.

MSVC often uses OR instead of MOV RAX, 0xFFFFFFFFFFFFFFFF, because resulting opcode is shorter.

And of course, it is equivalent: all bits are set, and a number with all bits set is \(-1\) in two’s complement arithmetic: 2.2 on page 448.

Why would the \(-1\) number be used in `strlen()`, one might ask. Due to optimizations, of course. Here is the code that MSVC generated:

```
Listing 3.66: Inlined ```strlen()``` by MSVC 2013 x64

; RCX = pointer to the input string
; RAX = current string length
or rax, -1
label:
  inc rax
  cmp BYTE PTR [rcx+rax], 0
  jne SHORT label
; RAX = string length
```

Try to write shorter if you want to initialize the counter at 0! OK, let’ try:

```
Listing 3.67: Our version of ```strlen()```

; RCX = pointer to the input string
; RAX = current string length
xor rax, rax
label:
  cmp byte ptr [rcx+rax], 0
  jz exit
  inc rax
  jmp label
exit:
; RAX = string length
```

We failed. We have to use additional JMP instruction!

So what the MSVC 2013 compiler did is to move the INC instruction to the place before the actual character loading.

If the first character is 0, that’s OK, RAX is 0 at this moment, so the resulting string length is 0.

The rest in this function seems easy to understand.

### 3.18.2 x64: Non-optimizing GCC 4.9.1

```
str_trim:
push rbp
mov rbp, rsp
sub rsp, 32
mov QWORD PTR [rbp-24], rdi
; for() first part begins here
mov rax, QWORD PTR [rbp-24]
```
mov rdi, rax
 call strlen
 mov QWORD PTR [rbp-8], rax ; str_len
 ; for() first part ends here
 jmp .L2
 ; for() body begins here

.L5:
   cmp BYTE PTR [rbp-9], 13 ; c=='\r'?  
   je .L3
   cmp BYTE PTR [rbp-9], 10 ; c=='\n'?  
   jne .L4

.L3:
   mov rax, QWORD PTR [rbp-8] ; str_len
   lea rdx, [rax-1] ; EDX=str_len-1
   mov rax, QWORD PTR [rbp-24] ; s
   add rax, rdx ; RAX=s+str_len-1
   mov BYTE PTR [rax], 0 ; s[str_len-1]=0
 ; for() body ends here
 ; for() third part begins here
   sub QWORD PTR [rbp-8], 1 ; str_len--
 ; for() third part ends here

.L2:
 ; for() second part begins here
   cmp QWORD PTR [rbp-8], 0 ; str_len==0?
   je .L4 ; exit then
 ; check second clause, and load "c"
   mov rax, QWORD PTR [rbp-8] ; RAX=str_len
   lea rdx, [rax-1] ; RDX=str_len-1
   mov rax, QWORD PTR [rbp-24] ; RAX=s
   add rax, rdx ; RAX=s+str_len-1
   movzx eax, BYTE PTR [rax] ; AL=s[str_len-1]
   mov BYTE PTR [rbp-9], al ; store loaded char into "c"
   cmp BYTE PTR [rbp-9], 0 ; is it zero?
   jne .L5 ; yes? exit then
 ; for() second part ends here

.L4: ; return "s"
   leave
   ret

Comments are added by the author of the book.

After the execution of strlen(), the control is passed to the L2 label, and there two clauses are checked, one after another.

The second will never be checked, if the first one (str_len==0) is false (this is “short-circuit”).

Now let’s see this function in short form:

- First for() part (call to strlen())
- goto L2
- L5: for() body. goto exit, if needed
- for() third part (decrement of str_len)
- L2: for() second part: check first clause, then second. goto loop body begin or exit.
- L4: // exit
- return s

3.18.3 x64: Optimizing GCC 4.9.1

str_trim:
   push rbx
   mov rbx, rdi
   ; RBX will always be "s"
   call strlen
   ; check for str_len==0 and exit if its so
Now this is more complex.

The code before the loop’s body start is executed only once, but it has the CR/LF characters check too! What is this code duplication for?

The common way to implement the main loop is probably this:

• (loop start) check for CR/LF characters, make decisions
• store zero character

But GCC has decided to reverse these two steps.

Of course, store zero character cannot be first step, so another check is needed:

• workout first character. match it to CR/LF, exit if character is not CR/LF
• (loop begin) store zero character
• check for CR/LF characters, make decisions

Now the main loop is very short, which is good for latest CPUs.

The code doesn’t use the str_len variable, but str_len-1. So this is more like an index in a buffer.

 Apparently, GCC notices that the str_len-1 statement is used twice.

So it’s better to allocate a variable which always holds a value that’s smaller than the current string length by one, and decrement it (this is the same effect as decrementing the str_len variable).
This implementation is straightforward:

```assembly
str_trim:
    stp  x29, x30, [sp, -48]!
    add  x29, sp, 0
    str  x0, [x29,24] ; copy input argument into local stack
    ldr  x0, [x29,24] ; s
    bl   strlen
    str  x0, [x29,40] ; str_len variable in local stack

; main loop begin
.L5:
    ldrb w0, [x29,39]
    ; W0=c
    cmpw w0, 13       ; is it '\r'?
    beq   .L3
    ldrb w0, [x29,39]
    ; W0=c
    cmpw w0, 10       ; is it '\n'?
    bne   .L4
    ; goto exit if it is not
.L3:
    ldr  x0, [x29,40]
    ; X0=str_len
    sub  x0, x0, #1
    ; X0=str_len-1
    ldr  x1, [x29,24]
    ; X1=s
    add  x0, x1, x0
    ; X0=s+str_len-1
    strb wzr, [x0]    ; write byte at s+str_len-1
    ; decrement str_len:
    ldr  x0, [x29,40]
    ; X0=str_len
    sub  x0, x0, #1
    ; X0=str_len-1
    str  x0, [x29,40]
    ; save X0 (or str_len-1) to local stack
.L2:
    ldr  x0, [x29,40]
    ; str_len==0?
    cmp  x0, xzr
    ; goto exit then
    beq   .L4
    ldr  x0, [x29,40]
    ; X0=str_len
    sub  x0, x0, #1
    ; X0=str_len-1
    ldr  x1, [x29,24]
    ; X1=s
    add  x0, x1, x0
    ; X0=s+str_len-1
; load byte at address s+str_len-1 to W0
    ldrb w0, [x0]
    strb w0, [x29,39] ; store loaded byte to "c"
    ldrb w0, [x29,39] ; reload it
; is it zero byte?
    cmp  w0, wzr
    ; goto exit, if its zero or to L5 if its not
    bne   .L5
.L4:
; return s
    ldr  x0, [x29,24]
    ldp  x29, x30, [sp], 48
    ret
```

Listing 3.68: Non-optimizing GCC (Linaro) 4.9
3.18.5 ARM64: Optimizing GCC (Linaro) 4.9

This is a more advanced optimization.

The first character is loaded at the beginning, and compared against 10 (the LF character). Characters are also loaded in the main loop, for the characters after first one. This is somewhat similar to the 3.18.3 on page 527 example.

Listing 3.69: Optimizing GCC (Linaro) 4.9

```
str_trim:
    strp x29, x30, [sp, -32]!
    add x29, sp, 0
    str x19, [sp,16]
    mov x19, x0
    ; X19 will always hold value of "s"
    bl strlen
    ; X0=str_len
    cbz x0, .L9 ; goto L9 (exit) if str_len==0
    sub x1, x0, #1
    ; X1=X0-1=str_len-1
    add x3, x19, x1
    ; X3=X19+X1=s+str_len-1
    ldrb w2, [x19,x1] ; load byte at address X19+X1=s+str_len-1
    ; W2=loaded character
    cbz w2, .L9 ; is it zero? jump to exit then
    cmp w2, 10 ; is it '\n'?
    bne .L12 ; main loop body. loaded character is always 10 or 13 at this moment!
    sub x2, x1, x0
    ; X2=X1-X0=str_len-1-str_len=-1
    add x2, x3, x2
    ; X2=X3+X2=s+str_len-1+(-1)=s+str_len-2
    strb wzr, [x2,1] ; store zero byte at address s+str_len-2+1=s+str_len-1
    cbz x1, .L9 ; str_len==0? goto exit, if so
    sub x1, x1, #1 ; str_len--
    ldrb w2, [x19,x1] ; load next character at address X19+X1=s+str_len-1
    cmp w2, 10 ; is it '\n'?
    bne .L12
    cmp w2, 13 ; is it '\r'?
    beq .L9 ; jump to begin loop, if its '\n'
.L12:
    cmp w2, 10 ; is it 's'?
    bne .L12 ; yes, jump to the loop body begin
.L9:
    ; return "s"
    mov x0, x19
    ldr x19, [sp,16]
    ldp x29, x30, [sp], 32
    ret
```

3.18.6 ARM: Optimizing Keil 6/2013 (ARM mode)

And again, the compiler took advantage of ARM mode’s conditional instructions, so the code is much more compact.

Listing 3.70: Optimizing Keil 6/2013 (ARM mode)

```
str_trim PROC
    PUSH {r4,lr}
    ; R0=s
    MOV r4,r0
    ; R4= buflen
    BL strlen ; strlen() takes "s" value from R0
    ; R0=str_len
    MOV r3,#0
    ; R3 will always hold 0
    CMP r0,#0 ; str_len==0?
    CMP r0,#0
    ; return "s"
    MOV x0, x19
    LDR x19, [sp,16]
    LDP x29, x30, [sp], 32
    RET
```

530
3.18.7 ARM: Optimizing Keil 6/2013 (Thumb mode)

There are less conditional instructions in Thumb mode, so the code is simpler.

But there are is really weird thing with the 0x20 and 0x1F offsets (lines 22 and 23). Why did the Keil compiler do so? Honestly, it’s hard to say.

It has to be a quirk of Keil’s optimization process. Nevertheless, the code works correctly.

```
str_trim PROC
PUSH {r4,lr}
MOVS r4,r0
; R4=s
BL strlen ; strlen() takes "s" value from R0
; R0=str_len
MOV r3,#0
; R3 will always hold 0
B |L0.24|
|L0.12|
CMP r1,#0xd ; is loaded byte '\r'? BEQ |L0.38| ; jump to exit, if no
CMP r1,#0xa ; is loaded byte '\n'? BNE |L0.20| ; jump to exit if no
SUBS r0,r0,#1 ; R0-- or str_len-- STRB r3,[r2,#0x1f] ; store 0 at address R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1 |L0.24|
CMP r0,#0 ; str_len==0?
BEQ |L0.38| ; yes? jump to exit
ADDs r2,r4,r0 ; R2=R4+R0=s+str_len SUBS r2,r2,#0x20 ; R2=R2-0x20=s+str_len-0x20 LDRB r1,[r2,#0x1f] ; load byte at address R2+0x1F=s+str_len-0x20+0x1F=s+str_len-1 to R1
CMP r1,#0 ; is loaded byte 0?
BNE |L0.12| ; jump to loop begin, if its not 0 |L0.38|
; return "s"
MOV r0,r4
POP {r4,pc}
ENDP
```

3.18.8 MIPS

```
str_trim: ; IDA is not aware of local variable names, we gave them manually:
saved_GP  = -0x10
saved_S0  = -8
saved_RA  = -4
```

Listing 3.72: Optimizing GCC 4.4.5 (IDA)
; call strl() , input string address is still in $a0 , strl() will take it from there:
lw    $t9, (strl & 0xFFFF)($gp)
or    $at, $zero ; load delay slot , NOP
jalr  $t9
; input string address is still in $a0 , put it to $s0:
move  $s0, $a0  ; branch delay slot
; result of strl() (i.e., length of string) is in $v0 now
; jump to exit if $v0==0 (i.e., if length of string is 0):
beqz  $v0, exit
or    $at, $zero ; branch delay slot , NOP
addiu $a1, $v0, -1
; $a1 = $v0-1 = strl_len
addu $a1, $s0, $a1
; $a1 = input string address + $a1 = s+strl_len-1
; load byte at address $a1:
lb     $a0, 0($a1)
or    $at, $zero ; load delay slot, NOP
; loaded byte is zero? jump to exit if its so:
beqz  $a0, exit
or    $at, $zero ; branch delay slot, NOP
addiu $v1, $v0, -2
; $v1 = strl_len-2
addu $v1, $s0, $v1
; $v1 = $s0+$v1 = s+strl_len-2
li     $a2, 0xD
; skip loop body:
lw    $ra, 0x20+saved_RA($sp)
move  $v0, $s0
lw     $s0, 0x20+saved_S0($sp)
jl     $ra
addiu $sp, 0x20  ; branch delay slot

Registers prefixed with S- are also called “saved temporaries”, so $S0 value is saved in the local stack and restored upon finish.
3.19 toupper() function

Another very popular function transforms a symbol from lower case to upper case, if needed:

```c
char toupper (char c)
{
    if(c>='a' && c<='z')
        return c-'a'+'A';
    else
        return c;
}
```

The ‘a’+'A' expression is left in the source code for better readability, it will be optimized by compiler, of course.

The ASCII code of “a” is 97 (or 0x61), and 65 (or 0x41) for “A”.

The difference (or distance) between them in the ASCII table is 32 (or 0x20).

For better understanding, the reader may take a look at the 7-bit standard ASCII table:

![7-bit ASCII table in Emacs](image)

Figure 3.3: 7-bit ASCII table in Emacs

### 3.19.1 x64

Two comparison operations

Non-optimizing MSVC is straightforward: the code checks if the input symbol is in [97..122] range (or in ['a'..'z'] range) and subtracts 32 if it’s true.

There are also some minor compiler artifact:

```
Listing 3.73: Non-optimizing MSVC 2013 (x64)
```

```
1 c$ = 8
2 toupper PROC
3     mov BYTE PTR [rsp+8], cl
4     movsx eax, BYTE PTR c$[rsp]
5     cmp eax, 97
6     jl SHORT $LN2@toupper
7     movsx eax, BYTE PTR c$[rsp]
8     cmp eax, 122
9     jg SHORT $LN2@toupper
10    movsx eax, BYTE PTR c$[rsp]
11    sub eax, 32
12    jmp SHORT $LN3@toupper
13    jmp SHORT $LN1@toupper ; compiler artefact
14 $LN2@toupper:
15     movzx eax, BYTE PTR c$[rsp] ; unnecessary casting
16 $LN1@toupper:
17 $LN3@toupper:
18    ret 0
19 toupper ENDP
```

It’s important to notice that the input byte is loaded into a 64-bit local stack slot at line 3.

However, to be meticulous, there still could be compilers which can’t optimize such expressions and will leave them right in the code.
All the remaining bits ([8..63]) are untouched, i.e., contain some random noise (you’ll see it in debugger). All instructions operate only on byte-level, so it’s fine.

The last MOVZX instruction at line 15 takes the byte from the local stack slot and zero-extends it to a int 32-bit data type.

Non-optimizing GCC does mostly the same:

Listing 3.74: Non-optimizing GCC 4.9 (x64)

```assembly
toupper:
push  rbp
mov   rbp, rsp
mov   eax, edi
mov   BYTE PTR [rbp-4], al
cmp   BYTE PTR [rbp-4], 96
jle   .L2
cmp   BYTE PTR [rbp-4], 122
jg    .L2
movzx eax, BYTE PTR [rbp-4]
sub   eax, 32
jmp   .L3
.L2:
    movzx eax, BYTE PTR [rbp-4]
.L3:
    pop   rbp
    ret
```

One comparison operation

Optimizing MSVC does a better job, it generates only one comparison operation:

Listing 3.75: Optimizing MSVC 2013 (x64)

```assembly
toupper PROC
    lea   eax, DWORD PTR [rcx-97]
    cmp   al, 25
    ja    SHORT $LN2@toupper
    movsx eax, cl
    sub   eax, 32
    ret
$LN2@toupper:
    movzx eax, cl
    ret  0
toupper ENDP
```

It was explained earlier how to replace the two comparison operations with a single one: 3.13.2 on page 505.

We will now rewrite this in C/C++:

```c
int tmp = c - 97;
if (tmp > 25)
    return c;
else
    return c - 32;
```

The `tmp` variable must be signed.

This makes two subtraction operations in case of a transformation plus one comparison.

In contrast the original algorithm uses two comparison operations plus one subtracting.

Optimizing GCC is even better, it gets rid of the jumps (which is good: 2.10.1 on page 462) by using the CMOVcc instruction:

Listing 3.76: Optimizing GCC 4.9 (x64)

```assembly
toupper:
    lea   edx, [rdi-97] ; 0x61
    lea   eax, [rdi-32] ; 0x20
```

534
At line 3 the code prepares the subtracted value in advance, as if the conversion will always happen.

At line 5 the subtracted value in EAX is replaced by the untouched input value if a conversion is not needed. And then this value (of course incorrect) is dropped.

Advance subtracting is a price the compiler pays for the absence of conditional jumps.

### 3.19.2 ARM

Optimizing Keil for ARM mode also generates only one comparison:

Listing 3.77: Optimizing Keil 6/2013 (ARM mode)

```assembly
toupper PROC
    SUB r1, r0, #0x61
    CMP r1, #0x19
    SUBLS r0, r0, #0x20
    ANDLS r0, r0, #0xff
    BX lr
ENDP
```

The SUBLS and ANDLS instructions are executed only if the value in R1 is less than 0x19 (or equal). They also do the actual conversion.

Optimizing Keil for Thumb mode generates only one comparison operation as well:

Listing 3.78: Optimizing Keil 6/2013 (Thumb mode)

```assembly
toupper PROC
    MOVS r1, r0
    SUBS r1, r1, #0x61
    CMP r1, #0x19
    BHI |L0.14|
    SUBS r0, r0, #0x20
    LSLS r0, r0, #24
    LSRS r0, r0, #24
    |L0.14|
    BX lr
ENDP
```

The last two LSLS and LSRS instructions work like `AND r0, 0xFF`: they are equivalent to the C/C++ expression `(i << 24) >> 24`.

Seems like that Keil for Thumb mode deduced that two 2-byte instructions are shorter than the code that loads the 0xFF constant into a register plus an AND instruction.

### GCC for ARM64

Listing 3.79: Non-optimizing GCC 4.9 (ARM64)

```assembly
toupper:
    sub sp, sp, #16
    strb w0, [sp,15]
    ldrb w0, [sp,15]
    cmp w0, 96
    bhs .L2
    ldrb w0, [sp,15]
    cmp w0, 122
    bhi .L2
    ldrb w0, [sp,15]
    sub w0, w0, #32
    uxtb w0, w0
    b .L3
.L2:
    ldrb w0, [sp,15]
.L3:
```

535
add    sp, sp, 16
ret

Listing 3.80: Optimizing GCC 4.9 (ARM64)

toupper:
     uxtb     w0, w0
     sub     w1, w0, #97
     uxtb     w1, w1
     cmp     w1, 25
     bhi    .L2
     sub     w0, w0, #32
     uxtb     w0, w0
 .L2:
     ret

3.19.3 Using bit operations

Given the fact that 5th bit (counting from 0th) is always present after the check, subtracting is merely clearing this sole bit, but the very same effect can be achieved with ANDing (2.5 on page 454).

Even simpler, with XOR-ing:

```c
char toupper (char c)
{
    if((c>='a' && c<='z'))
        return c^0x20;
    else
        return c;
}
```

The code is close to what the optimized GCC has produced for the previous example (3.76 on page 534):

Listing 3.81: Optimizing GCC 5.4 (x86)

toupper:
     mov     edx, DWORD PTR [esp+4]
     lea     ecx, [edx-97]
     mov     eax, edx
     xor     eax, 32
     cmp     cl, 25
     cmova   eax, edx
     ret

...but XOR is used instead of SUB.

Flipping 5th bit is just moving a cursor in ASCII table up and down by two rows.

Some people say that lowercase/uppercase letters has been placed in the ASCII table in such a way deliberately, because:

> Very old keyboards used to do Shift just by toggling the 32 or 16 bit, depending on the key; this is why the relationship between small and capital letters in ASCII is so regular, and the relationship between numbers and symbols, and some pairs of symbols, is sort of regular if you squint at it.


Therefore, we can write this piece of code, which just flips the case of letters:

```c
#include <stdio.h>

char flip (char c)
{
    if((c>='a' && c<='z') || (c>='A' && c<='Z'))
        return c^0x20;
    else
        return c;
}
```
int main()
{
    // will produce "hELLO, WORLD!"
    for (char *s="Hello, world!"; *s; s++)
        printf ("%c", flip(*s));
};

3.19.4 Summary
All these compiler optimizations are very popular nowadays and a practicing reverse engineer usually
sees such code patterns often.

3.20 Obfuscation
The obfuscation is an attempt to hide the code (or its meaning) from reverse engineers.

3.20.1 Textstrings
As we know from (5.4 on page 697), text strings may be really helpful.
Programmers who are aware of this try to hide them, making it impossible to find the string in IDA or any
hex editor.
Here is the simplest method.
This is how the string can be constructed:

```
mov byte ptr [ebx], 'h'
mov byte ptr [ebx+1], 'e'
mov byte ptr [ebx+2], 'l'
mov byte ptr [ebx+3], 'l'
mov byte ptr [ebx+4], 'o'
mov byte ptr [ebx+5], 'w'
mov byte ptr [ebx+6], 'o'
mov byte ptr [ebx+7], 'r'
mov byte ptr [ebx+8], 'l'
mov byte ptr [ebx+9], 'd'
```

The string can also be compared with another one like this:

```
mov ebx, offset username
cmp byte ptr [ebx], 'j'
jnz fail
cmp byte ptr [ebx+1], 'o'
jnz fail
cmp byte ptr [ebx+2], 'h'
jnz fail
cmp byte ptr [ebx+3], 'n'
jnz fail
jz it_is_john
```

In both cases, it is impossible to find these strings straightforwardly in a hex editor.
By the way, this is a way to work with the strings when it is impossible to allocate space for them in the
data segment, for example in a PIC21 or in shellcode.
Another method is to use sprintf() for the construction:

```
sprintf(buf, "%s%c%s%c%s", "hel","l","o w","o","rld");
```

The code looks weird, but as a simple anti-reversing measure, it may be helpful.
Text strings may also be present in encrypted form, then every string usage is to be preceded by a string
decrypting routine. For example: 8.8.2 on page 826.

21Position Independent Code
3.20.2 Executable code

Inserting garbage

Executable code obfuscation implies inserting random garbage code between real one, which executes but does nothing useful.

A simple example:

Listing 3.82: original code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>add eax, ebx</td>
<td></td>
</tr>
<tr>
<td>mul ecx</td>
<td></td>
</tr>
</tbody>
</table>

Listing 3.83: obfuscated code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>xor esi, 011223344h ; garbage</td>
<td></td>
</tr>
<tr>
<td>add esi, eax ; garbage</td>
<td></td>
</tr>
<tr>
<td>add eax, ebx ; garbage</td>
<td></td>
</tr>
<tr>
<td>mov edx, eax ; garbage</td>
<td></td>
</tr>
<tr>
<td>shl edx, 4 ; garbage</td>
<td></td>
</tr>
<tr>
<td>mul ecx       ; garbage</td>
<td></td>
</tr>
<tr>
<td>xor esi, ecx ; garbage</td>
<td></td>
</tr>
</tbody>
</table>

Here the garbage code uses registers which are not used in the real code (ESI and EDX). However, the intermediate results produced by the real code may be used by the garbage instructions for some extra mess—why not?

Replacing instructions with bloated equivalents

- MOV op1, op2 can be replaced by the PUSH op2 / POP op1 pair.
- JMP label can be replaced by the PUSH label / RET pair. IDA will not show the references to the label.
- CALL label can be replaced by the following instructions triplet: PUSH label_after_CALL_instruction / PUSH label / RET.
- PUSH op can also be replaced with the following instructions pair: SUB ESP, 4 (or 8) / MOV [ESP], op.

Always executed/never executed code

If the developer is sure that ESI at always 0 at that point:

```
mov esi, 1
... ; some code not touching ESI
dec esi
... ; some code not touching ESI
cmp esi, 0
jz real_code
; fake luggage
real_code:
```

The reverse engineer needs some time to get into it.

This is also called an opaque predicate.

Another example (and again, the developer is sure that ESI is always zero):

```
; ESI=0
add eax, ebx ; real code
mul ecx       ; real code
add eax, esi  ; opaque predicate. XOR, AND or SHL, etc, can be here instead of ADD.
```

Making a lot of mess

```
instruction 1
instruction 2
instruction 3
```
Using indirect pointers

dummy_data1 db 100h dup (0)
message1 db 'hello world',0
dummy_data2 db 200h dup (0)
message2 db 'another message',0

func proc
...mov eax, offset dummy_data1 ; PE or ELF reloc here
add eax, 100h
push eax
call dump_string
...
mov eax, offset dummy_data2 ; PE or ELF reloc here
add eax, 200h
push eax
call dump_string
...
func endp

IDA will show references only to dummy_data1 and dummy_data2, but not to the text strings.

Global variables and even functions may be accessed like that.

Now something slightly more advanced.

Honestly, I don’t know its exact name, but I would call it 
*shifted pointer*. This technique is quite common,
at least in copy protection schemes.

In short: while writing a value into global memory you use an address, but by reading you use a sum of
(other) addresses, or maybe a difference. The goal is to hide a real address from a reverse engineer who
debugs the code or exploring it in IDA (or another disassembler).

This can be a nuisance.

#include <stdio.h>

// 64KiB, but it's OK
unsigned char secret_array[0x10000];

void check_lic_key()
{
    // pretend licence check has been failed
    secret_array[0x6123]=1;  // 1 mean failed
    printf("check failed\n");  // exit(0);  // / a cracker may patch here
    // or put there another value if check is succeeded
    secret_array[0x6123]=0;
};

unsigned char get_byte_at_0x6000(unsigned char *a)
{
    return *(a+0x6000);
};
void check_again()
{
    if (get_byte_at_0x6000(secret_array+0x123)==1)
    {
        // do something mean (add watermark maybe) or report error:
        printf("check failed\n");
    }
    else
    {
        // proceed further
    }
};

int main()
{
    // at start:
    check_lic_key();
    // do something
    // ... and while in some very critical part:
    check_again();
};

If compiled by non-optimizing MSVC 2015:

_check_lic_key proc near
push ebp
mov ebp, esp
imul ecx, eax, 6123h
mov _secret_array[ecx], 1
pop ebp
retn
_check_lic_key endp

_get_byte_at_0x6000 proc near
a = dword ptr 8
push ebp
mov ebp, esp
mov eax, [ebp+a]
mov al, [eax+6000h]
pop ebp
retn
_get_byte_at_0x6000 endp

_check_again proc near
push ebp
mov ebp, esp
push offset point_passed_to_get_byte_at_0x6000
call j__get_byte_at_0x6000
add esp, 4
movzx eax, al
cmp eax, 1
jnz short loc_406735
push offset _Format ; "check failed\n"
call j__printf
add esp, 4
loc_406735:
pop ebp
retn
_check_again endp

.data:0045F5C0 ; char secret_array[65536]
.data:0045F5C0 _secret_array db 123h dup(?)

540
You see, IDA can only get two addresses: secret_array[] (start of the array) and point_passed_to_get_byte_at_0x6000.

How to deal with it: you can use hardware breakpoints on memory access operations tracer has BPMx options) or symbolic execution engine or maybe you can write a plugin for IDA...

Surely, one array can be used for many values, not limited to boolean ones...

N.B.: Optimizing MSVC 2015 is smart enough to optimize the get_byte_at_0x6000() function out.

### 3.20.3 Virtual machine / pseudo-code

A programmer can construct his/her own PL or ISA and interpreter for it.

(Unlike the pre-5.0 Visual Basic, .NET or Java machines). The reverse engineer will have to spend some time to understand the meaning and details of all of the ISA’s instructions.

He/she will also have to write a disassembler/decompiler of some sort.

### 3.20.4 Other things to mention

My own (yet weak) attempt to patch the Tiny C compiler to produce obfuscated code: http://go.yurichev.com/17220.

Using the MOV instruction for really complicated things: [Stephen Dolan, mov is Turing-complete, (2013)]

### 3.20.5 Exercise

- [http://challenges.re/29](http://challenges.re/29)

### 3.21 C++

#### 3.21.1 Classes

A simple example

Internally, the representation of C++ classes is almost the same as the structures.

Let’s try an example with two variables, two constructors and one method:

```c
#include <stdio.h>

class c
{
private:
    int v1;
    int v2;
public:
    c() // default ctor
    {
        v1=667;
        v2=999;
    };
    c(int a, int b) // ctor
    {
        v1=a;
        v2=b;
    };
    void dump()
    {
        printf("%d, %d\n", v1, v2);
    }

22Also available as [http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf](http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf)
```
int main()
{
    class c c1;
    class c c2(5,6);
    c1.dump();
    c2.dump();
    return 0;
};

MSVC: x86

Here is how the main() function looks like, translated into assembly language:

Listing 3.84: MSVC

_c2$ = -16 ; size = 8
_c1$ = -8 ; size = 8
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 16
    lea ecx, DWORD PTR _c1$[ebp]
    call ??0c@@QAE@XZ ; c::c
    push 6
    push 5
    lea ecx, DWORD PTR _c2$[ebp]
    call ??0c@@QAE@HH@Z ; c::c
    lea ecx, DWORD PTR _c1$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    lea ecx, DWORD PTR _c2$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret 0
_main ENDP

Here’s what’s going on. For each object (instance of class c) 8 bytes are allocated, exactly the size needed to store the 2 variables.

For c1 a default argumentless constructor ??0c@@QAE@XZ is called. For c2 another constructor ??0c@@QAE@HH@Z is called and two numbers are passed as arguments.

A pointer to the object (this in C++ terminology) is passed in the ECX register. This is called thiscall (3.21.1)—the method for passing a pointer to the object.

MSVC does it using the ECX register. Needless to say, it is not a standardized method, other compilers can do it differently, e.g., via the first function argument (like GCC).

Why do these functions have such odd names? That’s name mangling.

A C++ class may contain several methods sharing the same name but having different arguments—that is polymorphism. And of course, different classes may have their own methods with the same name.

Name mangling enable us to encode the class name + method name + all method argument types in one ASCII string, which is then used as an internal function name. That’s all because neither the linker, nor the DLL OS loader (mangled names may be among the DLL exports as well) knows anything about C++ or OOP23.

The dump() function is called two times.

Now let’s see the constructors’ code:

23Object-Oriented Programming
The constructors are just functions, they use a pointer to the structure in ECX, copying the pointer into their own local variable, however, it is not necessary.

From the C++ standard (C++11 12.1) we know that constructors are not required to return any values. In fact, internally, the constructors return a pointer to the newly created object, i.e., \texttt{this}.

Now the \texttt{dump()} method:
Simple enough: dump() takes a pointer to the structure that contains the two int’s from ECX, takes both values from it and passes them to printf().

The code is much shorter if compiled with optimizations (/Ox):

```
??0c@@QAE@XZ ; c::c, COMDAT
 ; _this$ = ecx
 mov eax, ecx
 mov DWORD PTR [eax], 667
 mov DWORD PTR [eax+4], 999
 ret 0
??0c@@QAE@XZ ENDP ; c::c

_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
??0c@H@H@Z PROC ; c::c, COMDAT
 ; _this$ = ecx
 mov edx, DWORD PTR _b$[esp-4]
 mov eax, ecx
 mov ecx, DWORD PTR _a$[esp-4]
 mov DWORD PTR [eax], ecx
 mov DWORD PTR [eax+4], edx
 ret 8
??0c@H@H@Z ENDP ; c::c

?dump@QEA@XZ PROC ; c::dump, COMDAT
 ; _this$ = ecx
 mov eax, DWORD PTR [ecx+4]
 mov ecx, DWORD PTR [ecx]
 push eax
 push ecx
 push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
 call _printf
 add esp, 12
 ret 0
?dump@QEA@XZ ENDP ; c::dump
```

That’s all. The other thing we must note is that the stack pointer hasn’t been corrected with add esp, X after the constructor has been called. At the same time, the constructor has ret 8 instead of RET at the end.

This is all because the thiscall (3.21.1 on page 542) calling convention is used here, which together with the stdcall (6.1.2 on page 725) method offers the callee to correct the stack instead of the caller. The ret x instruction adds X to the value in ESP, then passes the control to the caller function.

See also the section about calling conventions (6.1 on page 725).

It also has to be noted that the compiler decides when to call the constructor and destructor—but we already know that from the C++ language basics.

**MSVC: x86-64**

As we already know, the first 4 function arguments in x86-64 are passed in RCX, RDX, R8 and R9 registers, all the rest—via the stack.

Nevertheless, the this pointer to the object is passed in RCX, the first argument of the method in RDX, etc. We can see this in the c(int a, int b) method internals:

```
; void dump()

??0c@QEA@XZ PROC ; c::dump
 mov r8d, DWORD PTR [rcx+4]
 mov edx, DWORD PTR [rcx]
 lea rcx, OFFSET FLAT:?_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
 jmp _printf
??0c@QEA@XZ ENDP ; c::dump
```
The `int` data type is still 32-bit in x64\(^{24}\), so that is why 32-bit register parts are used here.

We also see `JMP printf` instead of `RET` in the `dump()` method, that `hack` we already saw earlier: \[1.21.1\] on page 154.

**GCC: x86**

It is almost the same story in GCC 4.4.1, with a few exceptions.

![Listing 3.89: GCC 4.4.1](image)

Here we see another name mangling style, specific to GNU\(^{25}\). It can also be noted that the pointer to the object is passed as the first function argument—invisible to programmer, of course.

First constructor:

\(^{24}\)Apparently, for easier porting of 32-bit C/C++ code to x64.

\(^{25}\)There is a good document about the various name mangling conventions in different compilers: [Agner Fog, Calling conventions (2015)].
It just writes two numbers using the pointer passed in the first (and only) argument.

Second constructor:

```
void _ZN1cC1Eii (int *obj, int a, int b) {
    *obj = a;
    *(obj+1) = b;
};
```

...and that is completely predictable.

Now the dump() function:
This function in its internal representation has only one argument, used as pointer to the object (this).

This function could be rewritten in C like this:

```c
void _ZN1c4dumpEv (int *obj) {
    printf("%d; %d\n", *obj, *(obj+1));
};
```

Thus, if we base our judgment on these simple examples, the difference between MSVC and GCC is the style of the encoding of function names (name mangling) and the method for passing a pointer to the object (via the ECX register or via the first argument).

**GCC: x86-64**

The first 6 arguments, as we already know, are passed in the RDI, RSI, RDX, RCX, R8 and R9 ([Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, System V Application Binary Interface. AMD64 Architecture Processor Supplement, (2013)](https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf) registers, and the pointer to this via the first one (RDI) and that is what we see here. The int data type is also 32-bit here.

The JMP instead of RET hack is also used here.

Listing 3.90: GCC 4.4.6 x64

```assembly
; default ctor
_ZN1cC2Ev:
    mov DWORD PTR [rdi], 667
    mov DWORD PTR [rdi+4], 999
    ret

; c(int a, int b)
_ZN1cC2Eii:
    mov DWORD PTR [rdi], esi
    mov DWORD PTR [rdi+4], edx
    ret

; dump()
_ZN1c4dumpEv:
    mov edx, DWORD PTR [rdi+4]
    mov esi, DWORD PTR [rdi]
    xor eax, eax
    mov edi, OFFSET FLAT:.LC0 ; "%d; %d\n"
    jmp printf
```

**Class inheritance**

Inherited classes are similar to the simple structures we already discussed, but extended in inheritable classes.

Let’s take this simple example:

```c
#include <stdio.h>

class object {
public:
    int color;
    object() { };
    object (int color) { this->color=color; };
    void print_color() { printf ("color=%d\n", color); };
};
```

[^26]: Also available as [https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf](https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf)
class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        }
        void dump()
        {
            printf ("this is a box. color=%d, width=%d, height=%d, depth=%d\n", color, width, \
                    height, depth);
        }
};

class sphere : public object
{
    private:
        int radius;
    public:
        sphere(int color, int radius)
        {
            this->color=color;
            this->radius=radius;
        }
        void dump()
        {
            printf ("this is sphere. color=%d, radius=%d\n", color, radius);
        }
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);
    b.print_color();
    s.print_color();
    b.dump();
    s.dump();
    return 0;
};

Let's investigate the generated code of the dump() functions/methods and also object::print_color(), and see the memory layout for the structures-objects (for 32-bit code).

So, here are the dump() methods for several classes, generated by MSVC 2008 with /0x and /0b0 options 27.

Listing 3.91: Optimizing MSVC 2008 /Ob0

27The /0b0 option stands for disabling inline expansion since function inlining can make our experiment harder.

??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@ @09GCEDOLPA@color?SDN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ; 'string'
?print_color@object@@QAEXXZ PROC ; object::print_color, COMDAT
    _this$ = ecx
    mov eax, DWORD PTR [ecx]
    push eax
    ; 'color=%d', 0aH, 00H
    push OFFSET ??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@ 
    call _printf
    add esp, 8
    ret 0

548
So, here is the memory layout:

(base class `object`)

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>int color</td>
</tr>
</tbody>
</table>

(inherited classes)

`box`:

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>int color</td>
</tr>
<tr>
<td>+0x4</td>
<td>int width</td>
</tr>
<tr>
<td>+0x8</td>
<td>int height</td>
</tr>
<tr>
<td>+0xC</td>
<td>int depth</td>
</tr>
</tbody>
</table>

`sphere`:

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>int color</td>
</tr>
<tr>
<td>+0x4</td>
<td>int radius</td>
</tr>
</tbody>
</table>

Let's see `main()` function body:

```c
PUBLIC _main
_TEXT SEGMENT
_s$ = -24 ; size = 8
_b$ = -16 ; size = 16
_main PROC
    sub esp, 24
    push 30
```
The inherited classes must always add their fields after the base classes’ fields, to make it possible for the base class methods to work with their own fields.

When the `object::print_color()` method is called, a pointers to both the `box` and `sphere` objects are passed as this, and it can work with these objects easily since the `color` field in these objects is always at the pinned address (at offset +0x0).

It can be said that the `object::print_color()` method is agnostic in relation to the input object type as long as the fields are pinned at the same addresses, and this condition is always true.

And if you create inherited class of the `box` class, the compiler will add the new fields after the `depth` field, leaving the `box` class fields at the pinned addresses.

Thus, the `box::dump()` method will work fine for accessing the `color`, `width`, `height` and `depths` fields, which are always pinned at known addresses.

The code generated by GCC is almost the same, with the sole exception of passing the this pointer (as it has been explained above, it is passed as the first argument instead of using the ECX register).

**Encapsulation**

Encapsulation is hiding the data in the private sections of the class, e.g. to allow access to them only from this class methods.

However, are there any marks in code the about the fact that some field is private and some other—not? No, there are no such marks.

Let's try this simple example:

```cpp
#include <stdio.h>

class box
{
  private:
    int color, width, height, depth;
  public:
    box(int color, int width, int height, int depth)
    {
      this->color=color;
      this->width=width;
      this->height=height;
      this->depth=depth;
    }
    void dump()
    {
      printf ("this is a box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, depth);
    }
};
```
Let’s compile it again in MSVC 2008 with `/Ox` and `/Ob0` options and see the `box::dump()` method code:

```assembly
?dump@box@@QAEXXZ ; box::dump, COMDAT
    _this$ = ecx
    mov eax, DWORD PTR [ecx+12]
    mov edx, DWORD PTR [ecx+8]
    push eax
    mov eax, DWORD PTR [ecx+4]
    push edx
    push eax
    push ecx
    'this is a box. color=%d, width=%d, height=%d, depth=%d', 0aH, 00H
    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?$Cf?0?5width?$DN?$Cf?0@
    call _printf
    add esp, 20
    ret 0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Here is a memory layout of the class:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>int color</td>
</tr>
<tr>
<td>+0x4</td>
<td>int width</td>
</tr>
<tr>
<td>+0x8</td>
<td>int height</td>
</tr>
<tr>
<td>+0xC</td>
<td>int depth</td>
</tr>
</tbody>
</table>

All fields are private and not allowed to be accessed from any other function, but knowing this layout, can we create code that modifies these fields?

To do this we’ll add the `hack_oop_encapsulation()` function, which is not going to compile if it looked like this:

```cpp
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // that code can't be compiled:
        // 'error C2248: 'box::width' : cannot access private member declared in class
}
```

Nevertheless, if we cast the `box` type to a `pointer to an int array`, and we modify the array of `int`s that we have, we can succeed.

```cpp
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(o);
    ptr_to_object[1]=123;
}
```

This function’s code is very simple—it can be said that the function takes a pointer to an array of `int`s for input and writes `123` to the second `int`:

```assembly
?hack_oop_encapsulation@@YAXPAVbox@@Z ; hack_oop_encapsulation
    mov eax, DWORD PTR _o$[esp-4]
    mov DWORD PTR [eax+4], 123
    ret 0
?hack_oop_encapsulation@@YAXPAVbox@@Z ENDP ; hack_oop_encapsulation
```

Let’s check how it works:

```cpp
int main()
{
    box b(1, 10, 20, 30);
    b.dump();
    hack_oop_encapsulation(&b);
}```
b.dump();
return 0;
};

Let's run:
this is a box. color=1, width=10, height=20, depth=30
this is a box. color=1, width=123, height=20, depth=30

We see that the encapsulation is just protection of class fields only in the compilation stage.
The C++ compiler is not allowing the generation of code that modifies protected fields straightforwardly, nevertheless, it is possible with the help of dirty hacks.

Multiple inheritance
Multiple inheritance is creating a class which inherits fields and methods from two or more classes.
Let's write a simple example again:

```c
#include <stdio.h>

class box {
    public:
        int width, height, depth;
        box() { };
        box(int width, int height, int depth) {
            this->width=width;
            this->height=height;
            this->depth=depth;
        }
        void dump() {
            printf("this is a box. width=%d, height=%d, depth=%d\n", width, height, depth);
        }
        int get_volume() {
            return width * height * depth;
        }
};
class solid_object {
    public:
        int density;
        solid_object() { };
        solid_object(int density) {
            this->density=density;
        }
        int get_density() {
            return density;
        }
        void dump() {
            printf("this is a solid_object. density=%d\n", density);
        }
};
class solid_box: box, solid_object {
    public:
        solid_box (int width, int height, int depth, int density) {
            this->width=width;
            this->height=height;
```
void dump()
{
    printf("this is a solid_box. width=%d, height=%d, depth=%d, density=%d\n", width, height, depth, density);
}

int get_weight() { return get_volume() * get_density(); }

int main()
{
    box b(10, 20, 30);
    solid_object so(100);
    solid_box sb(10, 20, 30, 3);

    b.dump();
    so.dump();
    sb.dump();
    printf("%d\n", sb.get_weight());

    return 0;
}

Let's compile it in MSVC 2008 with the /0x and /Ob0 options and see the code of box::dump(), solid_object::dump() and solid_box::dump():

Listing 3.95: Optimizing MSVC 2008 /Ob0

Listing 3.96: Optimizing MSVC 2008 /Ob0

Listing 3.97: Optimizing MSVC 2008 /Ob0
So, the memory layout for all three classes is:

**box** class:

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>width</td>
</tr>
<tr>
<td>+0x4</td>
<td>height</td>
</tr>
<tr>
<td>+0x8</td>
<td>depth</td>
</tr>
</tbody>
</table>

**solid_object** class:

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>density</td>
</tr>
</tbody>
</table>

It can be said that the **solid_box** class memory layout is *united*:

**solid_box** class:

<table>
<thead>
<tr>
<th>offset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0x0</td>
<td>width</td>
</tr>
<tr>
<td>+0x4</td>
<td>height</td>
</tr>
<tr>
<td>+0x8</td>
<td>depth</td>
</tr>
<tr>
<td>+0xC</td>
<td>density</td>
</tr>
</tbody>
</table>

The code of the **box::get_volume()** and **solid_object::get_density()** methods is trivial:

**Listing 3.98: Optimizing MSVC 2008 /Ob0**

```assembly
?get_volume@box@@QAEXXZ PROC ; box::get_volume, COMDAT
    ; _this$ = ecx
    mov eax, DWORD PTR [ecx+8]
    imul eax, DWORD PTR [ecx+4]
    imul eax, DWORD PTR [ecx]
    ret 0
?get_volume@box@@QAEXXZ ENDP ; box::get_volume
```

**Listing 3.99: Optimizing MSVC 2008 /Ob0**

```assembly
?get_density@solid_object@@QAEXXZ PROC ; solid_object::get_density, COMDAT
    ; _this$ = ecx
    mov eax, DWORD PTR [ecx]
    ret 0
?get_density@solid_object@@QAEXXZ ENDP ; solid_object::get_density
```

But the code of the **solid_box::get_weight()** method is much more interesting:

**Listing 3.100: Optimizing MSVC 2008 /Ob0**

```assembly
?get_weight@solid_box@@QAEXXZ PROC ; solid_box::get_weight, COMDAT
    ; _this$ = ecx
    push esi
    mov esi, ecx
    push edi
    lea ecx, DWORD PTR [esi+12]
    call ?get_density@solid_object@@QAEXXZ ; solid_object::get_density
    mov ecx, esi
    mov edi, eax
    call ?get_volume@box@@QAEXXZ ; box::get_volume
    imul eax, edi
    pop edi
    pop esi
    ret 0
?get_weight@solid_box@@QAEXXZ ENDP ; solid_box::get_weight
```
get_weight() just calls two methods, but for get_volume() it just passes pointer to this, and for get_density() it passes a pointer to this incremented by 12 (or 0xC) bytes, and there, in the solid_box class memory layout, the fields of the solid_object class start.

Thus, the solid_object::get_density() method will believe like it is dealing with the usual solid_object class, and the box::get_volume() method will work with its three fields, believing this is just the usual object of class box.

Thus, we can say, an object of a class, that inherits from several other classes, is representing in memory as a *united* class, that contains all inherited fields. And each inherited method is called with a pointer to the corresponding structure’s part.

**Virtual methods**

Yet another simple example:

```c
#include <stdio.h>

class object
{
  public:
    int color;
  object() { };
  object (int color) { this->color=color; };
  virtual void dump()
  {
    printf ("color=%d\n", color);
  };
};

class box : public object
{
  private:
    int width, height, depth;
  public:
    box(int color, int width, int height, int depth)
    {
      this->color=color;
      this->width=width;
      this->height=height;
      this->depth=depth;
    };
    void dump()
    {
      printf ("this is a box. color=%d, width=%d, height=%d, depth=%d\n", color, width, height, depth);
    };
};

class sphere : public object
{
  private:
    int radius;
  public:
    sphere(int color, int radius)
    {
      this->color=color;
      this->radius=radius;
    };
    void dump()
    {
      printf ("this is sphere. color=%d, radius=%d\n", color, radius);
    };
};

int main()
{
  box b(1, 10, 20, 30);
  sphere s(2, 40);
}  
```

Class object has a virtual method dump() that is being replaced in the inheriting box and sphere classes. If we are in an environment where it is not known the type of an object, as in the main() function in example, where the virtual method dump() is called, the information about its type must be stored somewhere, to be able to call the relevant virtual method.

Let's compile it in MSVC 2008 with the /Ox and /Ob0 options and see the code of main():

```c
object *o1=&b;
object *o2=&s;
o1->dump();
o2->dump();
return 0;
};
```

A pointer to the dump() function is taken somewhere from the object. Where could we store the address of the new method? Only somewhere in the constructors: there is no other place since nothing else is called in the main() function.

Let's see the code of the constructor of the box class:

```plaintext
??_R0?AVBox@@@8 DD FLAT:??_7type_info@6B@ ; box 'RTTI Type Descriptor'
  DD 00H
  DB '.?AVBox@@', 00H

??_R1A@0@?AVBox@@8 DD FLAT:??_R0?AVBox@@@8 ; box:'RTTI Base Class Descriptor at (0,-1,0,64)'
  DD 01H
  DD 00H
  DD 0fffffffH
  DD 00H
  DD 040H
  DD FLAT:??_R3box@8

??_R2box@8 DD FLAT:??_R1A@0@EA@box@8 ; box:'RTTI Base Class Array'
  DD FLAT:??_R1A@0@EA@object@8

??_R3box@8 DD 00H ; box:'RTTI Class Hierarchy Descriptor'
  DD 00H
  DD 02H
  DD FLAT:??_R2box@8
```

You can read more about pointers to functions in the relevant section: (1.33 on page 381)
Here we see a slightly different memory layout: the first field is a pointer to some table box::'vftable' (the name has been set by the MSVC compiler).

In this table we see a link to a table named box::'RTTI Complete Object Locator' and also a link to the box::dump() method.

These are called virtual methods table and RTTI\(^{29}\). The table of virtual methods has the addresses of methods and the RTTI table contains information about types.

By the way, the RTTI tables are used while calling \texttt{dynamic\_cast} and \texttt{typeid} in C++. You can also see here the class name as a plain text string.

Thus, a method of the base \texttt{object} class may call the virtual method \texttt{object::dump()}, which in turn will call a method of an inherited class, since that information is present right in the object’s structure.

Some additional CPU time is needed for doing look-ups in these tables and finding the right virtual method address, thus virtual methods are widely considered as slightly slower than common methods.

In GCC-generated code the RTTI tables are constructed slightly differently.

### 3.21.2 \texttt{ostream}

Let’s start again with a “hello world” example, but now we are going to use \texttt{ostream}:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Almost any C++ textbook tells us that the \texttt{\langle \rangle} operation can be defined (overloaded) for other types. That is what is done in \texttt{ostream}. We see that operator\texttt{\langle \rangle} is called for \texttt{ostream}:

\(^{29}\text{Run-Time Type Information}\)
Let's modify the example:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, " << "world!\n";
}
```

And again, from many C++ textbooks we know that the result of each operator<< in ostream is forwarded to the next one. Indeed:

```
Listing 3.102: MSVC 2012
```

If we would rename operator<< method name to `f()`, that code will looks like:

```cpp
f(f(std::cout, "Hello, "), "world!");
```

GCC generates almost the same code as MSVC.

### 3.21.3 References

In C++, references are pointers (3.23 on page 596) as well, but they are called safe, because it is harder to make a mistake while dealing with them (C++11 8.3.2).

For example, reference must always be pointing to an object of the corresponding type and cannot be NULL [Marshall Cline, C++ FAQ8.6].

Even more than that, references cannot be changed, it is impossible to point them to another object (reseat) [Marshall Cline, C++ FAQ8.5].

If we are going to change the example with pointers (3.23 on page 596) to use references instead ...

```cpp
void f2 (int x, int y, int &sum, int &product)
{
    sum=x+y;
    product=x*y;
}
```
...then we can see that the compiled code is just the same as in the pointers example (3.23 on page 596):

Listing 3.103: Optimizing MSVC 2010

```c
#include <string>
#include <stdio.h>
```

The reason why C++ functions has such strange names is explained here: 3.21.1 on page 542.) Hence, C++ references are as much efficient as usual pointers.

### 3.21.4 STL

N.B.: all examples here were checked only in 32-bit environment. x64 wasn’t checked.

**std::string**

**Internals**

Many string libraries [Dennis Yurichev, *C/C++ programming language notes*2.2] implement a structure that contains a pointer to a string buffer, a variable that always contains the current string length (which is very convenient for many functions: [Dennis Yurichev, *C/C++ programming language notes*2.2.1]) and a variable containing the current buffer size.

The string in the buffer is usually terminated with zero, in order to be able to pass a pointer to the buffer into the functions that take usual C **ASCII** strings.

It is not specified in the C++ standard how std::string has to be implemented, however, it is usually implemented as explained above.

The C++ string is not a class (as QString in Qt, for instance) but a template (basic_string), this is made in order to support various character types: at least `char` and `wchar_t`.

So, std::string is a class with `char` as its base type.

And std::wstring is a class with `wchar_t` as its base type.

**MSVC**

The MSVC implementation may store the buffer in place instead of using a pointer to a buffer (if the string is shorter than 16 symbols).

This implies that a short string is to occupy at least \(16 + 4 + 4 = 24\) bytes in 32-bit environment or at least \(16 + 8 + 8 = 32\) bytes in 64-bit one, and if the string is longer than 16 characters, we also have to add the length of the string itself.

Listing 3.104: example for MSVC

```c
#include <string>
#include <stdio.h>
```
Almost everything is clear from the source code.

A couple of notes:

If the string is shorter than 16 symbols, a buffer for the string is not to be allocated in the heap.

This is convenient because in practice, a lot of strings are short indeed.

Looks like that Microsoft’s developers chose 16 characters as a good balance.

One very important thing here can be seen at the end of main(): we’re not using the c_str() method, nevertheless, if we compile and run this code, both strings will appear in the console!

This is why it works.

In the first case the string is shorter than 16 characters and the buffer with the string is located in the beginning of the std::string object (it can be treated as a structure). printf() treats the pointer as a pointer to the null-terminated array of characters, hence it works.

Printing the second string (longer than 16 characters) is even more dangerous: it is a typical programmer’s mistake (or typo) to forget to write c_str().

This works because at the moment a pointer to buffer is located at the start of structure.

This may stay unnoticed for a long time, until a longer string appears there at some time, then the process will crash.

**GCC**

GCC’s implementation of this structure has one more variable—reference count.

One interesting fact is that in GCC, a pointer to an instance of std::string instance points not to the beginning of the structure, but to the buffer pointer. In `libstdc++-v3/include/bits/basic_string.h` we can read that it was done for more convenient debugging:
We consider this in our example:

```cpp
#include <string>
#include <stdio.h>

struct std_string
{
    size_t length;
    size_t capacity;
    size_t refcount;
};

void dump_std_string(std::string s)
{
    char *p1=*(char**)&s; // GCC type checking workaround
    struct std_string *p2=(struct std_string*)(p1-sizeof(struct std_string));
    printf("[%s] size:%d capacity:%d\n", p1, p2->length, p2->capacity);
}

int main()
{
    std::string s1="a short string";
    std::string s2="a string longer than 16 bytes";
    dump_std_string(s1);
    dump_std_string(s2);

    // GCC type checking workaround:
    printf("%s\", *(char**)&s1);
    printf("%s\", *(char**)&s2);
}
```

A trickery has to be used to imitate the mistake we already have seen above because GCC has stronger type checking, nevertheless, printf() works here without c_str() as well.

A more advanced example

```cpp
#include <string>
#include <stdio.h>

int main()
{
    std::string s1="Hello, ";
    std::string s2="world!\n";
    std::string s3=s1+s2;

    printf("%s\n", s3.c_str());
}
```

Listing 3.106: MSVC 2012

```
$SG39512 DB 'Hello, ', 00H
$SG39514 DB 'world!', 0aH, 00H
$SG39581 DB '%s', 0aH, 00H

_s2$ = -72 ; size = 24
_s3$ = -48 ; size = 24
_s1$ = -24 ; size = 24
_main PROC
    sub esp, 72
    push 7
    push OFFSET $SG39512
    lea ecx, DWORD PTR _s1$[esp+80]
    mov DWORD PTR _s1$[esp+100], 15
```

561
The compiler does not construct strings statically: it would not be possible anyway if the buffer needs to be located in the heap.

Instead, the ASCII strings are stored in the data segment, and later, at runtime, with the help of the “assign” method, the s1 and s2 strings are constructed. And with the help of operator+, the s3 string is constructed.

Please note that there is no call to the c_str() method, because its code is tiny enough so the compiler
inlined it right there: if the string is shorter than 16 characters, a pointer to buffer is left in EAX, otherwise the address of the string buffer located in the heap is fetched.

Next, we see calls to the 3 destructors, they are called if the string is longer than 16 characters: then the buffers in the heap have to be freed. Otherwise, since all three std::string objects are stored in the stack, they are freed automatically, when the function ends.

As a consequence, processing short strings is faster, because of less heap accesses.

GCC code is even simpler (because the GCC way, as we saw above, is to not store shorter strings right in the structure):

```
Listing 3.107: GCC 4.8.1

.LC0:
    .string "Hello,"
.LC1:
    .string "world!\n"
main:
    push ebp
    mov ebp, esp
    push esi
    push ebx
    and esp, -16
    sub esp, 32
    lea ebx, [esp+28]
    lea edi, [esp+20]
    mov DWORD PTR [esp+8], ebx
    lea esi, [esp+24]
    mov DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov DWORD PTR [esp], edi
    call _ZNSsC1EPKcRKSaIcE
    mov DWORD PTR [esp+8], ebx
    mov DWORD PTR [esp+4], OFFSET FLAT:.LC1
    mov DWORD PTR [esp], esi
    call _ZNSsC1EPKcRKSaIcE
    mov DWORD PTR [esp+4], edi
    mov DWORD PTR [esp], ebx
    call _ZNSsC1ERKSs
    mov DWORD PTR [esp+4], esi
    mov DWORD PTR [esp], ebx
    call _ZNSs6appendERKSs

; inlined c_str():
    mov eax, DWORD PTR [esp+28]
    mov DWORD PTR [esp], eax
    call puts
    mov eax, DWORD PTR [esp+28]
    lea ebx, [esp+19]
    mov DWORD PTR [esp+4], ebx
    sub eax, 12
    mov DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    mov eax, DWORD PTR [esp+24]
    mov DWORD PTR [esp+4], ebx
    sub eax, 12
    mov DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    mov eax, DWORD PTR [esp+20]
    mov DWORD PTR [esp+4], ebx
    sub eax, 12
    mov DWORD PTR [esp], eax
```
It can be seen that it’s not a pointer to the object that is passed to destructors, but rather an address 12 bytes (or 3 words) before, i.e., a pointer to the real start of the structure.

**std::string as a global variable**

Experienced C++ programmers knows that global variables of STL\textsuperscript{30} types can be defined without problems.

Yes, indeed:

```c
#include <stdio.h>
#include <string>

std::string s = "a string";

int main()
{
    printf ("%s\n", s.c_str());
}
```

But how and where std::string constructor will be called?

In fact, this variable is to be initialized even before main() start.

Listing 3.108: MSVC 2012: here is how a global variable is constructed and also its destructor is registered

```assembly
??_Es@YAXXZ PROC
    push 8
    push OFFSET $SG39512 ; 'a string'
    mov ecx, OFFSET ?@3V?$basic_string@DU?$char_traits@V?$allocator@D@2@@std@@A ; s
    call ?assign@@?$basic_string@DU?$char_traits@V?$allocator@D@2@@std@@@std@@QAEAAV12@PBDI@Z
    push OFFSET ??_Fs@YAXXZ ; 'dynamic atexit destructor for 's'"
    call _atexit
    pop ecx
    ret
??_Es@YAXXZ ENDP
```

Listing 3.109: MSVC 2012: here a global variable is used in main()

```assembly
SG39512 DB 'a string', 00H
SG39519 DB '%s', 0aH, 00H

_main PROC
    cmp DWORD PTR ?@3V?$basic_string@DU?$char_traits@V?$allocator@D@2@@std@@A+20, 16
    mov eax, OFFSET ?@3V?$basic_string@DU?$char_traits@V?$allocator@D@2@@std@@A ; s
    cmovae eax, DWORD PTR ?@3V?$basic_string@DU?$char_traits@V?$allocator@D@2@@std@@A
    push eax
    push OFFSET $SG39519 ; '%s'
    call _printf
    add esp, 8
    xor eax, eax
    ret 0
_main ENDP
```

Listing 3.110: MSVC 2012: this destructor function is called before exit

```assembly
??_Fs@YAXXZ PROC
    push ecx
```

\textsuperscript{30}(C++) Standard Template Library
In fact, a special function with all constructors of global variables is called from CRT, before main().

More than that: with the help of atexit() another function is registered, which contain calls to all destructors of such global variables.

GCC works likewise:

```
Listing 3.111: GCC 4.8.1
```

```
main:
push ebp
mov ebp, esp
and esp, .16
sub esp, 16
mov eax, DWORD PTR s
mov DWORD PTR [esp], eax
call puts
xor eax, eax
leave
ret

.LC0:
.string "a string"

GLOBAL__sub_I_s:
  sub esp, 44
  lea eax, [esp+31]
  mov DWORD PTR [esp+8], eax
  mov DWORD PTR [esp+4], OFFSET FLAT:.LC0
  mov DWORD PTR [esp], OFFSET FLAT:s
call _ZN5P1EpKcRKSaIcE
  mov DWORD PTR [esp+8], OFFSET FLAT:_dso_handle
  mov DWORD PTR [esp+4], OFFSET FLAT:s
  mov DWORD PTR [esp], OFFSET FLAT:_ZNSsD1Ev
call __cxa_atexit
add esp, 44
ret

.LFE645:

.size GLOBAL__sub_I_s, ._GLOBAL__sub_I_s
.section .init_array,"aw"
.align 4
.long GLOBAL__sub_I_s
globl s
.bss
.align 4
type s, @object
.size s, 4
s:
  .zero 4
  .hidden _dso_handle
```
But it does not create a separate function for this, each destructor is passed to atexit(), one by one.

**std::list**

This is the well-known doubly-linked list: each element has two pointers, to the previous and next elements.

This implies that the memory footprint is enlarged by 2 words for each element (8 bytes in 32-bit environment or 16 bytes in 64-bit).

C++ STL just adds the “next” and “previous” pointers to the existing structure of the type that you want to unite in a list.

Let’s work out an example with a simple 2-variable structure that we want to store in a list.

Although the C++ standard does not say how to implement it, both MSVC’s and GCC’s implementations are straightforward and similar, so here is only one source code for both:

```cpp
#include <stdio.h>
#include <list>
#include <iostream>

struct a {
    int x;
    int y;
};

struct List_node {
    struct List_node* _Next;
    struct List_node* _Prev;
    int x;
    int y;
};

void dump_List_node (struct List_node *n) {
    printf ("ptr=0x%p _Next=0x%p _Prev=0x%p x=%d y=%d\n",
            n, n->_Next, n->_Prev, n->x, n->y);
}

void dump_List_vals (struct List_node* n) {
    struct List_node* current=n;
    for (;;) {
        dump_List_node (current);
        current=current->_Next;
        if (current==n) // end
            break;
    }
}

void dump_List_val (unsigned int *a) {
    #ifdef _MSC_VER
    // GCC implementation does not have "size" field
    printf ("_Myhead=0x%p, _Mysize=%d\n", a[0], a[1]);
    #endif
    dump_List_vals ((struct List_node*)a[0]);
}

int main() {
    std::list<struct a> l;
    printf ("* empty list:\n");
    dump_List_val((unsigned int*)(void*)&l);
}
GCC

Let’s start with GCC.

When we run the example, we’ll see a long dump, let’s work with it in pieces.

* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0

Here we see an empty list.

Despite the fact it is empty, it has one element with garbage (aka dummy node) in x and y. Both the “next” and “prev” pointers are pointing to the self node:
At this moment, the .begin and .end iterators are equal to each other.

If we push 3 elements, the list internally will be:

```
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x0003b40 _Prev=0x000349a0 x=1 y=2
ptr=0x0003b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x0003b40 x=5 y=6
```

The last element is still at 0x0028fe90, it not to be moved until the list’s disposal.

It still contain random garbage in \(x\) and \(y\) (5 and 6). By coincidence, these values are the same as in the last element, but it doesn’t mean that they are meaningful.

Here is how these 3 elements are stored in memory:

The \(l\) variable always points to the first node.

The .begin() and .end() iterators are not variables, but functions, which when called return pointers to the corresponding nodes.

Having a dummy element (AKA sentinel node) is a very popular practice in implementing doubly-linked lists.

Without it, a lot of operations may become slightly more complex and, hence, slower.

The iterator is in fact just a pointer to a node. list.begin() and list.end() just return pointers.

```
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x0003b40 x=5 y=6
```

The fact that the last element has a pointer to the first and the first element has a pointer to the last one remind us of circular lists.
This is very helpful here: having a pointer to the first list element, i.e., that is in the \( l \) variable, it is easy to get a pointer to the last one quickly, without the necessity to traverse the whole list.

Inserting an element at the end of the list is also quick, thanks to this feature.

\texttt{operator--} and \texttt{operator++} just set the current iterator’s value to the \texttt{current_node->prev} or \texttt{current_node->next} values.

The reverse iterators (.\texttt{rbegin}, .\texttt{rend}) work just as the same, but in reverse.

\texttt{operator*} just returns a pointer to the point in the node structure, where the user’s structure starts, i.e., a pointer to the first element of the structure \( x \).

The list insertion and deletion are trivial: just allocate a new node (or deallocate) and update all pointers to be valid.

That’s why an iterator may become invalid after element deletion: it may still point to the node that has been already deallocated. This is also called a \textit{dangling pointer}.

And of course, the information from the freed node (to which iterator still points) cannot be used anymore.

The GCC implementation (as of 4.8.1) doesn’t store the current size of the list: this implies a slow \texttt{.size()} method: it has to traverse the whole list to count the elements, because it doesn’t have any other way to get the information.

This means that this operation is \( O(n) \), i.e., it steadily gets slower as the list grows.

\textbf{Listing 3.112: Optimizing GCC 4.8.1 -fno-inline-small-functions}

\begin{verbatim}
main proc near
    push ebp
    mov ebp, esp
    push esi
    push ebx
    and esp, 0FFFFFFFF0h
    sub esp, 20h
    lea ebx, [esp+10h]
    mov dword ptr [esp], offset s ; "* empty list:"
    mov [esp+10h], ebx
    mov [esp+14h], ebx
    call puts
    mov [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    lea esi, [esp+10h]
    mov [esp+4], esi
    mov [esp], ebx
    mov dword ptr [esp+18h], 1 ; X for new element
    mov dword ptr [esp+10h], ebx
    mov dword ptr [esp+1Ch], 2 ; Y for new element
    call _ZNSt4listIaSaIS0_E10push_frontERKS0_ ;
    std::list<a,std::allocator<a>>::push_front(a const&)
    mov [esp+4], esi
    mov [esp], ebx
    mov dword ptr [esp+18h], 3 ; X for new element
    mov dword ptr [esp+1Ch], 4 ; Y for new element
    call _ZNSt4listIaSaIS0_E10push_frontERKS0_ ;
    std::list<a,std::allocator<a>>::push_front(a const&)
    mov dword ptr [esp], 10h
    mov dword ptr [esp+10h], 5 ; X for new element
    mov dword ptr [esp+1Ch], 6 ; Y for new element
    call _Znwj ; operator new(uint)
    cmp eax, 0FFFFFFFF8h
    jz short loc_80002A6
    jz short loc_80002A6
    mov [eax+0Ch], ecx
    mov [eax+8], edx
    loc_80002A6: ; CODE XREF: main+86
    mov [esp+4], ebx
    mov [esp], eax
    call _ZNSt8_detail15_List_node_base7_M_hookEPS0_ ;
    std::__detail::List_node_base::M_hook(std::__detail::List_node_base*)
    mov dword ptr [esp], offset a3ElementsList ; "* 3-elements list:"
    call puts
\end{verbatim}
mov [esp], ebx

call Istdump_LsitvalPj : dump List val(uint *)
mov dword ptr [esp], offset aNodeAt_begin ; "node at .begin:"
call puts
mov eax, [esp+10h]
mov [esp], eax
call Istdump_LsitnodeP9List_node ; dump List node(List_node *)
mov dword ptr [esp], offset aNodeAt_end ; "node at .end:"
call puts
mov [esp], ebx

mov [esp], ebx
call Istdump_LsitnodeP9List_node ; dump List node(List_node *)
mov dword ptr [esp], offset aLetSCountFromT ; "* let's count from the beginning:"
call puts
mov esi, [esp+10h]
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a1stElementDD ; "1st element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esi]

operator++: get ->next pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov eax, [esi]

operator++: get ->next pointer
mov edx, [eax+0Ch]
mov [esp+0Ch], edx
mov eax, [eax+8]
mov dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov eax, [esi]

operator++: get ->next pointer
mov edx, [eax+0Ch]
mov [esp+0Ch], edx
mov eax, [eax+8]
mov dword ptr [esp+4], offset aElementAt_endD ; "element at .end: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov dword ptr [esp], offset aLetSCountFro_0 ; "* let's count from the end:"
call puts
mov eax, [esp+1Ch]
mov dword ptr [esp+4], offset aElementAt_endD ; "element at .end: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esp+14h]
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a3rdElementDD ; "3rd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
mov esi, [esi+4] ; operator--: get ->prev pointer
mov eax, [esi+0Ch]
mov [esp+0Ch], eax
mov eax, [esi+8]
mov dword ptr [esp+4], offset a2ndElementDD ; "2nd element: %d %d\n"
mov dword ptr [esp], 1
mov [esp+8], eax
call __printf_chk
msvc

MSVC’s implementation (2012) is just the same, but it also stores the current size of the list. This implies that the .size() method is very fast ($O(1)$): it just reads one value from memory.

On the other hand, the size variable must be updated at each insertion/deletion.

MSVC’s implementation is also slightly different in the way it arranges the nodes:
GCC has its dummy element at the end of the list, while MSVC’s is at the beginning.

Listing 3.114: Optimizing MSVC 2012 /Fa2.asm /GS- /Ob1

```asm
l$ = -16 ; size = 8
_t1$ = -8 ; size = 8

_main PROC
sub esp, 16
push ebx
push esi
push edi
push 0
push 0
lea ecx, DWORD PTR _l$[esp+36]
mov DWORD PTR _l$[esp+40], 0
; allocate first garbage element
call ?_Buynode0@_7_list_alloc@$0A@U_a@@V_std::list_base_types<a,std::allocator<a> >@a@@@Z
push esi
lea DWORD PTR __imp__printf[esp+32], ebx
call edi ; printf
lea eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@YAXPAIA@Z ; dump_List_val
mov esi, DWORD PTR [ebx]
add esp, 8
lea eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [esi+4]
lea ecx, DWORD PTR _l$[esp+36]
push esi
mov DWORD PTR _t1$[esp+40], 1 ; data for a new node
mov DWORD PTR _t1$[esp+44], 2 ; data for a new node
; allocate new node
call ?$_Buynode@_7_list_buy@$0A@U_a@@V_std::list_base_types<a,std::allocator<a> >@a@@@Z
lea ecx, DWORD PTR _l$[esp+36]
mov esi
push eax
push DWORD PTR [esi+4]
lea eax, DWORD PTR _t1$[esp+28]
push eax
lea eax, DWORD PTR _t1$[esp+44]
push eax
lea ecx, DWORD PTR _l$[esp+36]
push esi
```
allocate new node

```c
std::_List_buy<a,std::allocator<a> >::_Buynode<a const &>

mov DWORD PTR [esi+4], eax
mov ecx, DWORD PTR [eax+4]
mov DWORD PTR _t1$[esp+28], 5 ; data for a new node
mov DWORD PTR [ecx], eax
lea eax, DWORD PTR _t1$[esp+28]
push eax
push DWORD PTR [ebx+4]
lea ecx, DWORD PTR _l$[esp+36]
push ebx
mov DWORD PTR _t1$[esp+44], 6 ; data for a new node
; allocate new node

call ??$_List_buy@Ua@V?allocator@Ua@@std@@std@@QAEPAX@@PAU21@0@PAUa@@@Z
std::List_buy<a,std::allocator<a> >::_Buynode<a const &>

mov DWORD PTR [ebx+4], eax
mov ecx, DWORD PTR [eax+4]
push OFFSET SSG40689 ; '* 3-elements list:'
mov DWORD PTR _l$[esp+36], 3
mov DWORD PTR [ecx], eax
call edi; printf
lea eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@YAXPA@Z ; dump_List_val
push OFFSET SSG40831 ; 'node at .begin:'
call edi; printf
push DWORD PTR [ebx]; get next field of node "l" variable points to

call ??$_List_buy@Ua@V?allocator@Ua@@std@@std@@QAEPAX@@PAU21@0@PAUa@@@Z
std::List_buy<a,std::allocator<a> >::_Buynode<a const &>

mov DWORD PTR [ebx+4], eax
mov ecx, DWORD PTR [eax+4]
push OFFSET SSG40689 ; '* 3-elements list:'
mov DWORD PTR _l$[esp+36], 3
mov DWORD PTR [ecx], eax
call edi; printf
lea eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@YAXPA@Z ; dump_List_val
push OFFSET SSG40831 ; 'node at .begin:'
call edi; printf
push DWORD PTR [ebx]; get next field of node "l" variable points to

call ??$_List_buy@Ua@V?allocator@Ua@@std@@std@@QAEPAX@@PAU21@0@PAUa@@@Z
std::List_buy<a,std::allocator<a> >::_Buynode<a const &>

mov DWORD PTR [ebx+4], eax
mov ecx, DWORD PTR [eax+4]
push OFFSET SSG40689 ; '* 3-elements list:'
mov DWORD PTR _l$[esp+36], 3
mov DWORD PTR [ecx], eax
call edi; printf
lea eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@YAXPA@Z ; dump_List_val
push OFFSET SSG40831 ; 'node at .begin:'
call edi; printf
push DWORD PTR [ebx]; get next field of node "l" variable points to

call ??$_List_buy@Ua@V?allocator@Ua@@std@@std@@QAEPAX@@PAU21@0@PAUa@@@Z
std::List_buy<a,std::allocator<a> >::_Buynode<a const &>

mov DWORD PTR [ebx+4], eax
mov ecx, DWORD PTR [eax+4]
push OFFSET SSG40689 ; '* 3-elements list:'
mov DWORD PTR _l$[esp+36], 3
mov DWORD PTR [ecx], eax
call edi; printf
lea eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@YAXPA@Z ; dump_List_val
push OFFSET SSG40831 ; 'node at .begin:'
call edi; printf
push DWORD PTR [ebx]; get next field of node "l" variable points to

mov esi, DWORD PTR [esi+4]; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET SSG40846 ; '1st element: %d %d'
call edi; printf
mov esi, DWORD PTR [esi+4]; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET SSG40848 ; '2nd element: %d %d'
call edi; printf
mov esi, DWORD PTR [esi+4]; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET SSG40850 ; '3rd element: %d %d'
call edi; printf
mov eax, DWORD PTR [esi+4]; operator++: get ->next pointer
add esp, 64
push DWORD PTR [eax+12]
push DWORD PTR [eax+8]
push OFFSET SSG40852 ; 'element at .end(): %d %d'
call edi; printf
push OFFSET SSG40853 ; '* let''s count from the begin:'
call edi; printf
push DWORD PTR [ebx+12]; use x and y fields from the node "l" variable points to
push DWORD PTR [ebx+8]
push OFFSET SSG40860 ; 'element at .end(): %d %d'
call edi; printf
mov esi, DWORD PTR [ebx+4]; operator--: get ->prev pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET SSG40862 ; '3rd element: %d %d'
call edi; printf
mov esi, DWORD PTR [esi+4]; operator--: get ->prev pointer
```

573
Unlike GCC, MSVC’s code allocates the dummy element at the start of the function with the help of the “Buynode” function, it is also used to allocate the rest of the nodes (GCC’s code allocates the first element in the local stack).

Listing 3.115: The whole output

* empty list:
  _Myhead=0x003CC258, _Mysize=0
  ptr=0x003CC258 _Next=0x003CC258 _Prev=0x003CC258 x=6226002 y=4522072

* 3-elements list:

574
**C++11 std::forward_list**

The same thing as std::list, but singly-linked one, i.e., having only the “next” field at each node. It has a smaller memory footprint, but also don’t offer the ability to traverse list backwards.

**std::vector**

We would call std::vector a safe wrapper of the PODT\(^{31}\) C array. Internally it is somewhat similar to std::string (3.21.4 on page 559): it has a pointer to the allocated buffer, a pointer to the end of the array, and a pointer to the end of the allocated buffer.

The array’s elements lie in memory adjacently to each other, just like in a normal array (1.26 on page 264). In C++11 there is a new method called .data(), that returns a pointer to the buffer, like .c_str() in std::string.

The buffer allocated in the heap can be larger than the array itself.

Both MSVC’s and GCC’s implementations are similar, just the names of the structure’s fields are slightly different\(^ {32}\), so here is one source code that works for both compilers. Here is again the C-like code for dumping the structure of std::vector:

```c
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <functional>

struct vector_of_ints
{
  // MSVC names:
  int *Myfirst;
  int *Mylast;
  int *Myend;

  // GCC structure is the same, but names are: _M_start, _M_finish, _M_end_of_storage
};

void dump(struct vector_of_ints *in)
{
  printf("_Myfirst%p, _Mylast%p, _Myend%p\n", in->Myfirst, in->Mylast, in->Myend);
  size_t size=(in->Mylast-in->Myfirst);
}
```

\(^{31}\)(C++) Plain Old Data Type

\(^{32}\)GCC internals: http://go.yurichev.com/17086
size_t capacity=(in->Myend.in->Myfirst);
printf ("size=%d, capacity=%d\n", size, capacity);
for (size_t i=0; i<size; i++)
  printf ("element %d: %d\n", i, in->Myfirst[i]);
}

int main()
{
  std::vector<int> c;
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(1);
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(2);
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(3);
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(4);
dump ((struct vector_of_ints*)(void*)&c);
c.reserve (6);
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(5);
dump ((struct vector_of_ints*)(void*)&c);
c.push_back(6);
dump ((struct vector_of_ints*)(void*)&c);
printf ("%d\n", c.at(5)); // with bounds checking
printf ("%d\n", c[8]); // operator[], without bounds checking
}
As it can be seen, there is no allocated buffer when `main()` starts. After the first `push_back()` call, a buffer is allocated. And then, after each `push_back()` call, both array size and buffer size (capacity) are increased. But the buffer address changes as well, because `push_back()` reallocates the buffer in the heap each time. It is costly operation, that's why it is very important to predict the size of the array in the future and reserve enough space for it with the `.reserve()` method.

The last number is garbage: there are no array elements at this point, so a random number is printed. This illustrates the fact that `operator[]` of `std::vector` does not check of the index is in the array's bounds. The slower `.at()` method, however, does this checking and throws an `std::out_of_range` exception in case of error.

Let's see the code:

Listing 3.116: MSVC 2012 /GS- /Ob1

```assembly
$SG52650 DB '%d', 0aH, 00H
$SG52651 DB '%d', 0aH, 00H
_this$ = -4 ; size = 4
__Pos$ = 8 ; size = 4
?at?$vector@HV?$allocate@H@std@@@std@@QAEAAHI@Z PROC ;
    std::vector<int,std::allocator<int> >::at, COMDAT
    _this$ = ecx
    push ebp
    mov ebp, esp
    push ecx
    mov DWORD PTR _this$[ebp], ecx
    mov eax, DWORD PTR _this$[ebp]
    mov ecx, DWORD PTR _this$[ebp]
    mov edx, DWORD PTR [eax+4]
    sub edx, DWORD PTR [ecx]
    sar edx, 2
    cmp edx, DWORD PTR __Pos$[ebp]
    ja SHORT $LN1@at
    push OFFSET ?? C@_0BM@NMJKDPP@invalid?5vector?$DMT?$DO?5subscript?$AA@
    call DWORD PTR __imp_?_Xout_of_range@std@@YAXPBD@Z
$LN1@at:
    mov eax, DWORD PTR _this$[ebp]
    mov ecx, DWORD PTR [eax]
    mov edx, DWORD PTR __Pos$[ebp]
    lea eax, DWORD PTR [ecx+edx*4]
$LN3@at:
    mov esp, ebp
    pop ebp
    ret 4
?at?$vector@HV?$allocate@H@std@@@std@@QAEAAHI@Z ENDP ; std::vector<int,std::allocator<int> >::at
_c$ = -36 ; size = 12
$T1 = -24 ; size = 4
$T2 = -20 ; size = 4
$T3 = -16 ; size = 4
$T4 = -12 ; size = 4
$T5 = -8 ; size = 4
$T6 = -4 ; size = 4
_main PROC
    push ebp
    mov ebp, esp
    sub esp, 36
    mov DWORD PTR __c$[ebp], 0 ; Myfirst
    mov DWORD PTR __c$[ebp+4], 0 ; Mylast
    mov DWORD PTR __c$[ebp+8], 0 ; Myend
    lea eax, DWORD PTR __c$[ebp]
    push eax
    call ??dump@@YAXPAUvector_of_ints@@@Z ; dump
    add esp, 4
    mov DWORD PTR $T6[ebp], 1
```

577
We see how the .at() method checks the bounds and throws an exception in case of error. The number that the last printf() call prints is just taken from the memory, without any checks.

One may ask, why not use the variables like "size" and "capacity", like it was done in std::string. Supposedly, this was done for faster bounds checking.

The code GCC generates is in general almost the same, but the .at() method is inlined:

Listing 3.117: GCC 4.8.1 -fno-inline-small-functions -O1
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 4
lea eax, [esp+10h]
mov [esp+4], eax
lea eax, [esp+14h]
mov [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi;
std::vector<int,std::allocator<int>>::push_back(int const&)
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov ebx, [esp+14h]
sub eax, ebx
cmp eax, 17h
ja short loc_80001CF
mov edi, [esp+10h]
sub edi, ebx
sar edi, 2
mov dword ptr [esp], 18h
call _Znwj ; operator new(uint)
mov esi, eax
test edi, edi
jz short loc_80001AD
lea eax, ds:0[edi*4]
mov [esp+8], eax ; n
mov [esp+4], ebx ; src
mov [esp], esi ; dest
call memmove

loc_80001AD: ; CODE XREF: main+F8
mov eax, [esp+14h]
test eax, eax
jz short loc_80001BD
mov [esp], eax ; void *
call _ZdlPv ; operator delete(void *)

loc_80001BD: ; CODE XREF: main+117
mov [esp+14h], esi
lea eax, [esi+edi*4]
mov [esp+10h], eax
add esi, 10h
mov [esp+1Ch], esi

loc_80001CF: ; CODE XREF: main+DD
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov dword ptr [esp+10h], 5
lea eax, [esp+10h]
mov [esp+4], eax
lea eax, [esp+14h]
mov [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi;
std::vector<int,std::allocator<int>>::push_back(int const&)
lea eax, [esp+14h]
mov [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
.reserve() is inlined as well. It calls new() if the buffer is too small for the new size, calls memmove() to copy the contents of the buffer, and calls delete() to free the old buffer.

Let's also see what the compiled program outputs if compiled with GCC:

<table>
<thead>
<tr>
<th>Myfirst</th>
<th>_Mylast</th>
<th>_Myend</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x(nil)</td>
<td>0x(nil)</td>
<td>0x(nil)</td>
</tr>
<tr>
<td>size: 0, capacity: 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myfirst=0x8257008, _Mylast=0x825700c, _Myend=0x825700c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size: 1, capacity: 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>element 0: 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myfirst=0x8257020, _Mylast=0x8257020, _Myend=0x8257020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size: 2, capacity: 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>element 0: 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>element 1: 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myfirst=0x8257038, _Mylast=0x8257038, _Myend=0x8257038</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size: 3, capacity: 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>element 0: 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can spot that the buffer size grows in a different way that in MSVC.

Simple experimentation shows that in MSVC’s implementation the buffer grows by ~50% each time it needs to be enlarged, while GCC’s code enlarges it by 100% each time, i.e., doubles it.

**std::map and std::set**

The binary tree is another fundamental data structure.

As its name states, this is a tree where each node has at most 2 links to other nodes. Each node has key and/or value: std::set provides only key at each node, std::map provides both key and value at each node.

Binary trees are usually the structure used in the implementation of “dictionaries” of key-values (AKA “associative arrays”).

There are at least three important properties that a binary trees has:

- All keys are always stored in sorted form.
- Keys of any types can be stored easily. Binary tree algorithms are unaware of the key’s type, only a key comparison function is required.
- Finding a specific key is relatively fast in comparison with lists and arrays.

Here is a very simple example: let’s store these numbers in a binary tree: 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010.
All keys that are smaller than the node key’s value are stored on the left side.

All keys that are bigger than the node key’s value are stored on the right side.

Hence, the lookup algorithm is straightforward: if the value that you are looking for is smaller than the current node’s key value: move left, if it is bigger: move right, stop if the value required is equal to the node key’s value.

That is why the searching algorithm may search for numbers, text strings, etc., as long as a key comparison function is provided.

All keys have unique values.

Having that, one needs $\approx \log_2 n$ steps in order to find a key in a balanced binary tree with $n$ keys. This implies that $\approx 10$ steps are needed $\approx 1000$ keys, or $\approx 13$ steps for $\approx 10000$ keys.

Not bad, but the tree has always to be balanced for this: i.e., the keys has to be distributed evenly on all levels. The insertion and removal operations do some maintenance to keep the tree in a balanced state.

There are several popular balancing algorithms available, including the AVL tree and the red-black tree.

The latter extends each node with a “color” value to simplify the balancing process, hence, each node may be “red” or “black”.

Both GCC’s and MSVC’s `std::map` and `std::set` template implementations use red-black trees. `std::set` has only keys. `std::map` is the “extended” version of `std::set`: it also has a value at each node.

**MSVC**

```cpp
#include <map>
#include <set>
#include <string>
#include <iostream>

// Structure is not packed! Each field occupies 4 bytes.
struct tree_node
{
    struct tree_node *Left;
    struct tree_node *Parent;
    struct tree_node *Right;
    char Color; // 0 - Red, 1 - Black
    char Isnil;
    //std::pair Myval;
    unsigned int first; // called Myval in std::set
    const char *second; // not present in std::set
};

struct tree_struct
{
    struct tree_node *Myhead;
    size_t Mysize;
};
```
void dump_tree_node (struct tree_node *n, bool is_set, bool traverse)
{
    printf ("ptr=0x%p Left=0x%p Parent=0x%p Right=0x%p Color=%d Isnil=%d\n", n, n->Left, n->Parent, n->Right, n->Color, n->Isnil);
    if (n->Isnil==0)
    {
        if (is_set)
            printf ("first=%d\n", n->first);
        else
            printf ("first=%d second=[%s]\n", n->first, n->second);
    }
    if (traverse)
    {
        if (n->Isnil==1)
            dump_tree_node (n->Parent, is_set, true);
        else
        {
            if (n->Left->Isnil==0)
                dump_tree_node (n->Left, is_set, true);
            if (n->Right->Isnil==0)
                dump_tree_node (n->Right, is_set, true);
        };
    };
}

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    if (is_set)
        printf ("%d\n", n->first);
    else
        printf ("%d [%s]\n", n->first, n->second);
    if (n->Left->Isnil==0)
    {
        printf (".%sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Left, is_set);
    };
    if (n->Right->Isnil==0)
    {
        printf (".%sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Right, is_set);
    };
}

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, Myhead=0x%p, Mysize=%d\n", m, m->Myhead, m->Mysize);
    dump_tree_node (m->Myhead, is_set, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->Myhead->Parent, is_set);
};

int main()
{
    // map
    std::map<int, const char*> m;
    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
}
m[1]="one";
m[6]="six";
m[99]="ninety-nine";
m[5]="five";
m[11]="eleven";
m[1001]="one thousand one";
m[1010]="one thousand ten";
m[2]="two";
m[9]="nine";
printf("dumping m as map:\n");
dump_map_and_set(((struct tree_struct*)(void**)&m, false);

std::map<int, const char*>*::iterator it1=m.begin();
printf("m.begin():\n");
dump_tree_node(((struct tree_node*)(void**)&it1, false, false);

it1=m.end();
printf("m.end():\n");
dump_tree_node(((struct tree_node*)(void**)&it1, false, false);

// set
std::set<int> s;
s.insert(123);
s.insert(456);
s.insert(11);
s.insert(12);
s.insert(100);
s.insert(1001);
printf("dumping s as set:\n");
dump_map_and_set(((struct tree_struct*)(void**)&s, true);
std::set<int>::iterator it2=s.begin();
printf("s.begin():\n");
dump_tree_node(((struct tree_node*)(void**)&it2, true, false);

it2=s.end();
printf("s.end():\n");
dump_tree_node(((struct tree_node*)(void**)&it2, true, false);

Listing 3.118: MSVC 2012

dumping m as map:
ptr=0x0020FE04, Myhead=0x005BB3A0, Mysize=17
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0x005BB580 Color=1 Isnil=1
ptr=0x005BB3C0 Left=0x005BB4C0 Parent=0x005BB3A0 Right=0x005BB440 Color=1 Isnil=0
first=10 second=[ten]
ptr=0x005BB440 Left=0x005BB3E0 Parent=0x005BB440 Right=0x005BB560 Color=0 Isnil=0
first=20 second=[twenty]
ptr=0x005BB560 Left=0x005BB460 Parent=0x005BB440 Right=0x005BB500 Color=0 Isnil=0
first=30 second=[thirty]
ptr=0x005BB570 Left=0x005BB470 Parent=0x005BB460 Right=0x005BB540 Color=1 Isnil=0
first=40 second=[forty]
ptr=0x005BB580 Left=0x005BB480 Parent=0x005BB470 Right=0x005BB5C0 Color=0 Isnil=0
first=50 second=[fifty]
ptr=0x005BB590 Left=0x005BB490 Parent=0x005BB480 Right=0x005BB600 Color=1 Isnil=0
first=60 second=[sixty]
ptr=0x005BB600 Left=0x005BB5A0 Parent=0x005BB590 Right=0x005BB660 Color=0 Isnil=0
first=70 second=[seventy]
ptr=0x005BB610 Left=0x005BB5B0 Parent=0x005BB600 Right=0x005BB6A0 Color=1 Isnil=0
first=80 second=[eighty]
ptr=0x005BB620 Left=0x005BB5C0 Parent=0x005BB610 Right=0x005BB6E0 Color=0 Isnil=0
first=90 second=[ninety]
ptr=0x005BB630 Left=0x005BB5D0 Parent=0x005BB620 Right=0x005BB720 Color=1 Isnil=0
first=100 second=[one hundred]
ptr=0x005BB720 Left=0x005BB630 Parent=0x005BB620 Right=0x005BB760 Color=0 Isnil=0
first=110 second=[one hundred ten]
ptr=0x005BB730 Left=0x005BB640 Parent=0x005BB630 Right=0x005BB780 Color=1 Isnil=0
first=120 second=[one hundred twenty]
ptr=0x005BB740 Left=0x005BB650 Parent=0x005BB640 Right=0x005BB7A0 Color=0 Isnil=0
first=130 second=[one hundred thirty]
ptr=0x005BB750 Left=0x005BB660 Parent=0x005BB650 Right=0x005BB7C0 Color=1 Isnil=0
first=140 second=[one hundred forty]
ptr=0x005BB760 Left=0x005BB670 Parent=0x005BB660 Right=0x005BB7E0 Color=0 Isnil=0
first=150 second=[one hundred fifty]
ptr=0x005BB770 Left=0x005BB680 Parent=0x005BB670 Right=0x005BB800 Color=1 Isnil=0
first=160 second=[one hundred sixty]
ptr=0x005BB780 Left=0x005BB690 Parent=0x005BB680 Right=0x005BB820 Color=0 Isnil=0
first=170 second=[one hundred seventy]
ptr=0x005BB790 Left=0x005BB6A0 Parent=0x005BB690 Right=0x005BB840 Color=1 Isnil=0
first=180 second=[one hundred eighty]
ptr=0x005BB7A0 Left=0x005BB6B0 Parent=0x005BB6A0 Right=0x005BB860 Color=0 Isnil=0
first=190 second=[one hundred ninety]
ptr=0x005BB7B0 Left=0x005BB6C0 Parent=0x005BB6B0 Right=0x005BB880 Color=1 Isnil=0
first=200 second=[two hundred]
The structure is not packed, so both char values occupy 4 bytes each.

As for std::map, first and second can be viewed as a single value of type std::pair. std::set has only one value at this address in the structure instead.

The current size of the tree is always present, as in the case of the implementation of std::list in MSVC (3.21.4 on page 571).

As in the case of std::list, the iterators are just pointers to nodes. The .begin() iterator points to the minimal key.

That pointer is not stored anywhere (as in lists), the minimal key of the tree is looked up every time.
operator-- and operator++ move the current node pointer to the predecessor or successor respectively, i.e., the nodes which have the previous or next key.

The algorithms for all these operations are explained in [Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, *Introduction to Algorithms, Third Edition*, (2009)].

The .end() iterator points to the dummy node, it has 1 in `Isnil`, which implies that the node has no key and/or value. It can be viewed as a "landing zone" in HDD\(^{33}\) and often called sentinel [see N. Wirth, *Algorithms and Data Structures*, 1985]\(^{34}\).

The “parent” field of the dummy node points to the root node, which serves as a vertex of the tree and contains information.

 GCC

```c
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
  int key;
  const char *value;
};

struct tree_node
{
  int M_color; // 0 - Red, 1 - Black
  struct tree_node *M_parent;
  struct tree_node *M_left;
  struct tree_node *M_right;
};

struct tree_struct
{
  int M_key_compare;
  struct tree_node M_header;
  size_t M_node_count;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool traverse, bool dump_keys_and_values)
{
  printf("ptr=%p M_left=%p M_parent=%p M_right=%p M_color=%d\n",
          n, n->M_left, n->M_parent, n->M_right, n->M_color);

  void *point_after_struct=((char*)n)+sizeof(struct tree_node);

  if (dump_keys_and_values)
  {
    if (is_set)
      printf("key=%d\n", *(int*)point_after_struct);
    else
      {
        struct map_pair *p=(struct map_pair *)point_after_struct;
        printf("key=%d value=[%s]\n", p->key, p->value);
      }
  }

  if (traverse==false)
    return;

  if (n->M_left)
    dump_tree_node (n->M_left, is_set, traverse, dump_keys_and_values);
  if (n->M_right)
```
dump_tree_node (n->M_right, is_set, traverse, dump_keys_and_values);
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t\t
Listing 3.119: GCC 4.8.1

dumping m as map:
ptr=0x0028FE3C  M_key_compare=0x402b70  M_header=0x0028FE40  M_node_count=17
ptr=0x007A4988 M_left=0x007A4C00 M_parent=0x0028FE40 M_right=0x007A4B80 M_color=1
key=10 value=[ten]  key=1 value=[one]  key=0 value=[zero]  key=5 value=[five]  key=3 value=[three]  key=2 value=[two]
key=6 value=[six]  key=9 value=[nine]  key=100 value=[one hundred]

As a tree:
root----10 [ten]
   L-------0 [zero]
       R------5 [five]
   L-------3 [three]
       R------2 [two]
       R------6 [six]
       R------9 [nine]
   R-------100 [one hundred]
GCC’s implementation is very similar\(^\text{35}\). The only difference is the absence of the Isnil field, so the structure occupies slightly less space in memory than its implementation in MSVC.

The dummy node is also used as a place to point the `.end()` iterator also has no key and/or value.

**Rebalancing demo (GCC)**

Here is also a demo showing us how a tree is rebalanced after some insertions.

**Listing 3.120: GCC**

```c
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair {
    int key;
    const char *value;
};

struct tree_node {
    int M_color; // 0 - Red, 1 - Black
```
struct tree_node *M_parent;
struct tree_node *M_left;
struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

const char* ALOT_OF_TABS="								"
;

void dump_as_tree (int tabs, struct tree_node *n)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node);
    printf ("%d
", *(int*)point_after_struct);
    if (n->M_left)
    {
        printf (".%sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left);
    }
    if (n->M_right)
    {
        printf (".%sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right);
    }
};

void dump_map_and_set(struct tree_struct *m)
{
    printf ("root-----");
    dump_as_tree (1, m->M_header.M_parent);
};

int main()
{
    std::set<int> s;
    s.insert(123);
    s.insert(456);
    printf ("123, 456 has been inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(11);
    s.insert(12);
    printf ("\n");
    printf ("11, 12 has been inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(100);
    s.insert(1001);
    printf ("\n");
    printf ("100, 1001 has been inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(667);
    s.insert(1);
    s.insert(4);
    s.insert(7);
    printf ("\n");
    printf ("667, 1, 4, 7 has been inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    printf ("\n");
};

Listing 3.121: GCC 4.8.1

123, 456 has been inserted
root----123
    R-------456

591
3.21.5 Memory

Sometimes you may hear from C++ programmers “allocate memory on stack” and/or “allocate memory on heap”.

Allocating object on stack:

```cpp
void f()
{
    ...
    Class o=Class(...);
    ...
}
```

The memory for object (or structure) is allocated in stack, using simple SP shift. The memory is deallocated upon function exit, or, more precisely, at the end of scope—SP is returning to its state (same as at the start of function) and destructor of Class is called. In the same manner, memory for allocated structure in C is deallocated upon function exit.

Allocating object on heap:

```cpp
void f1()
{
    ...
    Class *o=new Class(...);
    ...
}
void f2()
{
    ...
    delete o;
    ...
}
```
This is the same as allocating memory for a structure using `malloc()` call. In fact, `new` in C++ is wrapper for `malloc()`, and `delete` is wrapper for `free()`. Since memory block has been allocated in heap, it must be deallocated explicitly, using `delete`. Class destructor will be automatically called right before that moment.

Which method is better? Allocating on stack is very fast, and good for small, short-lived object, which will be used only in the current function.

Allocating on heap is slower, and better for long-lived object, which will be used across many functions. Also, objects allocated in heap are prone to memory leakage, because they must to be freed explicitly, but one can forget about it.

Anyway, this is matter of taste.

### 3.22 Negative array indices

It’s possible to address the space before an array by supplying a negative index, e.g., `array[-1]`.

#### 3.22.1 Addressing string from the end

Python PL allows to address arrays and strings from the end. For example, `string[-1]` returns the last character, `string[-2]` returns penultimate, etc. Hard to believe, but this is also possible in C/C++:

```c
#include <string.h>
#include <stdio.h>

int main()
{
    char *s="Hello, world!";
    char *s_end=s+strlen(s);
    printf ("last character: %c\n", s_end[-1]);
    printf ("penultimate character: %c\n", s_end[-2]);
}
```

It works, but `s_end` must always has an address of terminating zero byte at the end of `s` string. If `s` string’s size get changed, `s_end` must be updated.

The trick is dubious, but again, this is a demonstration of negative indices.

#### 3.22.2 Addressing some kind of block from the end

Let’s first recall why stack grows backwards (1.9.1 on page 30). There is some kind of block in memory and you want to store both heap and stack there, and you are not sure, how big they both can grow during runtime.

You can set a heap pointer to the beginning of the block, then you can set a stack pointer to the end of the block (`heap + size_of_block`), and then you can address nth element of stack like `stack[-n]`. For example, `stack[-1]` for 1st element, `stack[-2]` for 2nd, etc.

This will work in the same fashion, as our trick of addressing string from the end.

You can easily check if the structures has not begun to overlap each other: just be sure that address of the last element in heap is below the address of the last element of stack.

Unfortunately, `-0` as index will not work, since two’s complement way of representing negative numbers (2.2 on page 448) don’t allow negative zero, so it cannot be distinguished from positive zero.

This method is also mentioned in “Transaction processing”, Jim Gray, 1993, “The Tuple-Oriented File System” chapter, p. 755.

#### 3.22.3 Arrays started at 1

Fortran and Mathematica defined first element of array as 1th, probably because this is tradition in mathematics. Other PLs like C/C++ defined it as 0th. Which is better? Edsger W. Dijkstra argued that latter is better.

36 See https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html
But programmers may still have a habit after Fortran, so using this little trick, it's possible to address the first element in C/C++ using index 1:

```c
#include <stdio.h>

int main()
{
    int random_value=0x11223344;
    unsigned char array[10];
    int i;
    unsigned char *fakearray=&array[-1];

    for (i=0; i<10; i++)
        array[i]=i;

    printf ("first element %d\n", fakearray[1]);
    printf ("second element %d\n", fakearray[2]);
    printf ("last element %d\n", fakearray[10]);

    printf ("array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X\n", array[-1], array[-2], array[-3], array[-4]);
}
```

Listing 3.122: Non-optimizing MSVC 2010
; eax=address of fakearray[0], eax+2 is fakearray[2] or array[1]
movzx ecx, BYTE PTR [eax+2]
push ecx
push OFFSET $SG2752 ; 'second element %d'
call _printf
add esp, 8
mov edx, DWORD PTR fakearray$[ebp]
; edx=address of fakearray[10], edx+10 is fakearray[10] or array[9]
movzx eax, BYTE PTR [edx+10]
push eax
push OFFSET $SG2753 ; 'last element %d'
call _printf
add esp, 8

; subtract 4, 3, 2 and 1 from pointer to array[0] in order to find values before array[]
lea ecx, DWORD PTR _array$[ebp]
movzx edx, BYTE PTR [ecx-4]
push edx
lea eax, DWORD PTR _array$[ebp]
movzx ecx, BYTE PTR [eax-3]
push ecx
lea edx, DWORD PTR _array$[ebp]
movzx eax, BYTE PTR [edx-2]
push eax
lea ecx, DWORD PTR _array$[ebp]
movzx edx, BYTE PTR [ecx-1]
push edx
push OFFSET $SG2754 :
array[-1]=%02X, array[-2]=%02X, array[-3]=%02X, array[-4]=%02X'
call _printf
add esp, 20
xor eax, eax
mov esp, ebp
pop ebp
ret 0

_main ENDP

So we have array[] of ten elements, filled with 0…9 bytes.

Then we have the fakearray[] pointer, which points one byte before array[].

fakearray[1] points exactly to array[0]. But we are still curious, what is there before array[]? We have added random_value before array[] and set it to 0x11223344. The non-optimizing compiler allocated the variables in the order they were declared, so yes, the 32-bit random_value is right before the array.

We ran it, and:

first element 0
second element 1
last element 9

Here is the stack fragment we will copypaste from OllyDbg’s stack window (with comments added by the author):

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>001DFBCC</td>
<td>/001DFBD3 ; fakearray pointer</td>
</tr>
<tr>
<td>001DFBD0</td>
<td>11223344 ; random_value</td>
</tr>
<tr>
<td>001DFBD4</td>
<td>03020100 ; 4 bytes of array[]</td>
</tr>
<tr>
<td>001DFBD8</td>
<td>07060504 ; 4 bytes of array[]</td>
</tr>
<tr>
<td>001DFBDC</td>
<td>00CB0908 ; random garbage + 2 last bytes of array[]</td>
</tr>
<tr>
<td>001DFBE0</td>
<td>0000000A ; last i value after loop was finished</td>
</tr>
<tr>
<td>001DFBE4</td>
<td>001DFC2C ; saved EBP value</td>
</tr>
<tr>
<td>001DFBE8</td>
<td>00CB129D ; Return Address</td>
</tr>
</tbody>
</table>

The pointer to the fakearray[] (0x001DFBD3) is indeed the address of array[] in the stack (0x001DFBD4), but minus 1 byte.

It’s still very hackish and dubious trick. Doubtfully anyone should use it in production code, but as a demonstration, it fits perfectly here.
3.23 More about pointers

The way C handles pointers, for example, was a brilliant innovation; it solved a lot of problems that we had before in data structuring and made the programs look good afterwards.

Donald Knuth, interview (1993)

For those, who still have hard time understanding C/C++ pointers, here are more examples. Some of them are weird and serves only demonstration purpose: use them in production code only if you really know what you’re doing.

3.23.1 Working with addresses instead of pointers

Pointer is just an address in memory. But why we write `char* string` instead of something like `address string`? Pointer variable is supplied with a type of the value to which pointer points. So then compiler will be able to catch data typization bugs during compilation.

To be pedantic, data typing in programming languages is all about preventing bugs and self-documentation. It’s possible to use maybe two of data types like `int` (or `int64_t`) and byte—these are the only types which are available to assembly language programmers. But it’s just very hard task to write big and practical assembly programs without nasty bugs. Any small typo can lead to hard-to-find bug.

Data type information is absent in a compiled code (and this is one of the main problems for decompilers), and I can demonstrate this.

This is what sane C/C++ programmer can write:

```c
#include <stdio.h>
#include <stdint.h>

void print_string (char *s)
{
    printf ("(address: 0x%llx)\n", s);
    printf ("%s\n", s);
};

int main()
{
    char *s="Hello, world!";
    print_string (s);
};
```

This is what I can write:

```c
#include <stdio.h>
#include <stdint.h>

void print_string (uint64_t address)
{
    printf ("(address: 0x%llx)\n", address);
    puts ((char*)address);
};

int main()
{
    char *s="Hello, world!";
    print_string ((uint64_t)s);
};
```

I use `uint64_t` because I run this example on Linux x64. `int` would work for 32-bit OS-es. First, a pointer to character (the very first in the greeting string) is casted to `uint64_t`, then it’s passed further. `print_string()` function casts back incoming `uint64_t` value into pointer to a character.

What is interesting is that GCC 4.8.4 produces identical assembly output for both versions:
(I've removed all insignificant GCC directives.)

I also tried UNIX diff utility and it shows no differences at all.

Let's continue to abuse C/C++ programming traditions heavily. Someone may write this:

```c
#include <stdio.h>
#include <stdint.h>

type8_t load_byte_at_address (type8_t* address)
{
    return *address;
    //this is also possible: return address[0];
};

void print_string (char *s)
{
    char* current_address=s;
    while (1)
    {
        char current_char=load_byte_at_address(current_address);
        if (current_char==0)
            break;
        printf ("%c", current_char);
        current_address++;
    }
};

int main()
{
    char *s="Hello, world!";
    print_string (s);
};
```

It can be rewritten like this:

```c
#include <stdio.h>
#include <stdint.h>

type8_t load_byte_at_address (type64_t address)
{
    return *(type8_t*)address;
};

void print_string (type64_t address)
```
Both source codes resulting in the same assembly output:

```c
#include <stdio.h>
#include <stdint.h>

uint64_t multiply1(uint64_t a, uint64_t b) {
    return a * b;
}
```

(I have also removed all insignificant GCC directives.)

No difference: C/C++ pointers are essentially addresses, but supplied with type information, in order to prevent possible mistakes at the time of compilation. Types are not checked during runtime—it would be huge (and unneeded) overhead.

### 3.23.2 Passing values as pointers; tagged unions

Here is an example on how to pass values in pointers:

```c
#include <stdio.h>
#include <stdint.h>

uint64_t multiply1 (uint64_t a, uint64_t b) {
    return a*b;
}
```
uint64_t* multiply2 (uint64_t *a, uint64_t *b)
{
    return (uint64_t*)((uint64_t)a*(uint64_t)b);
};

int main()
{
    printf ("%d\n", multiply1(123, 456));
    printf ("%d\n", (uint64_t)multiply2((uint64_t*)123, (uint64_t*)456));
};

It works smoothly and GCC 4.8.4 compiles both multiply1() and multiply2() functions identically!

multiply1:
    mov    rax, rdi
    imul   rax, rsi
    ret

multiply2:
    mov    rax, rdi
    imul   rax, rsi
    ret

As long as you do not dereference pointer (in other words, you don’t read any data from the address stored in pointer), everything will work fine. Pointer is a variable which can store anything, like usual variable.

Signed multiplication instruction (IMUL) is used here instead of unsigned one (MUL), read more about it here: 2.2.1 on page 450.

By the way, it’s well-known hack to abuse pointers a little called tagged pointers. In short, if all your pointers points to blocks of memory with size of, let’s say, 16 bytes (or it is always aligned on 16-byte boundary), 4 lowest bits of pointer is always zero bits and this space can be used somehow. It’s very popular in JISP compilers and interpreters. They store cell/object type in these unused bits, this can save some memory. Even more, you can judge about cell/object type using just pointer, with no additional memory access. Read more about it: [Dennis Yurichev, C/C++ programming language notes1.3].

3.23.3 Pointers abuse in Windows kernel

The resource section of PE executable file in Windows OS is a section containing pictures, icons, strings, etc. Early Windows versions allowed to address resources only by IDs, but then Microsoft added a way to address them using strings.

So then it would be possible to pass ID or string to FindResource() function. Which is declared like this:

HRSRC WINAPI FindResource(
    _In_opt_ HMODULE hModule,
    _In_ LPCTSTR lpName,
    _In_ LPCTSTR lpType
);

lpName and lpType has char* or wchar* types, and when someone still wants to pass ID, he/she have to use MAKEINTRESOURCE macro, like this:

result = FindResource(..., MAKEINTRESOURCE(1234), ...);

It’s interesting fact that MAKEINTRESOURCE is merely casting integer to pointer. In MSVC 2013, in the file Microsoft SDKs\Windows\v7.1A\Include\Ks.h we can find this:

...  
#if (!defined( MAKEINTRESOURCE ))
#define MAKEINTRESOURCE( res ) ((ULONG_PTR) (USHORT) res)
#endif
...

Sounds insane. Let’s peek into ancient leaked Windows NT4 source code. In private/windows/base/client/module.c we can find FindResource() source code:
HRSRC
FindResourceA(
    HMODULE hModule,
    LPCSTR lpName,
    LPCSTR lpType
)
...

{ NTSTATUS Status;
  ULONG IdPath[ 3 ];
  PVOID p;

  IdPath[ 0 ] = 0;
  IdPath[ 1 ] = 0;
  try {
    if ((IdPath[ 0 ] = BaseDllMapResourceIdA( lpType )) == -1) {
      Status = STATUS_INVALID_PARAMETER;
    }
    else
      if ((IdPath[ 1 ] = BaseDllMapResourceIdA( lpName )) == -1) {
        Status = STATUS_INVALID_PARAMETER;
      }
  }

  let's proceed to BaseDllMapResourceIdA() in the same source file:

ULONG BaseDllMapResourceIdA(
    LPCSTR lpId
)
{
  NTSTATUS Status;
  ULONG Id;
  UNICODE_STRING UnicodeString;
  ANSI_STRING AnsiString;
  PWSTR s;

  try {
    if ((ULONG)lpId & LDR_RESOURCE_ID_NAME_MASK) {
      if (*lpId == '#') {
        Status = RtlCharToInteger( lpId+1, 10, &Id );
        if (!NT_SUCCESS( Status ) || Id & LDR_RESOURCE_ID_NAME_MASK) {
          Status = STATUS_INVALID_PARAMETER;
        }
        BaseSetLastError( Status );
        Id = (ULONG)-1;
      }
    }
    else {
      RtlInitAnsiString( &AnsiString, lpId );
      Status = RtlAnsiStringToUnicodeString( &UnicodeString,
                                               &AnsiString,
                                               TRUE
                                            );
      if (!NT_SUCCESS( Status )){
        BaseSetLastError( Status );
        Id = (ULONG)-1;
      }
      else {
        s = UnicodeString.Buffer;
        while (*s != UNICODE_NULL) {
          *s = RtlUpcaseUnicodeChar( *s );
          s++;
        }
        Id = (ULONG)UnicodeString.Buffer;
      }
    }
  }
}
lpId is ANDed with LDR_RESOURCE_ID_NAME_MASK.
Which we can find in public/sdk/inc/ntldr.h:

```c
#define LDR_RESOURCE_ID_NAME_MASK 0xFFFF0000
```

So lpId is ANDed with 0xFFFF0000 and if some bits beyond lowest 16 bits are still present, first half of function is executed (lpId is treated as an address of string). Otherwise—second half (lpId is treated as 16-bit value).

Still, this code can be found in Windows 7 kernel32.dll file:
If value in input pointer is greater than 0x10000, jump to string processing is occurred. Otherwise, input value of lpId is returned as is. 0xFFFF0000 mask is not used here any more, because this is 64-bit code after all, but still, 0xFFFFFFFFFFFF0000 could work here.

Attentive reader may ask, what if address of input string is lower than 0x10000? This code relied on the fact that in Windows there are nothing on addresses below 0x10000, at least in Win32 realm.

Raymond Chen writes about this:

How does MAKE INT RESOURCE work? It just stashes the integer in the bottom 16 bits of a pointer, leaving the upper bits zero. This relies on the convention that the first 64KB of address space is never mapped to valid memory, a convention that is enforced starting in Windows 7.

In short words, this is dirty hack and probably one should use it only if there is a real necessity. Perhaps, FindResource() function in past had SHORT type for its arguments, and then Microsoft has added a way to pass strings there, but older code must also be supported.

Now here is my short distilled example:

```c
#include <stdio.h>
#include <stdint.h>

void f(char* a)
{
    if (((uint64_t)a)>0x10000)
        printf("Pointer to string has been passed: %s\n", a);
    else
        printf("16-bit value has been passed: %d\n", (uint64_t)a);
}

int main()
{
    f("Hello!"); // pass string
    f((char*)1234); // pass 16-bit value
}
```

It works!

**Pointers abuse in Linux kernel**

As it has been noted in comments on Hacker News, Linux kernel also has something like that.

For example, this function can return both error code and pointer:

```c
struct kernfs_node *kernfs_create_link(struct kernfs_node *parent,
    const char *name,
    struct kernfs_node *target)
{
    struct kernfs_node *kn;
    int error;
```
kn = kernfs_new_node(parent, name, S_IFLNK|S_IRWXUGO, KERNFS_LINK);
if (!kn)
    return ERR_PTR(-ENOMEM);
if (kernfs_ns_enabled(parent))
    kn->ns = target->ns;
kn->symlink.target_kn = target;
kernfs_get(target); /* ref owned by symlink */
error = kernfs_add_one(kn);
if (!error)
    return kn;
kernfs_put(kn);
return ERR_PTR(error);

ERR_PTR is a macro to cast integer to pointer:

static inline void *__must_check ERR_PTR(long error)
{
    return (void *)error;
}

This header file also has a macro helper to distinguish error code from pointer:

#define IS_ERR_VALUE(x) unlikely((x) >= (unsigned long)-MAX_ERRNO)

This means, error codes are the “pointers” which are very close to -1 and, hopefully, there are nothing in kernel memory on the addresses like 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFE, 0xFFFFFFFFFFFFFFFD, etc.

Much more popular solution is to return NULL in case of error and to pass error code via additional argument. Linux kernel authors don’t do that, but everyone who use these functions must always keep in mind that returning pointer must always be checked with IS_ERR_VALUE before dereferencing.

For example:

    fman->cam_offset = fman_muram_alloc(fman->muram, fman->cam_size);
    if (IS_ERR_VALUE(fman->cam_offset)) {
        dev_err(fman->dev, "%s: MURAM alloc for DMA CAM failed\n", __func__);
        return -ENOMEM;
    }

Pointers abuse in UNIX userland

mmap() function returns -1 in case of error (or MAP_FAILED, which equals to -1). Some people say, mmap() can map a memory at zeroth address in rare situations, so it can’t use 0 or NULL as error code.

3.23.4 Null pointers

“Null pointer assignment” error of MS-DOS era

Oldschool readers may recall a weird error message of MS-DOS era: “Null pointer assignment”. What does it mean?

It’s not possible to write a memory at zero address in *NIX and Windows OSes, but it was possible to do so in MS-DOS due to absence of memory protection whatsoever.
So I've pulled my ancient Turbo C++ 3.0 (later it was renamed to Borland C++) from early 1990s and tried to compile this:

```c
#include <stdio.h>

int main()
{
    int *ptr=NULL;
    *ptr=1234;
    printf("Now let's read at NULL\n");
    printf("%d\n", *ptr);
}
```

Hard to believe, but it works, with error upon exit, though:

```
Listing 3.124: Ancient Turbo C 3.0
C:\TC30\BIN\1
Now let's read at NULL
1234
Null pointer assignment
C:\TC30\BIN>
```

Let's dig deeper into the source code of CRT of Borland C++ 3.1, file c0.asm:

```asm
; _checknull() check for null pointer zapping copyright message
...

; Check for null pointers before exit
_checknull PROC DIST
PUBLIC _checknull

IF LDATA EQ false
IFNDEF __TINY__
    push si
    push di
    mov es, cs:DGROUP@:
    xor ax, ax
    mov si, ax
    mov cx, lgth_CopyRight
ComputeChecksum label near
    add al, es:[si]
    adc ah, 0
    inc si
loop ComputeChecksum
    sub ax, CheckSum
    jz @@SumOK
    mov cx, lgth_NullCheck
    mov dx, offset DGROUP: NullCheck
    call ErrorDisplay
@@SumOK:
    pop di
    pop si
ENDIF
ENDIF

_DATA SEGMENT
; Magic symbol used by the debug info to locate the data segment
public DATASEG@

DATASEG@
label byte

; The CopyRight string must NOT be moved or changed without
; changing the null pointer check logic
CopyRight db 4 dup(0)
CopyRight db 'Borland C++ - Copyright 1991 Borland Intl.',0
lgth_CopyRight equ $ - CopyRight
```

604
The MS-DOS memory model was really weird ([11.6 on page 986](#)) and probably not worth looking into it unless you’re fan of retrocomputing or retrogaming. One thing we have to keep in mind is that memory segment (included data segment) in MS-DOS is a memory segment in which code or data is stored, but unlike “serious” OSes, it’s started at address 0.

And in Borland C++ CRT, the data segment is started with 4 zero bytes and the copyright string “Borland C++ - Copyright 1991 Borland Intl.”. The integrity of the 4 zero bytes and text string is checked upon exit, and if it’s corrupted, the error message is displayed.

But why? Writing at null pointer is common mistake in C/C++, and if you do so in *NIX or Windows, your application will crash. MS-DOS has no memory protection, so CRT has to check this post-factum and warn about it upon exit. If you see this message, this means, your program at some point has written at address 0.

Our program did so. And this is why 1234 number has been read correctly: because it was written at the place of the first 4 zero bytes. Checksum is incorrect upon exit (because the number has been left there), so error message has been displayed.

Am I right? I’ve rewritten the program to check my assumptions:

```c
#include <stdio.h>

int main()
{
    int *ptr=NULL;
    *ptr=1234;
    printf("Now let's read at NULL\n");
    printf("%d\n", *ptr);
    *ptr=0; // psst, cover our tracks!
}
```

This program executes without error message upon exit.

Though method to warn about null pointer assignment is relevant for MS-DOS, perhaps, it can still be used today in low-cost MCUs with no memory protection and/or MMU[^37].

### Why would anyone write at address 0?

But why would sane programmer write a code which writes something at address 0? It can be done accidentally: for example, a pointer must be initialized to newly allocated memory block and then passed to some function which returns data through pointer.

```c
int *ptr=NULL;
...
strcpy (ptr, buf); // strcpy() terminates silently because MS-DOS has no memory protection
```

Even worse:

```c
int *ptr=malloc(1000);
...
strcpy (ptr, buf); // strcpy() terminates silently because MS-DOS has no memory protection
```

[^37]: Memory Management Unit
3.4 Generating a Core File on Errors

If the `error-abort' debug token has been enabled, when the library detects any problems with the heap memory, it will immediately attempt to dump a core file. *Note Debug Tokens::. Core files are a complete copy of the program and it's state and can be used by a debugger to see specifically what is going on when the error occurred. *Note Using With a Debugger::. By default, the low, medium, and high arguments to the library utility enable the `error-abort' token. You can disable this feature by entering `dmalloc -m error-abort' (-m for minus) to remove the `error-abort' token and your program will just log errors and continue. You can also use the `error-dump' token which tries to dump core when it sees an error but still continue running. *Note Debug Tokens::.

When a program dumps core, the system writes the program and all of its memory to a file on disk usually named `core'. If your program is called `foo' then your system may dump core as `foo.core'. If you are not getting a `core' file, make sure that your program has not changed to a new directory meaning that it may have written the core file in a different location. Also insure that your program has write privileges over the directory that it is in otherwise it will not be able to dump a core file. Core dumps are often security problems since they contain all program memory so systems often block their being produced. You will want to check your user and system's core dump size ulimit settings.

The library by default uses the `abort' function to dump core which may or may not work depending on your operating system. If the following program does not dump core then this may be the problem. See `KILL_PROCESS' definition in `settings.dist'.

```c
main()
{
    abort();
}
```

If `abort' does work then you may want to try the following setting in `settings.dist'. This code tries to generate a segmentation fault by dereferencing a `NULL' pointer.

```c
#define KILL_PROCESS { int *_int_p = 0L; *_int_p = 1; }
```

### NULL in C/C++

NULL in C/C++ is just a macro which is often defined like this:

```c
#define NULL ((void*)0)
```

( `libio.h` file )

`void*` is a data type reflecting the fact it's the pointer, but to a value of unknown data type (`void`). NULL is usually used to show absence of an object. For example, you have a single-linked list, and each node has a value (or pointer to a value) and next pointer. To show that there are no next node, 0 is stored to next field. (Other solutions are just worse.) Perhaps, you may have some crazy environment where you need to allocate memory blocks at zero address. How would you indicate absence of the next node? Some kind of `magic number'? Maybe -1? Or maybe using additional bit?

In Wikipedia we may find this:

[^38]: [http://dmalloc.com/](http://dmalloc.com/)
In fact, quite contrary to the zero page's original preferential use, some modern operating systems such as FreeBSD, Linux and Microsoft Windows[2] actually make the zero page inaccessible to trap uses of NULL pointers.


**Null pointer to function**

It's possible to call function by its address. For example, I compile this by MSVC 2010 and run it in Windows 7:

```c
#include <windows.h>
#include <stdio.h>

int main()
{
    printf ("0x%lx\n", &MessageBoxA);
}
```

The result is 0x7578feae and doesn't changing after several times I run it, because user32.dll (where MessageBoxA function resides) is always loads at the same address. And also because ASLR\(^39\) is not enabled (result would be different each time in that case).

Let's call `MessageBoxA()` by address:

```c
#include <windows.h>
#include <stdio.h>

typedef int (*msgboxtype)(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCaption, UINT uType);

int main()
{
    msgboxtype msgboxaddr=0x7578feae;
    // force to load DLL into process memory,
    // since our code doesn't use any function from user32.dll,
    // and DLL is not imported
    LoadLibrary ("user32.dll");
    msgboxaddr(NULL, "Hello, world!", "hello", MB_OK);
}
```

Weird, but works in Windows 7 x86.

This is commonly used in shellcodes, because it's hard to call DLL functions by name from there. And ASLR is a countermeasure.

Now what is really weird, some embedded C programmers may be familiar with a code like that:

```c
int reset()
{
    void (*foo)(void) = 0;
    foo();
}
```

Who will want to call a function at address 0? This is portable way to jump at zero address. Many low-cost cheap microcontrollers also have no memory protection or MMU and after reset, they start to execute code at address 0, where some kind of initialization code is stored. So jumping to address 0 is a way to reset itself. One could use inline assembly, but if it's not possible, this portable method can be used.

It even compiles correctly by my GCC 4.8.4 on Linux x64:

```
reset:
    sub    rsp, 8
    xor    eax, eax
    call   rax
```

\(^39\)Address Space Layout Randomization
The fact that stack pointer is shifted is not a problem: initialization code in microcontrollers usually completely ignores registers and RAM state and boots from scratch.

And of course, this code will crash on *NIX or Windows because of memory protection and even in absence of protection, there are no code at address 0.

GCC even has non-standard extension, allowing to jump to a specific address rather than call a function there: [http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html](http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html).

### 3.23.5 Array as function argument

Someone may ask, what is the difference between declaring function argument type as array and as pointer?

As it seems, there are no difference at all:

```c
void write_something1(int a[16])
{
    a[5]=0;
};

void write_something2(int *a)
{
    a[5]=0;
};

int f()
{
    int a[16];
    write_something1(a);
    write_something2(a);
};
```

Optimizing GCC 4.8.4:

```c
write_something1:
    mov    DWORD PTR [rdi+20], 0
    ret

write_something2:
    mov    DWORD PTR [rdi+20], 0
    ret
```

But you may still declare array instead of pointer for self-documenting purposes, if the size of array is always fixed. And maybe, some static analysis tool will be able to warn you about possible buffer overflow. Or is it possible with some tools today?


C99 standard also have static keyword [ISO/IEC 9899:TC3 (C C99 standard), (2007) 6.7.5.3]:

> If the keyword static also appears within the [ and ] of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

### 3.23.6 Pointer to a function

A function name in C/C++ without brackets, like “printf” is a pointer to function of void (*)() type. Let’s try to read function’s contents and patch it:

```c
#include <memory.h>
#include <stdio.h>
```
void print_something()
{
    printf(“we are in %s\n”, __FUNCTION__);
};

int main()
{
    print_something();
    printf(“first 3 bytes: %x %x %x...\n”,
            *(unsigned char*)print_something,
            *((unsigned char*)print_something+1),
            *((unsigned char*)print_something+2));

    *(unsigned char*)print_something=0xC3; // RET’s opcode
    printf(“going to call patched print_something():\n”);
    print_something();
    printf(“it must exit at this point\n”);
};

It tells, that the first 3 bytes of functions are 55 89 e5. Indeed, these are opcodes of PUSH EBP and MOV EBP, ESP instructions (these are x86 opcodes). But then our program crashes, because text section is readonly.

We can recompile our example and make text section writable:

```
gcc --static -g -Wl,--omagic -o example example.c
```

That works!

```
we are in print_something()
first 3 bytes: 55 89 e5...
going to call patched print_something():
it must exit at this point
```

### 3.23.7 Pointer to a function: copy protection

A software cracker can find a function that checks protection and return true or false. He/she then can put XOR EAX,EAX / RETN or MOV EAX, 1 / RETN there.

Can you check integrity of the function? As it turns out, this can be done easily.

According to objdump, the first 3 bytes of check_protection() are 0x55 0x89 0xE5 (given the fact this is non-optimizing GCC):

```
#include <stdlib.h>
#include <stdio.h>

int check_protection()
{
    // do something
    return 0;
    // or return 1;
};

int main()
{
    if (check_protection()==0)
    {
        printf(“no protection installed\n”);
        exit(0);
    );

    // ...and then, at some very important point...
    if (*((unsigned char*)check_protection)+0) != 0x55)
    {
        printf(“1st byte has been altered\n”);
    }
```

If someone would patch the beginning of the check_protection() function, your program can do something mean, maybe exit suddenly. To find such a trick, a cracker can set a memory read breakpoint on the address of the function’s beginning. (tracer has BPMx options for that.)

3.23.8 Pointer to a function: a common bug (or typo)
A notorious bug/typo:

```c
int expired()
{
    // check license key, current date/time, etc
};

int main()
{
    if (expired) // must be expired() here
    {
        print ("expired\n");
        exit(0);
    }
    else
    {
        // do something
    }
};
```

Since the function’s name alone is interpreted as a pointer to function, or address, the if(function_name) statement is like if(true).

Unfortunately, a C/C++ compiler wouldn’t issue a warning.

3.23.9 Pointer as object identificator
Both assembly language and C has no OOP features, but it’s possible to write a code in OOP style (just treat structure as an object).

It’s interesting, that sometimes, pointer to an object (or its address) is called as ID (in sense of data hiding/encapsulation).

For example, LoadLibrary(), according to MSDN\(^{41}\), returns “handle to the module” \(^{42}\). Then you pass this “handle” to other functions like GetProcAddress(). But in fact, LoadLibrary() returns pointer to DLL file

\(^{41}\)Microsoft Developer Network
\(^{42}\)https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms684175(v=vs.85).aspx
mapped into memory. You can read two bytes from the address `LoadLibrary()` returns, and that would be “MZ” (first two bytes of any .EXE/DLL file in Windows).

Apparently, Microsoft “hides” that fact to provide better forward compatibility. Also, `HMODULE` and `HANDLE` data types had another meaning in 16-bit Windows.

Probably, this is reason why `printf()` has “%p” modifier, which is used for printing pointers (32-bit integers on 32-bit architectures, 64-bit on 64-bit, etc) in hexadecimal form. Address of a structure dumped into debug log may help in finding it in another place of log.

Here is also from SQLite source code:

```c

struct Pager {
    sqlite3_vfs *pVfs; /* OS functions to use for I/O */
    u8 exclusiveMode;  /* Boolean. True if locking_mode==EXCLUSIVE */
    u8 journalMode;    /* One of the PAGER_JOURNALMODE_* values */
    u8 useJournal;     /* Use a rollback journal on this file */
    u8 noSync;         /* Do not sync the journal if true */
}

static int pagerLockDb(Pager *pPager, int eLock){
    int rc = SQLITE_OK;
    assert( eLock==SHARED_LOCK || eLock==RESERVED_LOCK || eLock==EXCLUSIVE_LOCK );
    if( pPager->eLock<eLock || pPager->eLock==UNKNOWN_LOCK ){
        rc = sqlite3OsLock(pPager->fd, eLock);
        if( rc==SQLITE_OK && (pPager->eLock!=UNKNOWN_LOCK||eLock==EXCLUSIVE_LOCK) ){
            pPager->eLock = (u8)eLock;
            IOTRACE("LOCK %p %d
", pPager, eLock))
        }
    }
    return rc;
}

PAGER_INCR(sqlite3_pager_readdb_count);
PAGER_INCR(pPager->nRead);
IOTRACE("PGIN %p %d
", pPager, pgno);
PAGERID(pPAGER, pgno, pager_pagehash(pPg));

```

### 3.23.10 Oracle RDBMS and a simple garbage collector for C/C++

There was a time, when the author of these lines tried to learn more about Oracle RDBMS, searching for vulnerabilities, etc. This is a huge piece of software, and a typical function can take very large nested objects as arguments. And I wanted to dump these objects, as trees (or graphs).

Also, I tracked all memory allocations/deallocations by intercepting memory allocating/deallocating functions. And when a function to be intercepted getting a pointer to a block in memory, I search for the block in a list of blocks allocated. I’m getting its size + short name of block (this is like “tagging” in Windows OS kernel).

Given a block, I can scan it for 32-bit words (on 32-bit OS) or for 64-bit words (on 64-bit OS). Each word can be a pointer to another block. And if it is so (I find this another block in my records), I can process it recursively.

And then, using GraphViz, I could render such a diagrams:

---

43[https://blogs.msdn.microsoft.com/oldnewthing/20041025-00/?p=37483](https://blogs.msdn.microsoft.com/oldnewthing/20041025-00/?p=37483)

44Read more about comments in allocated blocks: [Dennis Yurichev, C/C++ programming language notes](http://yurichev.com/C-book.html)
This is quite impressive, given the fact that I had no information about data types of all these structures. But I could get some information from it.

**Now the garbage collector for C/C++: Boehm GC**

If you use a block allocated in memory, its address has to be present somewhere, as a pointer in some structure/array in another allocated block, or in globally allocated structure, or in local variable in stack. If there are no pointer to a block, you can call it "orphan", and it will be a reason of memory leak.

And this is what GC does. It scans all blocks (because it keep tabs on all blocks allocated) for pointers. It’s important to understand, that it has no idea of data types of all these structure fields in blocks—this is important, GC has no information about types. It just scans blocks for 32-bit of 64-bit words and see, if they could be a pointers to another block(s). It also scans stack. It treats allocated blocks and stack as arrays of words, some of which may be pointers. And if it found a block allocated, which is “orphaned”, i.e., there are no pointer(s) to it from another block(s) or stack, this block considered unneeded, to be freed. Scanning process takes time, and this is what for GCs are criticized.

Also, GC like Boehm GC (for pure C) has function like GC_mallocc_atomic()—using it, you declare that the block allocated using this function will never contain any pointer(s) to other block(s). Maybe this could be a text string, or other type of data. (Indeed, GC_strdup() calls GC_mallocc_atomic().) GC will not scan it.

### 3.24 Loop optimizations

#### 3.24.1 Weird loop optimization

This is a simplest ever memcpy() function implementation:

```c
void memcpy (unsigned char* dst, unsigned char* src, size_t cnt) {
    size_t i;
```
for (i=0; i<cnt; i++)
    dst[i]=src[i];

At least MSVC 6.0 from the end of 1990s till MSVC 2013 can produce a really weird code (this listing is generated by MSVC 2013 x86):

```asm
_dst$ = 8 ; size = 4
_src$ = 12 ; size = 4
_cnt$ = 16 ; size = 4

_memcpy PROC
    mov    edx, DWORD PTR _cnt$[esp-4]
    test   edx, edx
    je     SHORT $LN1@f
    mov    eax, DWORD PTR _dst$[esp-4]
    push   esi
    mov    esi, DWORD PTR _src$[esp]
    sub    esi, eax
    ; ESI=src-dst, i.e., pointers difference
$LL8@f:
    mov    cl, BYTE PTR [esi+edx] ; load byte at "esi+dst" or at "src-dst+dst" at the beginning or at just "src"
    lea    eax, DWORD PTR [eax+1] ; dst++
    mov    BYTE PTR [eax-1], cl ; store the byte at "(dst++)--" or at just "dst" at the beginning
    dec    edx
    jne    SHORT $LL8@f
    pop    esi
$LN1@f:
    ret    0
_memcpy ENDP
```

This is weird, because how humans work with two pointers? They store two addresses in two registers or two memory cells. MSVC compiler in this case stores two pointers as one pointer (sliding dst in EAX) and difference between src and dst pointers (left unchanged over the span of loop body execution in ESI). (By the way, this is a rare case when ptrdiff_t data type can be used.) When it needs to load a byte from src, it loads it at diff + sliding dst and stores byte at just sliding dst.

This has to be some optimization trick. But I've rewritten this function to:

```asm
_f2 PROC
    mov    edx, DWORD PTR _cnt$[esp-4]
    test   edx, edx
    je     SHORT $LN1@f
    mov    eax, DWORD PTR _dst$[esp-4]
    push   esi
    mov    esi, DWORD PTR _src$[esp]
    ; eax=dst; esi=src
$LL8@f:
    mov    cl, BYTE PTR [esi+edx] ; load byte at "esi+dst" or at "src-dst+dst" at the beginning or at just "src"
    mov    BYTE PTR [eax+edx], cl
    dec    edx
    jne    SHORT $LL8@f
    pop    esi
$LN1@f:
    ret    0
_f2 ENDP
```

...and it works as efficient as the optimized version on my Intel Xeon E31220 @ 3.10GHz. Maybe, this optimization was targeted some older x86 CPUs of 1990s era, since this trick is used at least by ancient MS VC 6.0?

Any idea?

Hex-Rays 2.2 have a hard time recognizing patterns like that (hopefully, temporary?):

```c
void __cdecl f1(char *dst, char *src, size_t size)
{
    size_t counter; // edx@1
    char *sliding_dst; // eax@2
    char tmp; // cl@3
```
counter = size;
if ( size )
{
    sliding_dst = dst;
do
    {
        tmp = (sliding_dst++)[src - dst];   // difference (src-dst) is calculated once, at
        *(sliding_dst - 1) = tmp;
        --counter;
    }while ( counter );
}

Nevertheless, this optimization trick is often used by MSVC (not just in DIY\textsuperscript{47} homebrew memcpy() routines, but in many loops which uses two or more arrays), so it’s worth for reverse engineers to keep it in mind.

3.24.2 Another loop optimization

If you process all elements of some array which happens to be located in global memory, compiler can optimize it. For example, let’s calculate a sum of all elements of array of 128 int’s:

```c
#include <stdio.h>

int a[128];

int sum_of_a()
{
    int rt=0;
    for (int i=0; i<128; i++)
        rt=rt+a[i];
    return rt;
}

int main()
{
    // initialize
    for (int i=0; i<128; i++)
        a[i]=i;
    // calculate the sum
    printf("%d\n", sum_of_a());
}
```

Optimizing GCC 5.3.1 (x86) can produce this (IDA):

```
.text:080484B0 sum_of_a       proc near
.text:080484B0 mov     edx, offset a
.text:080484B5 xor     eax, eax
.text:080484B7 mov     esi, esi
.text:080484B9 lea     edi, [edi+0]
.text:080484C0 .text:080484C0 loc_80484C0: ; CODE XREF: sum_of_a+1B
.text:080484C0 add     eax, [edx]
.text:080484C2 add     edx, 4
.text:080484C5 cmp     edx, offset __libc_start_main@GLIBC_2.0
.text:080484CB jnz    short loc_80484C0
.text:080484CD rep     retn
.text:080484CD sum_of_a   endp
.text:080484CD
...
```

\textsuperscript{47}Do It Yourself

614
What the heck is __libc_start_main@GLIBC_2_0 at 0x080484C5? This is a label just after end of a[] array. The function can be rewritten like this:

```c
int sum_of_a_v2()
{
    int *tmp = a;
    int rt = 0;
    do
    {
        rt = rt + (*tmp);
        tmp++;
    }
    while (tmp < (a + 128));
    return rt;
}
```

First version has i counter, and the address of each element of array is to be calculated at each iteration. The second version is more optimized: the pointer to each element of array is always ready and is sliding 4 bytes forward at each iteration. How to check if the loop is ended? Just compare the pointer with the address just behind array’s end, which is, in our case, is happens to be address of imported __libc_start_main() function from Glibc 2.0. Sometimes code like this is confusing, and this is very popular optimizing trick, so that’s why I made this example.

My second version is very close to what GCC did, and when I compile it, the code is almost the same as in first version, but two first instructions are swapped:

```asm
.text:080484D0  public sum_of_a_v2
.text:080484D0  sum_of_a_v2 proc near
.text:080484D0  xor    eax, eax
.text:080484D2  mov    edx, offset a
.text:080484D7  mov    esi, esi
.text:080484D9  lea    edi, [edi+0]
.text:080484E0  text:080484E0  loc_80484E0:
.text:080484E0  add    eax, [edx]
.text:080484E2  add    edx, 4
.text:080484E5  cmp    edx, offset __libc_start_main@GLIBC_2_0
.text:080484EB  jnz    short loc_80484E0
.text:080484ED  rep    retn
.text:080484ED  sum_of_a_v2 endp
```

Needless to say, this optimization is possible if the compiler can calculate address of the end of array during compilation time. This happens if the array is global and it’s size is fixed.

However, if the address of array is unknown during compilation, but size is fixed, address of the label just behind array’s end can be calculated at the beginning of the loop.
3.25 More about structures

3.25.1 Sometimes a C structure can be used instead of array

Arithmetic mean

```c
#include <stdio.h>

int mean(int *a, int len)
{
    int sum=0;
    for (int i=0; i<len; i++)
        sum=sum+a[i];
    return sum/len;
};

struct five_ints
{
    int a0;
    int a1;
    int a2;
    int a3;
    int a4;
};

int main()
{
    struct five_ints a;
    a.a0=123;
    a.a1=456;
    a.a2=789;
    a.a3=10;
    a.a4=100;
    printf ("%d\n", mean(&a, 5));
    // test: https://www.wolframalpha.com/input/?i=mean(123,456,789,10,100)
};
```

This works: `mean()` function will never access behind the end of `five_ints` structure, because 5 is passed, meaning, only 5 integers will be accessed.

Putting string into structure

```c
#include <stdio.h>

struct five_chars
{
    char a0;
    char a1;
    char a2;
    char a3;
    char a4;
} __attribute__((aligned (1),packed));

int main()
{
    struct five_chars a;
    a.a0='h';
    a.a1='i';
    a.a2='!'
    a.a3='\n';
    a.a4=0;
    printf (&a); // prints "hi"
};
```

`((aligned (1),packed))` attribute must be used, because otherwise, each structure field will be aligned on 4-byte or 8-byte boundary.
Summary

This is just another example of how structures and arrays are stored in memory. Perhaps, no sane programmer will do something like in this example, except in case of some specific hack. Or maybe in case of source code obfuscation?

3.25.2 Unsized array in C structure

In some win32 structures we can find ones with last field defined as an array of one element:

```c
typedef struct _SYMBOL_INFO {
    ULONG SizeOfStruct;
    ULONG TypeIndex;
    ...
    ULONG MaxNameLen;
    TCHAR Name[1];
} SYMBOL_INFO, *PSYMBOL_INFO;
```

This is a hack, meaning, the last field is array of unknown size, which is to be calculated at the time of structure allocation.

Why: *Name* field may be short, so why to define it with some kind of *MAX_NAME* constant which can be 128, 256, or even bigger?

Why not to use pointer instead? Then you have to allocate two blocks: one for structure and the other for string. This may be slower and may require larger memory overhead. Also, you need dereference pointer (i.e., read address of the string from the structure)—not a big deal, but some people say this is still surplus cost.

This is also known as *struct hack*: [http://c-faq.com/struct/structhack.html](http://c-faq.com/struct/structhack.html).

Example:

```c
#include <stdio.h>

struct st
{
    int a;
    int b;
    char s[];
};

void f (struct st *s)
{
    printf ("%d %d %s\n", s->a, s->b, s->s);
    // f() can't replace s[] with bigger string - size of allocated block is unknown at this point
}

int main()
{
    #define STRING "Hello!"
    struct st *s=malloc(sizeof(struct st)+strlen(STRING)+1); // incl. terminating zero
    s->a=1;
    s->b=2;
    strcpy (s->s, STRING);
    f(s);
}
```

In short, it works because C has no array boundary checks. Any array is treated as having infinite size.

Problem: after allocation, the whole size of allocated block for structure is unknown (except for memory manager), so you can’t just replace string with larger string. You would still be able to do so if the field would be declared as something like `s[MAX_NAME]`.

In other words, you have a structure plus an array (or string) fused together in the single allocated memory block. Another problem is what you obviously can’t declare two such arrays in single structure, or to declare another field after such array.
Older compilers require to declare array with at least one element: \[s[1]\], newer allows to declare it as variable-sized array: \[s[]\]. This is also called flexible array member in C99 standard.

Read more about it in GCC documentation\(^4\), MSDN documentation\(^5\).

Dennis Ritchie (one of C creators) called this trick “unwarranted chumminess with the C implementation” (perhaps, acknowledging hackish nature of the trick).

Like it or not, use it or not: it is still another demonstration on how structures are stored in memory, that’s why I write about it.

### 3.25.3 Version of C structure

Many Windows programmers have seen this in MSDN:

<table>
<thead>
<tr>
<th>SizeOfStruct</th>
</tr>
</thead>
<tbody>
<tr>
<td>The size of the structure, in bytes. This member must be set to sizeof(SYMBOL_INFO).</td>
</tr>
</tbody>
</table>


Some structures like SYMBOL_INFO has started with this field indeed. Why? This is some kind of structure version.

Imagine you have a function which draws circle. It takes a single argument—a pointer to a structure with only three fields: X, Y and radius. And then color displays flooded a market, sometimes in 1980s. And you want to add color argument to the function. But, let’s say, you cannot add another argument to it (a lot of software use your API\(^6\) and cannot be recompiled). And if the old piece of software uses your API with color display, let your function draw a circle in (default) black white colors.

Another day you add another feature: circle now can be filled, and brush type can be set.

Here is one solution to the problem:

```c
#include <stdio.h>

struct ver1
{
    size_t SizeOfStruct;
    int coord_X;
    int coord_Y;
    int radius;
};

struct ver2
{
    size_t SizeOfStruct;
    int coord_X;
    int coord_Y;
    int radius;
    int color;
};

struct ver3
{
    size_t SizeOfStruct;
    int coord_X;
    int coord_Y;
    int radius;
    int color;
    int fill_brush_type; // 0 - do not fill circle
};

void draw_circle(struct ver3 *s) // latest struct version is used here
{
    // we presume SizeOfStruct, coord_X and coord_Y fields are always present
    printf(“We are going to draw a circle at %d:%d\n”, s->coord_X, s->coord_Y);
    if (s->SizeOfStruct>=sizeof(int)*5) // latest struct version is used here
    {
        // we presume SizeOfStruct, coord_X and coord_Y fields are always present
        printf(“We are going to draw a circle at %d:%d\n", s->coord_X, s->coord_Y);
        // do something
    }
}
```

\(^4\)https://gcc.gnu.org/onlinedocs/gcc/Zero-Length.html
\(^6\)Application Programming Interface
In other words, SizeOfStruct field takes a role of version of structure field. It could be enumerate type (1, 2, 3, etc.), but to set SizeOfStruct field to sizeof(struct...) is less prone to mistakes/bugs: we just write s.SizeOfStruct=sizeof(...) in caller’s code.

In C++, this problem is solved using inheritance (3.21.1 on page 547). You just extend your base class (let’s call it Circle), and then you will have ColoredCircle and then FilledColoredCircle, and so on. A current version of an object (or, more precisely, current type) will be determined using C++ RTTI.

So when you see SizeOfStruct somewhere in MSDN—perhaps this structure was extended at least once in past.
3.25.4 High-score file in “Block out” game and primitive serialization

Many videogames has high-score file, sometimes called “Hall of fame”. Ancient “Block out” game (3D tetris from 1989) isn’t exception, here is what we see at the end:

![High score table](image)

Figure 3.4: High score table

Now we can see that the file has changed after we added our name is `BLSCORE.DAT`. 

```
% xxd -g 1 BLSCORE.DAT
```

```
00000000: 0a 00 58 6e 69 61 2e 2e 2e 2e 2e 00 df 01 00 ..Xenia........
00000100: 00 30 33 3d 32 32 37 2d 32 30 31 38 00 50 61 75 6c .03-32-7-2018.Paul
00000200: 00 00 00 30 33 3d 32 32 37 2d 32 30 31 38 00 6a 65 73 ....03-32-7-2018.James
00000300: 00 00 00 30 33 3d 32 32 37 2d 32 30 31 38 00 4a 6f 68 6e ....03-32-7-2018.John
00000400: 00 46 01 00 00 30 33 3d 32 32 37 2d 32 30 31 38 00 77 7f 7b ...03-32-7-2018.Bob
00000500: 00 30 33 2d 32 37 2d 32 30 31 38 00 46 61 72 79 2e 2e ...03-27-2018.Mike..% 
00000600: 00 00 00 30 33 2d 32 37 2d 32 30 31 38 00 54 6f 6d 2e 2e ...03-27-2018.Tom...
00000700: 00 00 00 30 33 2d 32 37 2d 32 30 31 38 00 44 61 74 65 2e 2e ...03-27-2018.Charli
```

All entries are clearly visible. The very first byte is probably number of entries. Second is zero and, in fact, number of entries can be 16-bit value spanning over first two bytes.

Next, after “Xenia” name we see 0xDF and 0x01 bytes. Xenia has score of 479, and this is exactly 0x1DF in hexadecimal radix. So a high score value is probably 16-bit integer, or maybe 32-bit integer: there are two more zero bytes after.

Now let’s think about the fact that both array elements and structure elements are always placed in memory adjacently to each other. That enables us to write the whole array/structure to the file using simple `write()` or `fwrite()` function, and then restore it using `read()` or `fread()`, as simple as that. This is what is called serialization nowadays.

**Read**

Now let’s write C program to read highscore file:

#include <assert.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>

struct entry
{
    char name[11]; // incl. terminating zero
    uint32_t score;
    char date[11]; // incl. terminating zero
} __attribute__((aligned (1), packed));

struct highscore_file
{
    uint8_t count;
    uint8_t unknown;
    struct entry entries[10];
} __attribute__((aligned (1), packed));

struct highscore_file file;

int main(int argc, char* argv[])
{
    FILE* f=fopen(argv[1], "rb");
    assert (f!=NULL);
    size_t got=fread(&file, 1, sizeof(struct highscore_file), f);
    assert (got==sizeof(struct highscore_file));
    fclose(f);
    for (int i=0; i<file.count; i++)
    {
        printf("name=%s score=%d date=%s\n", 
               file.entries[i].name, 
               file.entries[i].score, 
               file.entries[i].date);
    }
}

We need GCC ((aligned (1),packed)) attribute so that all structure fields will be packed on 1-byte boundary.

Of course it works:

name=Xenia.... score=479 date=03-27-2018
name=Paul..... score=353 date=03-27-2018
name=John..... score=326 date=03-27-2018
name=James..... score=324 date=03-27-2018
name=Charlie... score=234 date=03-27-2018
name=Mike...... score=181 date=03-27-2018
name=Phil....... score=172 date=03-27-2018
name=Mary...... score=123 date=03-27-2018
name=Tom....... score=119 date=03-27-2018
name=Bob....... score=119 date=03-27-2018

(Needless to say, each name is padded with dots, both on screen and in the file, perhaps, for æsthetical reasons.)

Write

Let’s check if we right about width of score value. Is it really has 32 bits?

int main(int argc, char* argv[])
{
    FILE* f=fopen(argv[1], "rb");
    assert (f!=NULL);
    size_t got=fread(&file, 1, sizeof(struct highscore_file), f);
    assert (got==sizeof(struct highscore_file));
    fclose(f);
    strcpy(file.entries[1].name, "Mallory...");
    file.entries[1].score=12345678;
}
Let's run Blockout:

```c
strcpy (file.entries[1].date, "08-12-2016");

f=fopen(argv[1], "wb");
assert (f!=NULL);
got=fwrite(&file, 1, sizeof(struct highscore_file), f);
assert (got==sizeof(struct highscore_file));
fclose(f);
};
```

Figure 3.5: High score table

First two digits (1 and 2) are truncated: 12345678 becomes 345678. Perhaps, this is formatting issues... but the number is almost correct. Now I'm changing it to 999999 and run again:

Figure 3.6: High score table

Now it's correct. Yes, high score value is 32-bit integer.
Is it serialization?

...almost. Serialization like this is highly popular in scientific and engineering software, where efficiency and speed is much more important than converting into XML or JSON and back.

One important thing is that you obviously cannot serialize pointers, because each time you load the file into memory, all the structures may be allocated in different places.

But: if you work on some kind of low-cost MCU with simple OS on it and you have your structures allocated at always same places in memory, perhaps you can save and restore pointers as well.

Random noise

When I prepared this example, I had to run “Block out” many times and played for it a bit to fill high-score table with random names.

And when there were just 3 entries in the file, I saw this:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>03 00 54 6f 6d 61 73 2e 2e 2e 2e 2e 00 da 2a 00</td>
</tr>
<tr>
<td>0x00000010</td>
<td>00 30 38 2d 31 32 2d 32 30 31 36 00 43 68 61 72</td>
</tr>
<tr>
<td>0x00000020</td>
<td>6c 69 65 2e 2e 2e 2e 00 8b 1e 00 00 30 38 2d 31 32</td>
</tr>
<tr>
<td>0x00000030</td>
<td>2d 32 30 31 36 00 4a 6f 6e 2e 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x00000040</td>
<td>00 80 00 00 00 00 30 38 2d 31 32 2d 32 30 31 36 00</td>
</tr>
<tr>
<td>0x00000050</td>
<td>00 00 57 c8 a2 01 06 01 ba f9 47 c7 05 00 f8 4f</td>
</tr>
<tr>
<td>0x00000060</td>
<td>06 01 06 01 a6 32 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x00000070</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x00000080</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x00000090</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x000000a0</td>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x000000b0</td>
<td>8c f9 f6 c5 05 00 f8 4f 00 02 06 01 a6 32 06 01</td>
</tr>
<tr>
<td>0x000000c0</td>
<td>00 00 98 f9 f2 c8 05 00 f8 4f 00 02 a6 32 a2 4f</td>
</tr>
<tr>
<td>0x000000d0</td>
<td>00 c1 a6 32 a6 32 4f 4f aa f9 39 c1 a6 32 06 01</td>
</tr>
<tr>
<td>0x000000e0</td>
<td>b4 f9 2b c5 a6 32 e1 4f c7 e2 a2 01 82 72 c6 4f</td>
</tr>
<tr>
<td>0x000000f0</td>
<td>30 c6 05 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x00000100</td>
<td>a6 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

The first byte has value of 3, meaning there are 3 entries. And there are 3 entries present. But then we see a random noise at the second half of file.

The noise is probably has its origins in uninitialized data. Perhaps, “Block out” allocated memory for 10 entries somewhere in heap, where, obviously, some pseudorandom noise (left from something else) was present. Then it set first/second byte, fill 3 entries, and then it never touched 7 entries left, so they are written to the file as is.

When “Block out” loads high score file at the next run, it reads number of entries from the first/second byte (3) and then completely ignores what is after it.

This is common problem. Not a problem in strict sense: it’s not a bug, but information can be exposed outwards.

Microsoft Word versions from 1990s has been often left pieces of previously edited texts into the *.doc* files. It was some kind of amusement back then, to get a .doc file from someone, then open it in a hexadecimal editor and read something else, what has been edited on that computer before.

The problem can be even much more serious: Heartbleed bug in OpenSSL.

Homework

“Block out” has several polycubes (flat/basic/extended), size of pit can be configured, etc. And it seems, for each configuration, “Block out” has its own high score table. I’ve noticed that some information is probably stored in BLSCORE.IDX file. This can be a homework for hardcore “Block out” fans—to understand its structure as well.

The “Block out” files are here: [http://beginners.re/examples/blockout.zip](http://beginners.re/examples/blockout.zip) (including the binary high score files I’ve used in this example). You can use DosBox to run it.

---

52 Extensible Markup Language

53 JavaScript Object Notation
3.26  memmove() and memcpy()

The difference between these standard functions is that memcpy() blindly copies a block to another place, while memmove() correctly handles overlapping blocks. For example, you want to tug a string two bytes forward:

```
|.|.|.|h|e|l|l|o|...` -> `|h|e|l|l|o|...
```

memcpy() which copies 32-bit or 64-bit words at once, or even SIMD, will obviously fail here, a byte-wise copy routine must be used instead.

Now even more advanced example, insert two bytes in front of string:

```
|h|e|l|l|o|...` -> `|.|.|h|e|l|l|o|...
```

Now even byte-wise memory copy routine will fail, you have to copy bytes starting at the end.

That’s a rare case where DF x86 flag is to be set before REP MOVSB instruction: DF defines direction, and now we must move backwardly.

The typical memmove() routine works like this: 1) if source is below destination, copy forward; 2) if source is above destination, copy backward.

This is memmove() from uClibc:

```c
void *memmove(void *dest, const void *src, size_t n)
{
    int eax, ecx, esi, edi;
    __asm__ __volatile__ ("movl %%eax, %%edi\n"
        "cmpl  %%esi, %%eax\n"
        "je  2f\n" /* (optional) src == dest -> NOP */
        "jb  1f\n" /* src > dest -> simple copy */
        "leal -1(%%esi,%%ecx), %%esi\n"
        "leal -1(%%eax,%%ecx), %%edi\n"
        "std\n"
    :
        "1: rep; movsb\n"
    :
        "cld\n"
    "2: \n"
        : "=c" (ecx), "=S" (esi), "=a" (eax), "=D" (edi)
        : "=O" (n), "=1" (src), "=2" (dest)
        : "memory"
    );
    return (void*)eax;
}
```

In the first case, REP MOVSB is called with DF flag cleared. In the second, DF is set, then cleared.

More complex algorithm has the following piece in it:

“if difference between source and destination is larger than width of word, copy using words rather than bytes, and use byte-wise copy to copy unaligned parts”.

This how it happens in Glibc 2.24 in non-optimized C part.

Given all that, memmove() may be slower than memcpy(). But some people, including Linus Torvalds, argue that memcpy() should be an alias (or synonym) of memmove(), and the latter function must just check at start, if the buffers are overlapping or not, and then behave as memcpy() or memmove(). Nowadays, check for overlapping buffers is very cheap, after all.

3.26.1 Anti-debugging trick

I’ve heard about anti-debugging trick where all you need is just set DF to crash the process: the very next memcpy() routine will lead to crash because it copies backwardly. But I can’t check this: it seems all memory copy routines clear/set DF as they want to. On the other hand, memmove() from uClibc I cited here, has no explicit clear of DF (it assumes DF is always clear?), so it can really crash.

---

54https://bugzilla.redhat.com/show_bug.cgi?id=638477#c132
3.27 setjmp/longjmp

setjmp/longjmp is a mechanism in C which is very similar to throw/catch mechanism in C++ and other higher-level PLs. Here is an example from zlib:

```c
/* return if bits() or decode() tries to read past available input */
if (setjmp(s.env) != 0) /* if came back here via longjmp(), */
    err = 2; /* then skip decomp(), return error */
else
    err = decomp(&s); /* decompress */
```

( zlib/contrib/blast/blast.c )

Call to setjmp() saves current PC, SP and other registers into env structure, then it returns 0.

In case of error, longjmp() teleporting you into the point after right after setjmp() call, as if setjmp() call returned non-null value (which was passed to longjmp()). This reminds as fork() syscall in UNIX.

Now let’s take a look on distilled example:

```c
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void f2()
{
    printf ("%s() begin\n", __FUNCTION__);
    // something odd happened here
    longjmp (env, 1234);
    printf ("%s() end\n", __FUNCTION__);
}

void f1()
{
    printf ("%s() begin\n", __FUNCTION__);
    f2();
    printf ("%s() end\n", __FUNCTION__);
}

int main()
{
    int err=setjmp(env);
    if (err==0)
    {
        f1();
    }
    else
    {
        printf ("Error %d\n", err);
    }
};
```

If we run it, we will see:
**jmp_buf** structure usually comes undocumented, to preserve forward compatibility.

Let’s see how `setjmp()` implemented in MSVC 2013 x64:

```assembly
; RCX = address of jmp_buf
mov [rcx],rax
mov [rcx+8],rbx
mov [rcx+18h],rbp
mov [rcx+20h],rsi
mov [rcx+28h],rdi
mov [rcx+30h],r12
mov [rcx+38h],r13
mov [rcx+40h],r14
mov [rcx+48h],r15
lea r8, [rsp+arg_0]
mov [rcx+10h], r8
mov r8, [rsp+0] ; get saved RA from stack
mov [rcx+50h], r8 ; save it
stmxcsr dword ptr [rcx+58h]
fnstcw word ptr [rcx+5Ch]
movdqa xmmword ptr [rcx+60h], xmm6
movdqa xmmword ptr [rcx+70h], xmm7
movdqa xmmword ptr [rcx+80h], xmm8
movdqa xmmword ptr [rcx+90h], xmm9
movdqa xmmword ptr [rcx+0A0h], xmm10
movdqa xmmword ptr [rcx+0B0h], xmm11
movdqa xmmword ptr [rcx+0C0h], xmm12
movdqa xmmword ptr [rcx+0D0h], xmm13
movdqa xmmword ptr [rcx+0E0h], xmm14
movdqa xmmword ptr [rcx+0F0h], xmm15
retn
```

It just populates **jmp_buf** structure with current values of almost all registers. Also, current value of RA is taken from the stack and saved in **jmp_buf**: it will be used as new value of PC in future.

Now `longjmp()`:

```assembly
; RCX = address of jmp_buf
mov rax, rdx
mov rbx, [rcx+8]
mov rsi, [rcx+20h]
mov rdi, [rcx+28h]
mov r12, [rcx+30h]
mov r13, [rcx+38h]
mov r14, [rcx+40h]
mov r15, [rcx+48h]
ldecsr dword ptr [rcx+58h]
fnclex
fldcw word ptr [rcx+5Ch]
movdqa xmm6, xmmword ptr [rcx+60h]
movdqa xmm7, xmmword ptr [rcx+70h]
movdqa xmm8, xmmword ptr [rcx+80h]
movdqa xmm9, xmmword ptr [rcx+90h]
movdqa xmm10, xmmword ptr [rcx+0A0h]
movdqa xmm11, xmmword ptr [rcx+0B0h]
movdqa xmm12, xmmword ptr [rcx+0C0h]
movdqa xmm13, xmmword ptr [rcx+0D0h]
movdqa xmm14, xmmword ptr [rcx+0E0h]
```

626
It just restores (almost) all registers, takes RA from structure and jumps there. This effectively works as if setjmp() returned to caller. Also, RAX is set to be equal to the second argument of longjmp(). This works as if setjmp() returned non-zero value at first place.

As a side effect of SP restoration, all values in stack which has been set and used between setjmp() and longjmp() calls are just dropped. They will not be used anymore. Hence, longjmp() usually jumps backwards 55.

This implies that, unlike in throw/catch mechanism in C++, no memory will be freed, no destructors will be called, etc. Hence, this technique sometimes can be dangerous. Nevertheless, it’s still quite popular. It’s still used in Oracle RDBMS.

It also has an unexpected side-effect: if some buffer has been overflown inside of a function (maybe due to remote attack), and a function wants to report error, and it calls longjmp(), overwritten stack part just gets unused.

As an exercise, you can try to understand why not all registers are saved. Why XMM0-XMM5 and other registers are skipped?

### 3.28 Other weird stack hacks

#### 3.28.1 Accessing arguments/local variables of caller

From C/C++ basics we know that this is impossible for a function to access arguments of caller function or its local variables.

Nevertheless, it’s possible using dirty hacks. For example:

```c
#include <stdio.h>

void f(char *text)
{
    // print stack
    int *tmp=&text;
    for (int i=0; i<20; i++)
    {
        printf("0x%x\n", *tmp);
        tmp++;
    }
}

void draw_text(int X, int Y, char* text)
{
    f(text);
    printf("We are going to draw [%s] at %d:%d\n", text, X, Y);
}

int main()
{
    printf("address of main()=0x%x\n", &main);
    printf("address of draw_text()=0x%x\n", &draw_text);
    draw_text(100, 200, "Hello!");
}
```

On 32-bit Ubuntu 16.04 and GCC 5.4.0, I got this:

```
address of main()=0x80484f8
address of draw_text()=0x80484cb
```

55However, there are some people who can use it for much more complicated things, imitating coroutines, etc: https://www.embeddedrelated.com/showarticle/455.php, http://fanf.livejournal.com/105413.html
(Comments are mine.)

Since `f()` starting to enumerate stack elements at its first argument, the first stack element is indeed a pointer to “Hello!” string. We see its address is also used as third argument to `draw_text()` function.

In `f()` we could read all functions arguments and local variables if we know exact stack layout, but it’s always changed, from compiler to compiler. Various optimization levels affect stack layout greatly.

But if we can somehow detect information we need, we can use it and even modify it. As an example, I’ll rework `f()` function:

```c
void f(char *text)
{
    // find 100, 200 values pair and modify the second on
    tmp=&text;
    for (int i=0; i<20; i++)
    {
        if (*tmp==100 && *(tmp+1)==200)
        {
            printf("found\n");
            *(tmp+1)=210; // change 200 to 210
            break;
        }
        tmp++;
    }
}
```

Holy moly, it works:

```
found
We are going to draw [Hello!] at 100:210
```

### Summary

It’s extremely dirty hack, intended to demonstrate stack internals. I never ever seen or heard that anyone used this in a real code. But still, this is a good example.

### Exercise

The example has been compiled without optimization on 32-bit Ubuntu using GCC 5.4.0 and it works. But when I turn on `-O3` maximum optimization, it’s failed. Try to find why.

Use your favorite compiler and OS, try various optimization levels, find if it works and if it doesn’t, find why.
3.28.2 Returning string

This is classic bug from Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999):

```c
#include <stdio.h>

char* amsg(int n, char* s)
{
    char buf[100];
    sprintf(buf, "error %d: %s\n", n, s);
    return buf;
};

int main()
{
    printf("%s\n", amsg(1234, "something wrong"));
};
```

It would crash. First, let's understand, why.

This is a stack state before `amsg()` return:

```
(lower addresses)
...
[amsg(): 100 bytes] <- current SP
[two amsg arguments]
[something else]
[main() local variables]
...
```

When `amsg()` returns control flow to `main()`, so far so good. But `printf()` is called from `main()`, which is, in turn, use stack for its own needs, zapping 100-byte buffer. A random garbage will be printed at the best.

Hard to believe, but I know how to fix this problem:

```c
#include <stdio.h>

char* amsg(int n, char* s)
{
    char buf[100];
    sprintf(buf, "error %d: %s\n", n, s);
    return buf;
};

char* interim (int n, char* s)
{
    char large_buf[8000];
    // make use of local array.
    // it will be optimized away otherwise, as useless.
    large_buf[0]=0;
    return amsg (n, s);
};

int main()
{
    printf("%s\n", interim (1234, "something wrong"));
};
```

It will work if compiled by MSVC 2013 with no optimizations and with /GS- option\(^5\). MSVC will warn:  
\(^5\)Turn off buffer security check
“warning C4172: returning address of local variable or temporary”, but the code will run and message will be printed. Let’s see stack state at the moment when amsg() returns control to interim():

(lower addresses)
...
[amsg(): 100 bytes]                      <- current SP
[RA]                                    [two amsg() arguments]
[interim() stuff, incl. 8000 bytes]     [something else]
[main() local variables]                ...
(upper addresses)

Now the stack state at the moment when interim() returns control to main():

(lower addresses)
...
[amsg(): 100 bytes]                      <- current SP
[RA]                                    [two amsg() arguments]
[interim() stuff, incl. 8000 bytes]     [something else]
[main() local variables]                ...
(upper addresses)

So when main() calls printf(), it uses stack at the place where interim()'s buffer was allocated, and doesn’t zap 100 bytes with error message inside, because 8000 bytes (or maybe much less) is just enough for everything printf() and other descending functions do!

It may also work if there are many functions between, like: main() → f1() → f2() → f3() ... → amsg(), and then the result of amsg() is used in main(). The distance between SP in main() and address of buf[] must be long enough,

This is why bugs like these are dangerous: sometimes your code works (and bug can be hiding unnoticed), sometimes not. Bugs like these are jokingly called heisenbugs or schrödinbugs.

### 3.29 OpenMP

OpenMP is one of the simplest ways to parallelize simple algorithms.

As an example, let’s try to build a program to compute a cryptographic nonce.

In my simplistic example, the nonce is a number added to the plain unencrypted text in order to produce a hash with some specific features.

For example, at some step, the Bitcoin protocol requires to find such nonce so the resulting hash contains a specific number of consecutive zeros. This is also called proof of work (i.e., the system proves that it did some intensive calculations and spent some time for it).

My example is not related to Bitcoin in any way, it will try to add numbers to the “hello, world!_<number>” string in order to find such number that when “hello, world!_<number>” is hashed with the SHA512 algorithm, it will contain at least 3 zero bytes.

Let’s limit our brute-force to the interval in 0..INT32_MAX-1 (i.e., 0x7FFFFFFE or 2147483646).

The algorithm is pretty straightforward:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```
#include <time.h>
#include "sha512.h"

int found=0;
int32_t checked=0;

int32_t* __min;
int32_t* __max;

time_t start;

#ifdef __GNUC__
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
#endif

void check_nonce (int32_t nonce)
{
    uint8_t buf[32];
    struct sha512_ctx ctx;
    uint8_t res[64];

    // update statistics
    int t=omp_get_thread_num();
    if (__min[t]==-1)
        __min[t]=nonce;
    if (__max[t]==-1)
        __max[t]=nonce;
    __min[t]=min(__min[t], nonce);
    __max[t]=max(__max[t], nonce);

    // idle if valid nonce found
    if (found)
        return;

    memset (buf, 0, sizeof(buf));
    sprintf (buf, "hello, world!_%d", nonce);

    sha512_init_ctx (&ctx);
    sha512_process_bytes (buf, strlen(buf), &ctx);
    sha512_finish_ctx (&ctx, &res);
    if (res[0]==0 && res[1]==0 & res[2]==0)
    {
        printf ("found (thread %d): [%s]. seconds spent=%d\n", t, buf, time(NULL)-start);
        found=1;
    }
    #pragma omp atomic
    checked++;

    #pragma omp critical
    if ((checked % 100000)==0)
        printf ("checked=%d\n", checked);
}

int main()
{
    int32_t i;
    int threads=omp_get_max_threads();
    printf ("threads=%d\n", threads);

    __min=(int32_t*)malloc(threads*sizeof(int32_t));
    __max=(int32_t*)malloc(threads*sizeof(int32_t));
    for (i=0; i<threads; i++)
        __min[i]=__max[i]=-1;

    start=time(NULL);
}
#pragma omp parallel for
for (i=0; i<INT32_MAX; i++)
    check_nonce (i);

for (i=0; i<threads; i++)
    printf ("__min[%d]=%x __max[%d]=%x
", i, __min[i], i, __max[i]);

free(__min); free(__max);
};

The check_nonce() function just adds a number to the string, hashes it with the SHA512 algorithm and checks for 3 zero bytes in the result.

A very important part of the code is:

#pragma omp parallel for
for (i=0; i<INT32_MAX; i++)
    check_nonce (i);

Yes, that simple, without #pragma we just call check_nonce() for each number from 0 to INT32_MAX (0x7fffffff or 2147483647). With #pragma, the compiler adds some special code which slices the loop interval into smaller ones, to run them on all CPU cores available 57.

The example can be compiled 58 in MSVC 2012:

```bash
cl openmp_example.c sha512.obj /openmp /O1 /Zi /Faopenmp_example.asm
```

Or in GCC:

```bash
gcc -fopenmp 2.c sha512.c -S -masm=intel
```

### 3.29.1 MSVC

Now this is how MSVC 2012 generates the main loop:

```
Listing 3.125: MSVC 2012
push OFFSET _main$omp$1
push 0
push 1
call __vcomp_fork
add esp, 16
```

All functions prefixed by vcomp are OpenMP-related and are stored in the vcomp*.dll file. So here a group of threads is started.

Let's take a look on _main$omp$1:

```
Listing 3.126: MSVC 2012
$T1 = -8    ; size = 4
$T2 = -4    ; size = 4
_main$omp$1 PROC
push ebp
mov ebp, esp
push ecx
push ecx
push esi
lea eax, DWORD PTR $T2[ebp]
push eax
lea eax, DWORD PTR $T1[ebp]
push eax
push 1
push 1
push 2147483646   ; 7fffffffH
push 0
call __vcomp_for_static_simple_init
```

57 N.B.: This is intentionally simplest possible example, but in practice, the usage of OpenMP can be harder and more complex
58 sha512.(c|h) and u64.h files can be taken from the OpenSSL library: [http://go.yurichev.com/17324](http://go.yurichev.com/17324)
This function is to be started $n$ times in parallel, where $n$ is the number of CPU cores. vcomp_for_static_simple_init() calculates the interval for the for() construct for the current thread, depending on the current thread's number.

The loop's start and end values are stored in the $T1$ and $T2$ local variables. You may also notice 7ffffffeh (or 2147483646) as an argument to the vcomp_for_static_simple_init() function—this is the number of iterations for the whole loop, to be divided evenly.

Then we see a new loop with a call to the check_nonce() function, which does all the work.

Let's also add some code at the beginning of the check_nonce() function to gather statistics about the arguments with which the function has been called.

This is what we see when we run it:

```
threads=4
...
checked=2800000
checked=3000000
checked=3200000
checked=3300000
found (thread 3): [hello, world!_1611446522]. seconds spent=3
__min[0]=0x00000000 __max[0]=0xffffffff
__min[1]=0x20000000 __max[1]=0x3fffffff
__min[2]=0x40000000 __max[2]=0x5fffffff
__min[3]=0x60000000 __max[3]=0x7fffffff
```

Yes, the result is correct, the first 3 bytes are zeros:

```
C:\...\sha512sum test
000000f4a8fac5a4ed38794da4c1e39f54279ad5d9bb3c5465cdf57adaf60403
df6e3fe6019f5764cf9975e505a7395fed780fee59eb38dd4c0279c8144672e2 *test
```

The running time is $\approx 2..3$ seconds on 4-core Intel Xeon E3-1220 3.10 GHz. In the task manager we see 5 threads: 1 main thread + 4 more. No further optimizations are done to keep this example as small and clear as possible. But probably it can be done much faster. My CPU has 4 cores, that is why OpenMP started exactly 4 threads.

By looking at the statistics table we can clearly see how the loop has been sliced into 4 even parts. Oh well, almost even, if we don't consider the last bit.

There are also pragmas for atomic operations.

Let's see how this code is compiled:

```
#pragma omp atomic
    checked++;

#pragma omp critical
    if ((checked % 100000)==0)
        printf ("checked=%d\n", checked);
```

Listing 3.127: MSVC 2012
As it turns out, the `vcomp_atomic_add_i4()` function in the `vcomp*.dll` is just a tiny function with the LOCK XADD instruction in it.

`vcomp_enter_critsect()` eventually calling win32 API function `EnterCriticalSection()`.

### 3.29.2 GCC

GCC 4.8.1 produces a program which shows exactly the same statistics table, so, GCC's implementation divides the loop in parts in the same fashion.

#### Listing 3.128: GCC 4.8.1

```
mov  edi, OFFSET FLAT:main._omp_fn.0
call GOMP_parallel_start
mov edi, 0
call main._omp_fn.0
call GOMP_parallel_end
```

Unlike MSVC's implementation, what GCC code does is to start 3 threads, and run the fourth in the current thread. So there are 4 threads instead of the 5 in MSVC.

Here is the `main._omp_fn.0` function:

#### Listing 3.129: GCC 4.8.1

```
main._omp_fn.0:
push rbp
mov rbp, rsp
push rbx
sub rbp, 40
mov QWORD PTR [rbp-40], rdi
call omp_get_num_threads
mov ebx, eax
call omp_get_thread_num
mov esi, eax
mov eax, 2147483647 ; 0xFFFFFFFF
cdq
idiv ebx
mov ecx, eax
mov eax, 2147483647 ; 0xFFFFFFFF
```
Here we see the division clearly: by calling `omp_get_num_threads()` and `omp_get_thread_num()` we get the number of threads running, and also the current thread’s number, and then determine the loop’s interval. Then we run `check_nonce()`.

GCC also inserted the LOCK ADD instruction right in the code, unlike MSVC, which generated a call to a separate DLL function:

```
lock add DWORD PTR checked[rip], 1
```

```
lock add DWORD PTR checked[rip], 1
```

The functions prefixed with GOMP are from GNU OpenMP library. Unlike vcomp*.dll, its source code is freely available: [GitHub](https://github.com).
3.30 Signed division using shifts

Unsigned division by \(2^n\) numbers is easy, just use bit shift right by \(n\). Signed division by \(2^n\) is easy as well, but some correction needs to be done before or after shift operation.

First, most CPU architectures support two right shift operations: logical and arithmetical. During logical shift right, free bit(s) at left are set to zero bit(s). This is SHR in x86. During arithmetical shift right, free bit(s) at left are set equal to the bit which was at the same place. Thus, it preserves sign bit while shifting. This is SAR in x86.

Interesting to know, there is no special instruction for arithmetical shift left, because it works just as logical shift left. So, SAL and SHL instructions in x86 are mapped to the same opcode. Many disassemblers don’t even know about SAL instruction and decode this opcode as SHL.

Hence, arithmetical shift right is used for signed numbers. For example, if you shift -4 (11111100b) by 1 bit right, logical shift right operation will produce 01111110b, which is 126. Arithmetical shift right will produce 11111110b, which is -2. So far so good.

What if we need to divide -5 by 2? This is -2.5, or just -2 in integer arithmetic. -5 is 11111011b, by shifting this value by 1 bit right, we’ll get 11111101b, which is -3. This is slightly incorrect.

Another example: \(-\frac{1}{2} = -0.5\) or just 0 in integer arithmetic. But -1 is 11111111b, and 11111111b \(\gg\) 1 = 11111111b, which is -1 again. This is also incorrect.

One solution is to add 1 to the input value if it’s negative.

That is why, if we compile \(x/2\) expression, where \(x\) is signed int, GCC 4.8 will produce something like that:

```assembly
mov eax, edi
shr eax, 31 ; isolate leftmost bit, which is 1 if the number is negative and 0 if positive
add eax, edi ; add 1 to the input value if it's negative, do nothing otherwise
sar eax ; arithmetical shift right by one bit
ret
```

If you divide by 4, 3 needs to be added to the input value if it’s negative. So this is what GCC 4.8 does for \(x/4\):

```assembly
lea eax, [rdi+3] ; prepare x+3 value ahead of time
test edi, edi
    ; if the sign is not negative (i.e., positive), move input value to EAX
    ; if the sign is negative, x+3 value is left in EAX untouched
    cmovns eax, edi
    ; do arithmetical shift right by 2 bits
sar eax, 2
ret
```

If you divide by 8, 7 will be added to the input value, etc.

MSVC 2013 is slightly different. This is division by 2:

```assembly
mov eax, DWORD PTR _a$[esp-4]
    ; sign-extend input value to 64-bit value into EDX:EAX
    ; effectively, that means EDX will be set to 0FFFFFFFFh if the input value is negative
    ; ... or to 0 if positive
cdq
    ; subtract -1 from input value if it's negative
    ; this is the same as adding 1
sub eax, edx
    ; do arithmetical shift right
sar eax, 1
ret 0
```

Division by 4 in MSVC 2013 is little more complex:

```assembly
mov eax, DWORD PTR _a$[esp-4]
cdq
    ; now EDX is 0FFFFFFFFh if input value is negative
    ; EDX is 0 if it's positive
    and edx, 3
    ; now EDX is 3 if input is negative or 0 otherwise
ret
```
Division by 8 in MSVC 2013 is similar, but 3 bits from EDX is taken instead of 2, producing correction value of 7 instead of 3.

Sometimes, Hex-Rays 6.8 can’t handle such code correctly, and it may produce something like this:

```c
int v0;
...
__int64 v14 ...
...
v14 = ...;
v0 = ((signed int)v14 - HIDWORD(v14)) >> 1;
```

... it can be safely rewritten to v0 = v14/2.

Hex-Rays 6.8 can also handle signed division by 4 like that:

```c
result = ((BYTE4(v25) & 3) + (signed int)v25) >> 2;
```

... can be rewritten to v25 / 4.

Also, such correction code is used often when division is replaced by multiplication by magic numbers: read Mathematics for Programmers about multiplicative inverse. And sometimes, additional shifting is used after multiplication. For example, when GCC optimizes x/10, it can’t find multiplicative inverse for 10, because diophantine equation has no solutions. So it generates code for x/5 and then adds arithmetical shift right operation by 1 bit, to divide the result by 2. Of course, this is true only for signed integers.

So here is division by 10 by GCC 4.8:

```c
mov eax, edi
mov edx, 1717986919 ; magic number
sar edi, 31 ; isolate leftmost bit (which reflects sign)
imul edx ; multiplication by magic number (calculate x/5)
sar edx, 2 ; now calculate (x/5)/2
; subtract -1 (or add 1) if the input value is negative.
; do nothing otherwise:
sub edx, edi
mov eax, edx
ret
```

Summary: $2^n - 1$ must be added to input value before arithmetical shift, or 1 must be added to the final result after shift. Both operations are equivalent to each other, so compiler developers may choose what is more suitable to them. From the reverse engineer’s point of view, this correction is a clear evidence that the value has signed type.

### 3.31 Another heisenbug

Sometimes, array (or buffer) can overflow due to fencepost error:

```c
#include <stdio.h>

int array1[128];
int important_var1;
int important_var2;
int important_var3;
int important_var4;
int important_var5;
```

---

61 https://yurichev.com/writings/Math-for-programmers.pdf
int main()
{
    important_var1=1;
    important_var2=2;
    important_var3=3;
    important_var4=4;
    important_var5=5;

    array1[0]=123;
    array1[128]=456; // BUG

    printf ("important_var1=%d\n", important_var1);
    printf ("important_var2=%d\n", important_var2);
    printf ("important_var3=%d\n", important_var3);
    printf ("important_var4=%d\n", important_var4);
    printf ("important_var5=%d\n", important_var5);
}

This is what this program printed in my case (non-optimized GCC 5.4 x86 on Linux):

```plaintext
important_var1=1
important_var2=456
important_var3=3
important_var4=4
important_var5=5
```

As it happens, `important_var2` has been placed by compiler right after `array1[]`:

Listing 3.131: objdump -x

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Size</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0804a040</td>
<td>g</td>
<td>0.bss</td>
<td>00000200</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>array1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0804a240</td>
<td>g</td>
<td>0.bss</td>
<td>00000004</td>
</tr>
<tr>
<td>0804a244</td>
<td>g</td>
<td>0.bss</td>
<td>00000004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>important_var2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0804a248</td>
<td>g</td>
<td>0.bss</td>
<td>00000004</td>
</tr>
<tr>
<td>0804a24c</td>
<td>g</td>
<td>0.bss</td>
<td>00000004</td>
</tr>
<tr>
<td>0804a250</td>
<td>g</td>
<td>0.bss</td>
<td>00000004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>important_var1</td>
</tr>
</tbody>
</table>

Another compiler can arrange variables in another order, and another variable would be zapped. This is also *heisenbug* (3.28.2 on page 630)—bug may appear or may left unnoticed depending on compiler version and optimization switches.

If all variables and arrays are allocated in local stack, stack protection may be triggered, or may not. However, Valgrind can find bugs like these.

Related example in the book (Angband game): 1.27 on page 301.

### 3.32 The case of forgotten return

Let’s revisit the “attempt to use the result of a function returning *void*” part: .

This is a bug I once hit.

And this is also yet another demonstration, how C/C++ places return value into EAX/RAX register.

In the piece of code like that, I forgot to add return:

```c
#include <stdio.h>
#include <stdlib.h>

struct color
{
    int R;
    int G;
    int B;
};

struct color* create_color (int R, int G, int B)
{
```
struct color* rt=(struct color*)malloc(sizeof(struct color));
    rt->R=R;
    rt->G=G;
    rt->B=B;
    // must be "return rt;" here
}

int main()
{
    struct color* a=create_color(1,2,3);
    printf("%d %d %d\n", a->R, a->G, a->B);
}

Non-optimizing GCC 5.4 silently compiles this with no warnings. And the code works! Let’s see, why:

Listing 3.132: Non-optimizing GCC 5.4

create_color:
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov DWORD PTR [rbp-20], edi
    mov DWORD PTR [rbp-24], esi
    mov DWORD PTR [rbp-28], edx
    mov edi, 12
    call malloc
    ; RAX is pointer to newly allocated buffer
    ; now fill it with R/G/B:
    mov QWORD PTR [rbp-8], rax
    mov rax, QWORD PTR [rbp-8]
    mov edx, DWORD PTR [rbp-20]
    mov DWORD PTR [rax], edx
    mov rax, QWORD PTR [rbp-8]
    mov edx, DWORD PTR [rbp-24]
    mov DWORD PTR [rax+4], edx
    mov rax, QWORD PTR [rbp-8]
    mov edx, DWORD PTR [rbp-28]
    mov DWORD PTR [rax+8], edx
    nop
    leave
    ; RAX wasn’t modified till that point!
    ret

If I add return rt;, the only instruction is added at the end, which is redundant:

Listing 3.133: Non-optimizing GCC 5.4

create_color:
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov DWORD PTR [rbp-20], edi
    mov DWORD PTR [rbp-24], esi
    mov DWORD PTR [rbp-28], edx
    mov edi, 12
    call malloc
    ; RAX is pointer to buffer
    mov QWORD PTR [rbp-8], rax
    mov rax, QWORD PTR [rbp-8]
    mov edx, DWORD PTR [rbp-20]
    mov DWORD PTR [rax], edx
    mov rax, QWORD PTR [rbp-8]
    mov edx, DWORD PTR [rbp-24]
    mov DWORD PTR [rax+4], edx
    mov rax, QWORD PTR [rbp-8]
    mov edx, DWORD PTR [rbp-28]
    mov DWORD PTR [rax+8], edx
    ; reload pointer to RAX again, and this is redundant operation...
    mov rax, QWORD PTR [rbp-8]; new instruction
    leave
Bugs like that are very dangerous, sometimes they appear, sometimes hide. It's like Heisenbug.

Now I'm trying optimizing GCC:

```c
create_color:
    rep ret

main:
    xor eax, eax
    ; as if create_color() was called and returned 0
    sub r8d, DWORD PTR ds:8
    mov ecx, DWORD PTR [rax+4]
    mov edx, DWORD PTR [rax]
    mov esi, OFFSET FLAT:.LC1
    mov edi, 1
    call ___printf_chk
    xor eax, eax
    add rsp, 8
    ret
```

Compiler deducing that nothing returns from the function, so it optimizes it away. And it assumes, that is returns 0 by default. The zero is then used as an address to a structure in main().. Of course, this code crashes.

GCC is C++ mode silent about it as well.

Let's try non-optimizing MSVC 2015 x86. It warns about the problem:

```plaintext
c:\\tmp\\3.c(19) : warning C4716: 'create_color': must return a value
```

And generates crashing code:

```c
_create_color PROC
    push ebp
    mov ebp, esp
    push ecx
    push 12
    call _malloc
    ; EAX -> ptr to buffer
    add esp, 4
    mov DWORD PTR _rt$[ebp], eax
    mov eax, DWORD PTR _rt$[ebp]
    mov ecx, DWORD PTR _r$[ebp]
    mov DWORD PTR [eax], ecx
    mov edx, DWORD PTR _rt$[ebp]
    mov eax, DWORD PTR _G$[ebp]
    ; EAX is set to G argument:
    mov DWORD PTR [edx+4], eax
    mov ecx, DWORD PTR _rt$[ebp]
    mov edx, DWORD PTR _B$[ebp]
    mov DWORD PTR [ecx+8], edx
    mov esp, ebp
    pop ebp
    ; EAX = G at this point:
    ret 0
_create_color ENDP
```

Now optimizing MSVC 2015 x86 generates crashing code as well, but for the different reason:
Listing 3.136: Optimizing MSVC 2015 x86

main PROC
; this is inlined optimized version of create_color():
push ecx
push 12
call malloc
mov DWORD PTR [eax], 1
mov DWORD PTR [eax+4], 2
mov DWORD PTR [eax+8], 3
; EAX -> to allocated buffer, and it's filled, OK
; now we reload ptr to buffer, thinking it's in "a" variable
; but inlined function didn't store pointer to "a" variable!
mov eax, DWORD PTR a$[esp+8]
; EAX = some random garbage at this point
push DWORD PTR [eax+8]
push DWORD PTR [eax+4]
push DWORD PTR [eax]
push OFFSET $SG6074
call printf
xor eax, eax
add esp, 24
ret 0
main ENDP

_R$ = 8
_G$ = 12
_B$ = 16
_create_color PROC
push 12
call malloc
mov ecx, DWORD PTR _R$[esp]
add esp, 4
mov DWORD PTR [eax], ecx
mov ecx, DWORD PTR _G$[esp-4]
mov DWORD PTR [eax+4], ecx
mov ecx, DWORD PTR _B$[esp-4]
mov DWORD PTR [eax+8], ecx
; EAX -> to allocated buffer, OK
ret 0
_create_color ENDP

However, non-optimizing MSVC 2015 x64 generates working code:

Listing 3.137: Non-optimizing MSVC 2015 x64

rt$ = 32
_R$ = 64
_G$ = 72
_B$ = 80
_create_color PROC
mov DWORD PTR [rsp+24], r8d
mov DWORD PTR [rsp+16], edx
mov DWORD PTR [rsp+8], ecx
sub rsp, 56
mov ecx, 12
call malloc
; RAX = allocated buffer
mov QWORD PTR rt$[rsp], rax
mov rax, QWORD PTR rt$[rsp]
mov ecx, DWORD PTR _R$[esp]
mov DWORD PTR [rax], ecx
mov rax, QWORD PTR rt$[rsp]
mov ecx, DWORD PTR _G$[esp]
mov DWORD PTR [rax+4], ecx
mov rax, QWORD PTR rt$[rsp]
mov ecx, DWORD PTR _B$[esp]
mov DWORD PTR [rax+8], ecx
add esp, 56
; RAX didn’t change down to this point
Optimizing MSVC 2015 x64 also inlines the function, as in case of x86, and the resulting code also crashes.

This is a real piece of code from my octothorpe library\(^62\), that worked and all tests passed. It was so, without return for quite a time...

```c
uint32_t LPHM_u32_hash(void *key)
{
    jenkins_one_at_a_time_hash_u32((uint32_t)key);
}
```

The moral of the story: warnings are very important, use `-Wall`, etc, etc... When return statement is absent, compiler can just silently do nothing at that point.

Such a bug left unnoticed can ruin a day.

Also, *shotgun debugging* is bad, because again, such a bug can left unnoticed ("everything works now, so be it").

\section*{3.33 Homework: more about function pointers and unions}

This code was copypasted from \textit{dwm}\(^63\), probably, the smallest ever Linux window manager.

The problem: keystrokes from user must be dispatched to various functions inside of \textit{dwm}. This is usually solved using a big \textit{switch()}. Supposedly, \textit{dwm}'s creators wanted to make the code neat and modifiable by users:

```c
...  
typedef union {
    int i;
    unsigned int ui;
    float f;
    const void *v;
} Arg;
...

typedef struct {
    unsigned int mod;
    KeySym keysym;
    void (*func)(const Arg *);
    const Arg arg;
} Key;
...

static Key keys[] = {
    /* modifier    key    function     argument */
    { MODKEY,    XK_p,    spawn,     {.v = dmenucmd } },
    { MODKEY|ShiftMask,  XK_Return, spawn,    {.v = termcmd } },
    { MODKEY,    XK_b,    togglebar,  {0} },
    { MODKEY,    XK_j,    focusstack, {.i = +1 } },
    { MODKEY,    XK_k,    focusstack, {.i = -1 } },
    { MODKEY,    XK_i,    incnmaster, {.i = +1 } },
    { MODKEY,    XK_d,    incnmaster, {.i = -1 } },
    { MODKEY,    XK_h,    setmfact,   {.f = -0.05} },
    { MODKEY,    XK_l,    setmfact,   {.f = +0.05} },
    { MODKEY,    XK_Return, zoom,     {0} },
    { MODKEY,    XK_Tab,  view,       {0} },
    { MODKEY|ShiftMask,  XK_c,    killclient, {0} },

\end{verbatim}

\(^62\)https://github.com/DennisYurichev/octothorpe
\(^63\)https://dwm.suckless.org/
For each keystroke (or shortcut) a function is defined. Even more: a parameters (or arguments) to be passed to a function at each case. But parameters can have various type. So union is used here. A value of needed type is filled in the table. Each function takes what it needs.

As a homework, try to write a code like that, or get into dwm’s and see how union is passed into functions and handled.

### 3.34 Windows 16-bit

16-bit Windows programs are rare nowadays, but can be used in the cases of retrocomputing or dongle hacking ([8.8 on page 818](#)).

16-bit Windows versions were up to 3.11. 95/98/ME also support 16-bit code, as well as the 32-bit versions of the Windows NT line. The 64-bit versions of Windows NT line do not support 16-bit executable code at all.

The code resembles MS-DOS’s one.

Executable files are of type NE-type (so-called “new executable”).

All examples considered here were compiled by the OpenWatcom 1.9 compiler, using these switches:

```
wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows -bcl=windows example.c
```

### 3.34.1 Example#1

```c
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow )
{
    MessageBeep(MB_ICONEXCLAMATION);
    return 0;
};
```

```
WinMain proc near
    push    bp
    mov     bp, sp
    mov     ax, 30h ; '0' ; MB_ICONEXCLAMATION constant
    push    ax
    call    MESSAGEBEEP
    xor     ax, ax
          ; return 0
    pop     bp
    retn    0Ah
WinMain endp
```

Semms to be easy, so far.
### 3.34.2 Example #2

```c
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow )
{
    MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);
    return 0;
}
```

```asm
WinMain proc near
    push bp
    mov bp, sp
    xor ax, ax ; NULL
    push ax
    push ds
    mov ax, offset aHelloWorld ; 0x18. "hello, world"
    push ax
    push ds
    mov ax, offset aCaption ; 0x10. "caption"
    push ax
    mov ax, 3 ; MB_YESNOCANCEL
    push ax
    call MESSAGEBOX
    xor ax, ax ; return 0
    pop bp
    retn 0Ah
WinMain endp
```

Couple important things here: the PASCAL calling convention dictates passing the first argument first (MB_YESNOCANCEL), and the last argument—last (NULL). This convention also tells the callee to restore the stack pointer: hence the RETN instruction has 0Ah as argument, which implies that the pointer has to be increased by 10 bytes when the function exits. It is like stdcall (6.1.2 on page 725), but the arguments are passed in “natural” order.

The pointers are passed in pairs: first the data segment is passed, then the pointer inside the segment. There is only one segment in this example, so DS always points to the data segment of the executable.

### 3.34.3 Example #3

```c
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int nCmdShow )
{
    int result=MessageBox (NULL, "hello, world", "caption", MB_YESNOCANCEL);

    if (result==IDCANCEL)
        MessageBox (NULL, "you pressed cancel", "caption", MB_OK);
    else if (result==IDYES)
        MessageBox (NULL, "you pressed yes", "caption", MB_OK);
    else if (result==IDNO)
        MessageBox (NULL, "you pressed no", "caption", MB_OK);
    return 0;
}
```

```asm
WinMain proc near
    push bp
    mov bp, sp
```

644
Somewhat extended example from the previous section.

### 3.34.4 Example #4

```c
#include <windows.h>

int PASCAL func1 (int a, int b, int c)
{
    return a*b+c;
};

long PASCAL func2 (long a, long b, long c)
{
    return a*b+c;
};

long PASCAL func3 (long a, long b, long c, int d)
{
    return a*b+c-d;
};

int PASCAL WinMain( HINSTANCE hInstance,
```
HINSTANCE hPrevInstance,
LPSTR lpCmdLine,
int nCmdShow )
{
    func1 (123, 456, 789);
    func2 (600000, 700000, 800000);
    func3 (600000, 700000, 800000, 123);
    return 0;
};

func1 proc near
c = word ptr 4
b = word ptr 6
a = word ptr 8
push bp
mov bp, sp
mov ax, [bp+a]
imul [bp+b]
add ax, [bp+c]
pop bp
retn 6
func1 endp

func2 proc near
arg_0 = word ptr 4
arg_2 = word ptr 6
arg_4 = word ptr 8
arg_6 = word ptr 0Ah
arg_8 = word ptr 0Ch
arg_A = word ptr 0Eh
push bp
mov bp, sp
mov ax, [bp+arg_8]
mov dx, [bp+arg_A]
mov bx, [bp+arg_4]
mov cx, [bp+arg_6]
call sub_B2 ; long 32-bit multiplication
add ax, [bp+arg_0]
adc dx, [bp+arg_2]
pop bp
retn 12
func2 endp

func3 proc near
arg_0 = word ptr 4
arg_2 = word ptr 6
arg_4 = word ptr 8
arg_6 = word ptr 0Ah
arg_8 = word ptr 0Ch
arg_A = word ptr 0Eh
arg_C = word ptr 10h
push bp
mov bp, sp
mov ax, [bp+arg_A]
mov dx, [bp+arg_C]
mov bx, [bp+arg_6]
mov cx, [bp+arg_8]
call sub_B2 ; long 32-bit multiplication
mov cx, [bp+arg_2]
add cx, ax
mov bx, [bp+arg_4]
adc bx, dx ; BX=high part, CX=low part
mov ax, [bp+arg_0] ; AX=low part d, DX=high part d
32-bit values (the long data type implies 32 bits, while int is 16-bit) in 16-bit code (both MS-DOS and Win16) are passed in pairs. It is just like when 64-bit values are used in a 32-bit environment (1.34 on page 392).

sub_B2 here is a library function written by the compiler’s developers that does “long multiplication”, i.e., multiplies two 32-bit values. Other compiler functions that do the same are listed here: .5 on page 1029, .4 on page 1029.

The ADD/ADC instruction pair is used for addition of compound values: ADD may set/clear the CF flag, and ADC uses it after.

The SUB/SBB instruction pair is used for subtraction: SUB may set/clear the CF flag, SBB uses it after.

32-bit values are returned from functions in the DX:AX register pair.

Constants are also passed in pairs in WinMain() here.

The int-typed 123 constant is first converted according to its sign into a 32-bit value using the CWD instruction.
3.34.5 Example #5

```c
#include <windows.h>

int PASCAL string_compare (char *s1, char *s2)
{
    while (1)
    {
        if (*s1==*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    }
};

int PASCAL string_compare_far (char far *s1, char far *s2)
{
    while (1)
    {
        if (*s1==*s2)
            return 0;
        if (*s1==0 || *s2==0)
            return 1; // end of string
        s1++;
        s2++;
    }
};

void PASCAL remove_digits (char *s)
{
    while (*s)
    {
        if (*s>='0' && *s<='9')
            *s='-';
        s++;
    }
};

char str[]="hello 1234 world";

int PASCAL WinMain( HINSTANCE hInstance,
        HINSTANCE hPrevInstance,
        LPSTR lpCmdLine,
        int nCmdShow )
{
    string_compare ("asd", "def");
    string_compare_far ("asd", "def");
    remove_digits (str);
    MessageBox (NULL, str, "caption", MB_YESNOCANCEL);
    return 0;
};
```

```assembly
string_compare proc near
arg_0 = word ptr 4
arg_2 = word ptr 6

    push bp
    mov bp, sp
    push si
    mov si, [bp+arg_0]
    mov bx, [bp+arg_2]
loc_12: ; CODE XREF: string_compare+21j
    mov al, [bx]
    cmp al, [si]
```
jz short loc_1C
xor ax, ax
jmp short loc_2B

loc_1C: ; CODE XREF: string_compare+Ej
test al, al
jz short loc_22
jnz short loc_27

loc_22: ; CODE XREF: string_compare+16j
mov ax, 1
jmp short loc_2B

loc_27: ; CODE XREF: string_compare+18j
inc bx
inc si
jmp short loc_12

loc_2B: ; CODE XREF: string_compare+12j
; string_compare+1Dj
pop si
pop bp
retn 4
string_compare endp

string_compare_far proc near ; CODE XREF: WinMain+18p
arg_0 = word ptr 4
arg_2 = word ptr 6
arg_4 = word ptr 8
arg_6 = word ptr 0Ah
    push bp
    mov bp, sp
    push si
    mov si, [bp+arg_0]
    mov bx, [bp+arg_4]
loc_3A: ; CODE XREF: string_compare_far+35j
mov es, [bp+arg_6]
mov al, es:[bx]
mov es, [bp+arg_2]
cmp al, es:[si]
jz short loc_4C
xor ax, ax
jmp short loc_67

loc_4C: ; CODE XREF: string_compare_far+16j
mov es, [bp+arg_6]
cmp byte ptr es:[bx], 0
jz short loc_5E
mov es, [bp+arg_2]
cmp byte ptr es:[si], 0
jnz short loc_63

loc_5E: ; CODE XREF: string_compare_far+23j
mov ax, 1
jmp short loc_67

loc_63: ; CODE XREF: string_compare_far+2Cj
inc bx
inc si
jmp short loc_3A
loc_67: ; CODE XREF: string_compare_far+1Aj
        ; string_compare_far+31j
        pop    si
        pop    bp
        retn   8
string_compare_far endp

remove_digits    proc near ; CODE XREF: WinMain+1Fp
    arg_0 = word ptr 4
    push   bp
    mov    bp, sp
    mov    bx, [bp+arg_0]
loc_72: ; CODE XREF: remove_digits+18j
    mov    al, [bx]
    test   al, al
    jz     short loc_86
    cmp    al, 30h ; '0'
    jb     short loc_83
    cmp    al, 39h ; '9'
    ja     short loc_83
    mov    byte ptr [bx], 2Dh ; '-'
loc_83: ; CODE XREF: remove_digits+Ej
        ; remove_digits+12j
    inc    bx
    jmp    short loc_72
loc_86: ; CODE XREF: remove_digits+Aj
    pop    bp
    retn   2
remove_digits    endp

WinMain    proc near ; CODE XREF: start+EDp
    push   bp
    mov    bp, sp
    mov    ax, offset aAsd ; "asd"
    push   ax
    mov    ax, offset aDef ; "def"
    push   ax
    call   string_compare
    push   ds
    mov    ax, offset aAsd ; "asd"
    push   ax
    push   ds
    mov    ax, offset aDef ; "def"
    push   ax
    call   string_compare_far
    mov    ax, offset aHello1234World ; "hello 1234 world"
    push   ax
    call   remove_digits
    xor    ax, ax
    push   ax
    push   ds
    mov    ax, offset aHello1234World ; "hello 1234 world"
    push   ax
    push   ds
    mov    ax, offset aCaption ; "caption"
    push   ax
    mov    ax, 3 ; MB_YESNOCANCEL
    push   ax
    call   MESSAGEBOX
    xor    ax, ax
    pop    bp
    retn   0Ah
WinMain    endp
Here we see a difference between the so-called “near” pointers and the “far” pointers: another weird artifact of segmented memory in 16-bit 8086.

You can read more about it here: 11.6 on page 986.

“near” pointers are those which point within the current data segment. Hence, the string_compare() function takes only two 16-bit pointers, and accesses the data from the segment that DS points to (The mov al, [bx] instruction actually works like mov al, ds:[bx] — DS is implicit here).

“far” pointers are those which may point to data in another memory segment. Hence string_compare_far() takes the 16-bit pair as a pointer, loads the high part of it in the ES segment register and accesses the data through it (mov al, es:[bx]). “far” pointers are also used in my MessageBox() win16 example: 3.34.2 on page 644. Indeed, the Windows kernel is not aware which data segment to use when accessing text strings, so it need the complete information.

The reason for this distinction is that a compact program may use just one 64kb data segment, so it doesn’t need to pass the high part of the address, which is always the same. A bigger program may use several 64kb data segments, so it needs to specify the segment of the data each time.

It’s the same story for code segments. A compact program may have all executable code within one 64kb-segment, then all functions in it will be called using the CALL NEAR instruction, and the code flow will be returned using RETN. But if there are several code segments, then the address of the function is to be specified by a pair, it is to be called using the CALL FAR instruction, and the code flow is to be returned using RETF.

This is what is set in the compiler by specifying “memory model”.

The compilers targeting MS-DOS and Win16 have specific libraries for each memory model: they differ by pointer types for code and data.

### 3.34.6 Example #6

```c
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    struct tm *t;
    time_t unix_time;

    unix_time=time(NULL);
    t=localtime (&unix_time);

    sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, t->tm_mon, t->tm_mday,
             t->tm_hour, t->tm_min, t->tm_sec);

    MessageBox (NULL, strbuf, "caption", MB_OK);
    return 0;
}
```

<table>
<thead>
<tr>
<th>WinMain</th>
<th>proc near</th>
</tr>
</thead>
<tbody>
<tr>
<td>var_4</td>
<td>= word ptr -4</td>
</tr>
<tr>
<td>var_2</td>
<td>= word ptr -2</td>
</tr>
<tr>
<td></td>
<td>push bp</td>
</tr>
<tr>
<td></td>
<td>mov bp, sp</td>
</tr>
<tr>
<td></td>
<td>push ax</td>
</tr>
<tr>
<td></td>
<td>push ax</td>
</tr>
<tr>
<td></td>
<td>xor ax, ax</td>
</tr>
</tbody>
</table>

651
UNIX time is a 32-bit value, so it is returned in the DX:AX register pair and stored in two local 16-bit variables. Then a pointer to the pair is passed to the localtime() function. The localtime() function has a struct tm allocated somewhere in the guts of the C library, so only a pointer to it is returned.

By the way, this also implies that the function cannot be called again until its results are used.

For the time() and localtime() functions, a Watcom calling convention is used here: the first four arguments are passed in the AX, DX, BX and CX, registers, and the rest arguments are via the stack.

The functions using this convention are also marked by underscore at the end of their name.

sprintf() does not use the PASCAL calling convention, nor the Watcom one, so the arguments are passed in the normal cdecl way (6.1.1 on page 725).

**Global variables**

This is the same example, but now these variables are global:
t=localtime (&unix_time);

sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->tm_year+1900, t->tm_mon, t->
\t->tm_mday, 
\t->tm_hour, t->tm_min, t->tm_sec);

MessageBox (NULL, strbuf, "caption", MB_OK);
return 0;

unix_time_low dw 0
unix_time_high dw 0
t dw 0

WinMain proc near
push bp
mov bp, sp
xor ax, ax
call time_
mov unix_time_low, ax
mov unix_time_high, dx
mov ax, offset unix_time_low

call localtime_
mov bx, ax
mov t, ax ; will not be used in future...
push word ptr [bx]; seconds
push word ptr [bx+2]; minutes
push word ptr [bx+4]; hour
push word ptr [bx+6]; day
push word ptr [bx+8]; month
mov ax, [bx+0Ah]; year
add ax, 1900
push ax
mov ax, offset a04d02d02d02d02 ; "%04d-%02d-%02d %02d:%02d:%02d"
push ax
mov ax, offset strbuf
push ax

call sprintf_
add sp, 10h
xor ax, ax ; NULL
push ax
push ds
mov ax, offset strbuf
push ax
push ds
mov ax, offset aCaption ; "caption"
push ax
xor ax, ax ; MB_OK
push ax

call MESSAGEBOX
xor ax, ax ; return 0
pop bp
ret
WinMain endp

653

t is not to be used, but the compiler emitted the code which stores the value.
Because it is not sure, maybe that value will eventually be used in some other module.
Chapter 4

Java

4.1 Java

4.1.1 Introduction

There are some well-known decompilers for Java (or JVM bytecode in general)\(^1\).

The reason is the decompilation of JVM-bytecode is somewhat easier than for lower level x86 code:

- There is much more information about the data types.
- The JVM memory model is much more rigorous and outlined.
- The Java compiler don’t do any optimizations (the JVM JIT\(^2\) does them at runtime), so the bytecode in the class files is usually pretty readable.

When can the knowledge of JVM be useful?

- Quick-and-dirty patching tasks of class files without the need to recompile the decompiler’s results.
- Analyzing obfuscated code.
- Analyzing of a code generated by newer Java compiler for which no updated decompiler exists yet.
- Building your own obfuscator.
- Building a compiler codegenerator (back-end) targeting JVM (like Scala, Clojure, etc. \(^3\)).

Let’s start with some simple pieces of code. JDK 1.7 is used everywhere, unless mentioned otherwise.

This is the command used to decompile class files everywhere: javap -c -verbose.

This is the book I used while preparing all examples: [Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, The Java(R) Virtual Machine Specification / Java SE 7 Edition]\(^4\).

4.1.2 Returning a value

Probably the simplest Java function is the one which returns some value.

Oh, and we must keep in mind that there are no “free” functions in Java in common sense, they are “methods”.

Each method is related to some class, so it’s not possible to define a method outside of a class. But we’ll call them “functions” anyway, for simplicity.

```java
public class ret {
    public static int main(String[] args) {
        return 0;
    }
}
```

---

\(^1\)For example, JAD: [http://varaneckas.com/jad/](http://varaneckas.com/jad/)

\(^2\)Just-In-Time compilation


\(^4\)Also available as [https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf](https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf) [http://docs.oracle.com/javase/specs/jvms/se7/html/](http://docs.oracle.com/javase/specs/jvms/se7/html/)
Let’s compile it:

```
javac ret.java
```

...and decompile it using the standard Java utility:

```
javap -c -verbose ret.class
```

And we get:

**Listing 4.1: JDK 1.7 (excerpt)**

```java
public static int main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=1, locals=1, args_size=1
  0: iconst_0
  1: ireturn
```

The Java developers decided that 0 is one of the busiest constants in programming, so there is a separate short one-byte `iconst_0` instruction which pushes 0.

There are also `iconst_1` (which pushes 1), `iconst_2`, etc., up to `iconst_5`.

There is also `iconst_m1` which pushes -1.

The stack is used in JVM for passing data to called functions and also for return values. So `iconst_0` pushes 0 into the stack. `ireturn` returns an integer value (i in name means integer) from the TOS.

Let’s rewrite our example slightly, now we return 1234:

```java
public class ret {
    public static int main(String[] args) {
        return 1234;
    }
}
```

...we get:

**Listing 4.2: JDK 1.7 (excerpt)**

```java
public static int main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=1, locals=1, args_size=1
  0: sipush 1234
  3: ireturn
```

`sipush (short integer)` pushes 1234 into the stack. short in name implies a 16-bit value is to be pushed. The number 1234 indeed fits well in a 16-bit value.

What about larger values?

```java
public class ret {
    public static int main(String[] args) {
        return 12345678;
    }
}
```

5 Just like in MIPS, where a separate register for zero constant exists: 1.5.4 on page 25.

6 Top of Stack
It’s not possible to encode a 32-bit number in a JVM instruction opcode, the developers didn’t leave such possibility.

So the 32-bit number 12345678 is stored in so called “constant pool” which is, let’s say, the library of most used constants (including strings, objects, etc.).

This way of passing constants is not unique to JVM.

MIPS, ARM and other RISC CPUs also can’t encode a 32-bit number in a 32-bit opcode, so the RISC CPU code (including MIPS and ARM) has to construct the value in several steps, or to keep it in the data segment: 1.39.3 on page 438, 1.40.1 on page 441.

MIPS code also traditionally has a constant pool, named “literal pool”, the segments are called “.lit4” (for 32-bit single precision floating point number constants) and “.lit8” (for 64-bit double precision floating point number constants).

Let’s try some other data types!

Boolean:

```java
public class ret {
    public static boolean main(String[] args) {
        return true;
    }
}
```

This JVM bytecode is no different from one returning integer 1.

32-bit data slots in the stack are also used here for boolean values, like in C/C++.

But one could not use returned boolean value as integer or vice versa — type information is stored in the class file and checked at runtime.

It’s the same story with a 16-bit short:

```java
public class ret {
    public static short main(String[] args) {
        return 1234;
    }
}
```
...and char!

```java
public class ret {
    public static char main(String[] args) {
        return 'A';
    }
}
```

```java
public static char main(java.lang.String[]); flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
    0: bipush 65
    2: ireturn
```

*bipush means “push byte”.* Needless to say that a *char* in Java is 16-bit UTF-16 character, and it's equivalent to *short*, but the ASCII code of the “A” character is 65, and it's possible to use the instruction for pushing a byte in the stack.

Let's also try a *byte*:

```java
public class retc {
    public static byte main(String[] args) {
        return 123;
    }
}
```

```java
public static byte main(java.lang.String[]); flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
    0: bipush 123
    2: ireturn
```

One may ask, why bother with a 16-bit *short* data type which internally works as a 32-bit integer?

Why use a *char* data type if it is the same as a *short* data type?

The answer is simple: for data type control and source code readability.

*A char* may essentially be the same as a *short*, but we quickly grasp that it's a placeholder for an UTF-16 character, and not for some other integer value.

When using *short*, we show everyone that the variable’s range is limited by 16 bits.

It's a very good idea to use the *boolean* type where needed to, instead of the C-style *int*.

There is also a 64-bit integer data type in Java:

```java
public class ret3 {
    public static long main(String[] args) {
        return 1234567890123456789L;
    }
}
```

```java
Listing 4.4: Constant pool
...
    #2 = Long 1234567890123456789L
...
```

```java
public static long main(java.lang.String[]); flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
```

657
The 64-bit number is also stored in a constant pool, `ldc2_w` loads it and `lreturn` (long return) returns it. The `ldc2_w` instruction is also used to load double precision floating point numbers (which also occupy 64 bits) from a constant pool:

```java
public class ret {
    public static double main(String[] args) {
        return 123.456d;
    }
}
```

Listing 4.5: Constant pool

```
... #2 = Double 123.456d ...
```

`dreturn` stands for “return double”.

And finally, a single precision floating point number:

```java
public class ret {
    public static float main(String[] args) {
        return 123.456f;
    }
}
```

Listing 4.6: Constant pool

```
... #2 = Float 123.456f ...
```

The `ldc` instruction used here is the same one as for loading 32-bit integer numbers from a constant pool. `freturn` stands for “return float”.

Now what about function that return nothing?

```java
public class ret {
    public static void main(String[] args) {
        return;
    }
}
```
This means that the return instruction is used to return control without returning an actual value. Knowing all this, it's very easy to deduce the function's (or method's) returning type from the last instruction.

### 4.1.3 Simple calculating functions

Let's continue with a simple calculating functions.

```java
public class calc {
    public static int half(int a) {
        return a/2;
    }
}
```

Here's the output when the `iconst_2` instruction is used:

```java
public static int half(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
    0: iload_0
    1: iconst_2
    2: idiv
    3: ireturn
```

`iload_0` takes the zeroth function argument and pushes it to the stack.

`iconst_2` pushes 2 in the stack. After the execution of these two instructions, this is how stack looks like:

```
+---+
TOS ->| 2 |
+---+
| a |
+---+
```

`idiv` just takes the two values at the TOS, divides one by the other and leaves the result at TOS:

```
+--------+
TOS ->| result |
+--------+
```

`ireturn` takes it and returns.

Let's proceed with double precision floating point numbers:

```java
public class calc {
    public static double half_double(double a) {
        return a/2.0;
    }
}
```

Listing 4.7: Constant pool

```java
...  #2 = Double 2.0d
...`
public static double half_double(double);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=4, locals=2, args_size=1
  0: dload_0
  1: ldc2_w #2 // double 2.0d
  4: ddiv
  5: dreturn

It's the same, but the ldc2_w instruction is used to load the constant 2.0 from the constant pool.
Also, the other three instructions have the d prefix, meaning they work with double data type values.

Let's now use a function with two arguments:

public class calc
{
    public static int sum(int a, int b)
    {
        return a+b;
    }
}

public static int sum(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=2, args_size=2
  0: iload_0
  1: iload_1
  2: iadd
  3: ireturn

iload_0 loads the first function argument (a), iload_1—second (b).

Here is the stack after the execution of both instructions:

+---+
TOS ->| b |
| a | +---+
iadd adds the two values and leaves the result at TOS:

+--------+
TOS ->| result |
| +--------+

Let's extend this example to the long data type:

public static long lsum(long a, long b)
{
    return a+b;
}

...we got:

public static long lsum(long, long);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=4, locals=4, args_size=2
  0: lload_0
  1: lload_2
  2: ladd
  3: lreturn

The second lload instruction takes the second argument from the 2nd slot.
That's because a 64-bit long value occupies exactly two 32-bit slots.

Slightly more advanced example:
public class calc
{
    public static int mult_add(int a, int b, int c)
    {
        return a*b+c;
    }
}

public static int mult_add(int, int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=3, args_size=3
0: iload_0
1: iload_1
2: imul
3: iload_2
4: iadd
5: ireturn

The first step is multiplication. The product is left at the TOS:

+----------+
TOS ->| product |
+----------+

iload_2 loads the third argument (c) in the stack:

+----------+
TOS ->| c |
| product |
+----------+

Now the iadd instruction can add the two values.

### 4.1.4 JVM memory model

x86 and other low-level environments use the stack for argument passing and as a local variables storage. JVM is slightly different.

It has:

- Local variable array (LVA\(^7\)). Used as storage for incoming function arguments and local variables. Instructions like iload_0 load values from it.
  istore stores values in it. At the beginning the function arguments are stored: starting at 0 or at 1 (if the zeroth argument is occupied by this pointer).

Then the local variables are allocated.

Each slot has size of 32-bit.

Hence, values of long and double data types occupy two slots.

- Operand stack (or just “stack”). It’s used for computations and passing arguments while calling other functions.
  Unlike low-level environments like x86, it’s not possible to access the stack without using instructions which explicitly pushes or pops values to/from it.

- Heap. It is used as storage for objects and arrays.

These 3 areas are isolated from each other.

\(^7\)(Java) Local Variable Array
4.1.5 Simple function calling

Math.random() returns a pseudorandom number in range of [0.0 ... 1.0], but let’s say that for some reason we need to devise a function that returns a number in range of [0.0 ... 0.5]:

```java
public class HalfRandom {
    public static double f() {
        return Math.random() / 2;
    }
}
```

Listing 4.8: Constant pool

```
... 
#2 = Methodref #18.#19 // java/lang/Math.random:()D
#3 = Double 2.0d
...
#12 = Utf8 ()D
...
#18 = Class #22 // java/lang/Math
#19 = NameAndType #23:#12 // random:()D
#22 = Utf8 java/lang/Math
#23 = Utf8 random
```

invokestatic calls the Math.random() function and leaves the result at the TOS. Then the result is divided by 2.0 and returned.

But how is the function name encoded? It’s encoded in the constant pool using a Methodref expression. It defines the class and method names.

The first field of Methodref points to a Class expression which, in turn, points to the usual text string ("java/lang/Math"). The second Methodref expression points to a NameAndType expression which also has two links to the strings.

The first string is “random”, which is the name of the method. The second string is “()D”, which encodes the function’s type. It means that it returns a double value (hence the D in the string).

This is the way 1) JVM can check data for type correctness; 2) Java decompilers can restore data types from a compiled class file.

Now let’s try the “Hello, world!” example:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

Listing 4.9: Constant pool

```
... 
#2 = Fieldref #16.#17 // java/lang/System.out:Ljava/io/PrintStream;
```
ldc at offset 3 takes a pointer to the “Hello, World” string in the constant pool and pushes in the stack. It’s called a reference in the Java world, but it’s rather a pointer, or an address.

The familiar invokevirtual instruction takes the information about the println function (or method) from the constant pool and calls it.

As we may know, there are several println methods, one for each data type. Our case is the version of println intended for the String data type.

But what about the first getstatic instruction?

This instruction takes a reference (or address of) a field of the object System.out and pushes it in the stack.

This value is acts like the this pointer for the println method.

Thus, internally, the println method takes two arguments for input: 1) this, i.e., a pointer to an object; 2) the address of the “Hello, World” string.

Indeed, println() is called as a method within an initialized System.out object.

For convenience, the javap utility writes all this information in the comments.

4.1.6 Calling beep()

This is a simple calling of two functions without arguments:

```
public static void main(String[] args)
{
    java.awt.Toolkit.getDefaultToolkit().beep();
};
```

```
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=1, locals=1, args_size=1
    0: invokestatic   #2 // Method java/awt/Toolkit.getDefaultToolkit():Ljava/awt/Toolkit;
    3: invokevirtual #3 // Method java/awt/Toolkit.beep:()V
    6: return
```
First `invokestatic` at offset 0 calls `java.awt.Toolkit.getDefaultToolkit()`, which returns a reference to an object of class `Toolkit`. The `invokevirtual` instruction at offset 3 calls the `beep()` method of this class.

### 4.1.7 Linear congruential PRNG

Let’s try a simple pseudorandom numbers generator, which we already considered once in the book (1.29 on page 336):

```java
public class LCG {
    public static int rand_state;
    public void my_srand (int init) {
        rand_state=init;
    }
    public static int RNG_a=1664525;
    public static int RNG_c=1013904223;
    public int my_rand () {
        rand_state=rand_state*RNG_a;
        rand_state=rand_state+RNG_c;
        return rand_state & 0x7fff;
    }
}
```

There are couple of class fields which are initialized at start. But how? In javap output we can find the class constructor:

```java
static {};
flags: ACC_STATIC
Code:
    stack=1, locals=0, args_size=0
0: ldc #5 // int 1664525
2: putstatic #3 // Field RNG_a:I
5: ldc #6 // int 1013904223
7: putstatic #4 // Field RNG_c:I
10: return
```

That’s the way variables are initialized. `RNG_a` occupies the 3rd slot in the class and `RNG_c`—4th, and `putstatic` puts the constants there.

The `my_srand()` function just stores the input value in `rand_state`:

```java
public void my_srand(int);
flags: ACC_PUBLIC
Code:
    stack=1, locals=2, args_size=2
664
```

`iload_1` takes the input value and pushes it into stack. But why not `iload_0`?

It’s because this function may use fields of the class, and so `this` is also passed to the function as a zeroth argument.

The field `rand_state` occupies the 2nd slot in the class, so `putstatic` copies the value from the TOS into the 2nd slot.

Now `my_rand()`:

```java
public int my_rand();
flags: ACC_PUBLIC
Code:
    stack=2, locals=1, args_size=1
```
It just loads all the values from the object's fields, does the operations and updates rand_state's value using the putstatic instruction.

At offset 20, rand_state is reloaded again (because it has been dropped from the stack before, by putstatic).

This looks like non-efficient code, but be sure, the JVM is usually good enough to optimize such things really well.

### 4.1.8 Conditional jumps

Now let's proceed to conditional jumps.

```java
public class abs {
    public static int abs(int a) {
        if (a<0)
            return -a;
        return a;
    }
}
```

```java
public static int abs(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=1, locals=1, args_size=1
0: iload_0
1: ifge 7
4: iload_0
5: ineg
6: ireturn
7: iload_0
8: ireturn
```

ifge jumps to offset 7 if the value at TOS is greater or equal to 0.

Don't forget, any ifXX instruction pops the value (to be compared) from the stack.

ineg just negates value at TOS.

Another example:

```java
public static int min (int a, int b) {
    if (a>b)
        return b;
    return a;
}
```

We get:

```java
public static int min(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
0: iload_0
```
if_icmple pops two values and compares them. If the second one is lesser than (or equal to) the first, a jump to offset 7 is performed.

When we define max() function ...

```java
public static int max (int a, int b) {
    if (a>b)
        return a;
    return b;
}
```

...the resulting code is the same, but the last two iload instructions (at offsets 5 and 7) are swapped:

```java
public static int max(int, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
0: iload_0
1: iload_1
2: if_icmple 7
5: iload_0
6: ireturn
7: iload_1
8: ireturn
```

A more advanced example:

```java
public class cond {
    public static void f(int i) {
        if (i<100)
            System.out.print("<100");
        if (i==100)
            System.out.print("==100");
        if (i>100)
            System.out.print(">100");
        if (i==0)
            System.out.print("==0");
    }
}
```

```java
public static void f(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=1, args_size=1
0: iload_0
1: bipush 100
3: if_icmpge 14
6: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
9: ldc #3 // String <100
11: invokevirtual #4 // Method java/io/PrintStream.print:(Ljava/lang/String;)V
14: iload_0
15: bipush 100
17: if_icmpne 28
20: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
23: ldc #5 // String ==100
25: invokevirtual #4 // Method java/io/PrintStream.print:(Ljava/lang/String;)V
28: iload_0
29: bipush 100
31: if_icmpne 42
34: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
```
if `icmpge` pops two values and compares them. If the second one is larger or equal than the first, a jump to offset 14 is performed.

if `icmpne` and if `icmple` work just the same, but implement different conditions.

There is also a `ifne` instruction at offset 43.

Its name is misnomer, it would’ve be better to name it `ifnz` (jump if the value at TOS is not zero).

And that is what it does: it jumps to offset 54 if the input value is not zero.

If zero, the execution flow proceeds to offset 46, where the “==0” string is printed.

N.B.: JVM has no unsigned data types, so the comparison instructions operate only on signed integer values.

4.1.9 Passing arguments

Let’s extend our `min()`/`max()` example:

```java
public class minmax
{
    public static int min (int a, int b)
    {
        if (a>b)
            return b;
        return a;
    }

    public static int max (int a, int b)
    {
        if (a>b)
            return a;
        return b;
    }

    public static void main(String[] args)
    {
        int a=123, b=456;
        int max_value=max(a, b);
        int min_value=min(a, b);
        System.out.println(min_value);
        System.out.println(max_value);
    }
}
```

Here is `main()` function code:

```java
public static void main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=5, args_size=1
    0: bipush    123
    2: istore_1
    3: sipush    456
    6: istore_2
    7: iload_1
    8: iload_2
    9: invokestatic #2   // Method max:(II)I
   12: istore_3
   13: iload_1
   14: iload_2
```
15: invokestatic #3 // Method min:(II)I
18: istore 4
20: getstatic #4 // Field java/lang/System.out:Ljava/io/PrintStream;
23: iload 4
25: invokevirtual #5 // Method java/io/PrintStream.println:(I)V
28: getstatic #4 // Field java/lang/System.out:Ljava/io/PrintStream;
31: iload_3
32: invokevirtual #5 // Method java/io/PrintStream.println:(I)V
35: return

Arguments are passed to the other function in the stack, and the return value is left on TOS.

4.1.10 Bitfields

All bit-wise operations work just like in any other ISA:

```java
public static int set (int a, int b) {
    return a | 1<<b;
}

public static int clear (int a, int b) {
    return a & (~(1<<b));
}

public static long lset(long a, int b) {
    return a | 1L<<b;
}

public static long lclear (long a, int b) {
    return a & (~(1L<<b));
}
```

iconst_m1 loads −1 in the stack, it’s the same as the 0xFFFFFFFF number.
XORing with 0xFFFFFFFF has the same effect of inverting all bits ( 2.6 on page 456).

Let’s extend all data types to 64-bit long:

```java
public static long lset(long a, int b) {
    return a | 1L<<b;
}

public static long lclear (long a, int b) {
    return a & (~(1L<<b));
}
```

public static long lset(long, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
```java
public static long lclear(long, int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=4, locals=3, args_size=2
0: lload_0
1: iconst_1
2: iload_2
3: ishl
4: i2l
5: lor
6: lreturn
```

The code is the same, but instructions with `l` prefix are used, which operate on 64-bit values.

Also, the second argument type of the function is still `int`, and when the 32-bit value in it needs to be promoted to 64-bit value the `i2l` instruction is used, which essentially extend the value of an `integer` type to a `long` one.

### 4.1.11 Loops

```java
public class Loop {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
        }
    }
}
```

```java
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=1
0: iconst_1
1: istore_1
2: iload_1
3: bipush 10
5: if_icmpgt 21 // Field java/lang/System.out:Ljava/io/PrintStream;
8: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
11: iload_1
12: invokevirtual #3 // Method java/io/PrintStream.println:(I)V
15: iinc 1, 1
18: goto 2
21: return
```

`iconst_1` loads 1 into TOS, `istore_1` stores it in the LVA at slot 1.

Why not the zeroth slot? Because the `main()` function has one argument (array of `String`) and a pointer to it (or reference) is now in the zeroth slot.

So, the `i` local variable will always be in 1st slot.

Instructions at offsets 3 and 5 compare `i` with 10.

If `i` is larger, execution flow passes to offset 21, where the function ends.

If it's not, `println` is called.
$i$ is then reloaded at offset 11, for `println`.

By the way, we call the `println` method for an integer, and we see this in the comments: “(I)V” (I means integer and V means the return type is void).

When `println` finishes, $i$ is incremented at offset 15.

The first operand of the instruction is the number of a slot (1), the second is the number (1) to add to the variable.

goto is just GOTO, it jumps to the beginning of the loop’s body offset 2.

Let’s proceed with a more complex example:

```java
public class Fibonacci {
    public static void main(String[] args) {
        int limit = 20, f = 0, g = 1;
        for (int i = 1; i <= limit; i++) {
            f = f + g;
            g = f - g;
            System.out.println(f);
        }
    }
}
```

public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=5, args_size=1
0: bipush 20
2: istore_1
3: iconst_0
4: istore_2
5: iconst_1
6: istore_3
7: iconst_1
8: istore 4
10: iload 4
12: iload_1
13: if_icmpgt 37
16: iload_2
17: iload_3
18: iadd
19: istore_2
20: iload_2
21: iload_3
22: isub
23: istore_3
24: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
27: iload_2
28: invokevirtual #3 // Method java/io/PrintStream.println:(I)V
31: iinc 4, 1
34: goto 10
37: return
```

Here is a map of the LVA slots:

- 0 — the sole argument of `main`
- 1 — `limit`, always contains 20
- 2 — $f$
- 3 — $g$
- 4 — $i$

We can see that the Java compiler allocates variables in LVA slots in the same order they were declared in the source code.
There are separate istore instructions for accessing slots 0, 1, 2 and 3, but not for 4 and larger, so there is istore with an additional operand at offset 8 which takes the slot number as an operand.

It's the same with iload at offset 10.

But isn't it dubious to allocate another slot for the limit variable, which always contains 20 (so it's a constant in essence), and reload its value so often?

JVM JIT compiler is usually good enough to optimize such things.

Manual intervention in the code is probably not worth it.

4.1.12 switch()

The switch() statement is implemented with the tablesswitch instruction:

```
public static void f(int a)
{
    switch (a)
    {
    case 0: System.out.println("zero"); break;
    case 1: System.out.println("one\n"); break;
    case 2: System.out.println("two\n"); break;
    case 3: System.out.println("three\n"); break;
    case 4: System.out.println("four\n"); break;
    default: System.out.println("something unknown\n"); break;
    }
}
```

As simple, as possible:

```
public static void f(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
   0: iload_0
   1: tableswitch { // 0 to 4
       0: 36
       1: 47
       2: 58
       3: 69
       4: 80
       default: 91
   }
   36: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
   39: ldc #3 // String zero
   41: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   44: goto 99
   47: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
   50: ldc #5 // String one\n
   52: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   55: goto 99
   58: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
   61: ldc #6 // String two\n
   63: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   66: goto 99
   69: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
   72: ldc #7 // String three\n
   74: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   77: goto 99
   80: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
   83: ldc #8 // String four\n
   85: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   88: goto 99
   91: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
   94: ldc #9 // String something unknown\n
   96: invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
   99: return
```

671
4.1.13 Arrays

Simple example

Let's first create an array of 10 integers and fill it:

```java
public static void main(String[] args) {
    int a[] = new int[10];
    for (int i=0; i<10; i++)
        a[i] = i;
    dump(a);
}
```

The ```newarray``` instruction creates an array object of 10 `int` elements.
The array's size is set with ```bipush``` and left at TOS.
The array's type is set in ```newarray``` instruction's operand.
After ```newarray```'s execution, a reference (or pointer) to the newly created array in the heap is left at the TOS.
```astore_1``` stores the reference to the 1st slot in LVA.
The second part of the ```main()``` function is the loop which stores ```i``` into the corresponding array element.
```aload_1``` gets a reference of the array and places it in the stack.
```iastore``` then stores the integer value from the stack in the array, reference of which is currently in TOS.
The third part of the ```main()``` function calls the ```dump()``` function.
An argument for it is prepared by ```aload_1``` (offset 23).
Now let's proceed to the ```dump()``` function:

```java
public static void dump(int a[]) {
    for (int i=0; i<a.length; i++)
        System.out.println(a[i]);
}
```

```java
public static void main(java.lang.String[]); flags: ACC_PUBLIC, ACC_STATIC
Code:
  stack=3, locals=3, args_size=1
  0: bipush 10
  2: newarray int
  4: astore_1
  5: iconst_0
  6: istore_2
  7: iload_2
  8: bipush 10
 10: if_icmpge 23
13: aload_1
14: iload_2
15: iload_2
16: iastore
17: iinc 2, 1
20: goto 7
23: aload_1
24: invokestatic #4 // Method dump:([I)V
27: return
```
The incoming reference to the array is in the zeroth slot.

The a.length expression in the source code is converted to an arraylength instruction: it takes a reference to the array and leaves the array size at TOS.

iaload at offset 13 is used to load array elements, it requires an array reference to be present in the stack (prepared by aload_0 at 11), and also an index (prepared by iload_1 at offset 12).

Needless to say, instructions prefixed with a may be mistakenly comprehended as array instructions.

It’s not correct. These instructions works with references to objects.

And arrays and strings are objects too.

Summing elements of array

Another example:

```java
public class ArraySum {

    public static int f (int[] a) {
        int sum=0;
        for (int i=0; i<a.length; i++)
            sum=sum+a[i];
        return sum;
    }
}
```

The only argument of the main() function is an array too

We’ll be using the only argument of the main() function, which is an array of strings:
public class UseArgument {
    public static void main(String[] args) {
        System.out.print("Hi, ");
        System.out.print(args[1]);
        System.out.println(" How are you?");
    }
}

The zeroth argument is the program's name (like in C/C++, etc.), so the 1st argument supplied by the user is 1st.

public static void main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
stack=3, locals=1, args_size=1  
0: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc #3 // String Hi,
5: invokevirtual #4 // Method java/io/PrintStream.print:(Ljava/lang/String;)V
8: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
11: aload_0
12: iconst_1
13: aaload
14: invokevirtual #4 // Method java/io/PrintStream.print:(Ljava/lang/String;)V
17: getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
20: ldc #5 // String . How are you?
22: invokevirtual #6 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
25: return

aload_0 at 11 loads a reference of the zeroth LVA slot (1st and only main() argument).  
iconst_1 and aaload at 12 and 13 take a reference to the first (counting at 0) element of array.  
The reference to the string object is at TOS at offset 14, and it is taken from there by println method.

Pre-initialized array of strings

class Month {
    public static String[] months = {
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
    };

    public String get_month (int i) {
        return months[i];
    }
}

The get_month() function is simple:

public java.lang.String get_month(int);  
flags: ACC_PUBLIC  
Code:  
stack=2, locals=2, args_size=2  
0: getstatic #2 // Field months:[Ljava/lang/String;
aaload operates on an array of references. Java String are objects, so the a-instructions are used to operate on them. areturn returns a reference to a String object.

How is the months[] array initialized?

```java
static {};
flags: ACC_STATIC
Code:
stack=4, locals=0, args_size=0
0: bipush 12
2: anewarray #3 // class java/lang/String
5: dup
6: iconst_0
7: ldc #4 // String January
9: aastore
10: dup
11: iconst_1
12: ldc #5 // String February
14: aastore
15: dup
16: iconst_2
17: ldc #6 // String March
19: aastore
20: dup
21: iconst_3
22: ldc #7 // String April
24: aastore
25: dup
26: iconst_4
27: ldc #8 // String May
29: aastore
30: dup
31: iconst_5
32: ldc #9 // String June
34: aastore
35: dup
36: bipush 6
38: ldc #10 // String July
40: aastore
41: dup
42: bipush 7
44: ldc #11 // String August
46: aastore
47: dup
48: bipush 8
50: ldc #12 // String September
52: aastore
53: dup
54: bipush 9
56: ldc #13 // String October
58: aastore
59: dup
60: bipush 10
62: ldc #14 // String November
64: aastore
65: dup
66: bipush 11
68: ldc #15 // String December
70: aastore
71: putstatic #2 // Field months:[Ljava/lang/String;
74: return
```

anewarray creates a new array of references (hence a prefix).
The object’s type is defined in the anewarray’s operand, it is the “java/lang/String” string.

The bipush 12 before anewarray sets the array’s size.

We see here a new instruction for us: dup.

It’s a standard instruction in stack computers (including the Forth programming language) which just duplicates the value at TOS.

By the way, FPU 80x87 is also a stack computer and it has similar instruction – FDUP.

It is used here to duplicate a reference to an array, because the aastore instruction pops the reference to array from the stack, but subsequent aastore will need it again.

The Java compiler concluded that it’s better to generate a dup instead of generating a getstatic instruction before each array store operation (i.e., 11 times).

aastore puts a reference (to string) into the array at an index which is taken from TOS.

Finally, putstatic puts reference to the newly created array into the second field of our object, i.e., months field.

Variadic functions

Variadic functions actually use arrays:

```java
public static void f(int... values)
{
    for (int i=0; i<values.length; i++)
        System.out.println(values[i]);
}

public static void main(String[] args)
{
    f(1,2,3,4,5);
}
```

```java
public static void f(int...);
flags: ACC_PUBLIC, ACC_STATIC, ACC_VARARGS
Code:
    stack=3, locals=2, args_size=1
      0: iconst_0
      1: istore_1
      2: iload_1
      3:aload_0
      4:arraylength
      5:if_icmpge 23
      8:getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
      11:aload_0
      12:iload_1
      13:iaload
      14:invokevirtual #3 // Method java/io/PrintStream.println:(I)V
      17:iinc 1, 1
      20:goto 2
      23:return
```

f() just takes an array of integers using aload_0 at offset 3.

Then it gets the array’s size, etc.

```java
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=4, locals=1, args_size=1
      0:iconst_5
      1:newarray int
      3:dup
      4:iconst_0
      5:iconst_1
      6:iastore
      7:dup
```

f() just takes an array of integers using aload_0 at offset 3.

Then it gets the array’s size, etc.
The array is constructed in main() using the newarray instruction, then it's filled, and f() is called.

Oh, by the way, array object is not destroyed at the end of main().

There are no destructors in Java at all, because the JVM has a garbage collector which does this automatically, when it feels it needs to.

What about the format() method?

It takes two arguments at input: a string and an array of objects:

```java
public PrintStream format(String format, Object... args)
```

(http://docs.oracle.com/javase/tutorial/java/data/numberformat.html)

Let's see:

```java
public static void main(String[] args)
{
    int i=123;
    double d=123.456;
    System.out.format("int: %d double: %f.%n", i, d);
}
```

So values of the `int` and `double` types are first promoted to `Integer` and `Double` objects using the `valueOf` methods.
The format() method needs objects of type Object at input, and since the Integer and Double classes are derived from the root Object class, they are suitable for elements in the input array.

On the other hand, an array is always homogeneous, i.e., it can't hold elements of different types, which makes it impossible to push int and double values in it.

An array of Object objects is created at offset 13, an Integer object is added to the array at offset 22, and a Double object is added to the array at offset 29.

The penultimate pop instruction discards the element at TOS, so when return is executed, the stack becomes empty (or balanced).

**Two-dimensional arrays**

Two-dimensional arrays in Java are just one-dimensional arrays of references to another one-dimensional arrays.

Let's create a two-dimensional array:

```java
public static void main(String[] args) {
    int[][] a = new int[5][10];
    a[1][2]=3;
}
```

```java
public static void main(java.lang.String[]);  
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=3, locals=2, args_size=1
    0: iconst_5
    1: bipush 10
    3: multianewarray #2, 2 // class "[[I"
    7: astore_1
    8: aload_1
    9: iconst_1
10: aaload
11: iconst_2
12: aaload
13: iastore
14: return
```

It's created using the multianewarray instruction: the object's type and dimensionality are passed as operands.

The array's size (10*5) is left in stack (using the instructions iconst_5 and bipush).

A reference to row #1 is loaded at offset 10 (iconst_1 and aaload).

The column is chosen using iconst_2 at offset 11.

The value to be written is set at offset 12.

iastore at 13 writes the array's element.

How it is an element accessed?

```java
public static int get12 (int[][] in) {
    return in[1][2];
}
```

```java
public static int get12(int[][]);  
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
    0: aaload
    1: iconst_1
    2: aaload
    3: iconst_2
    4: iaload
    5: ireturn
```

678
A Reference to the array’s row is loaded at offset 2, the column is set at offset 3, then iaload loads the array's element.

Three-dimensional arrays

Three-dimensional arrays are just one-dimensional arrays of references to one-dimensional arrays of references to one-dimensional arrays.

```java
public static void main(String[] args)
{
    int[][][] a = new int[5][10][15];
    a[1][2][3]=4;
    get_elem(a);
}
```

```java
public static int get_elem(int[][][] a)
{
    return a[1][2][3];
}
```

Summary

Is it possible to do a buffer overflow in Java?

No, because the array's length is always present in an array object, array bounds are controlled, and an exception is to be raised in case of out-of-bounds access.

There are no multi-dimensional arrays in Java in the C/C++ sense, so Java is not very suited for fast scientific computations.
### 4.1.14 Strings

#### First example

Strings are objects and are constructed in the same way as other objects (and arrays).

```java
public static void main(String[] args)
{
    System.out.println("What is your name?");
    String input = System.console().readLine();
    System.out.println("Hello, "+input);
}
```

The `readLine()` method is called at offset 11, a *reference* to string (which is supplied by the user) is then stored at TOS.

At offset 14 the *reference* to string is stored in slot 1 of LVA.

The string the user entered is reloaded at offset 30 and concatenated with the “Hello, ” string using the `StringBuilder` class.

The constructed string is then printed using `println` at offset 37.

#### Second example

Another example:

```java
public class strings
{
    public static char test (String a)
    {
        return a.charAt(3);
    }

    public static String concat (String a, String b)
    {
        return a+b;
    }
}
```

```java
public static char test(java.lang.String);
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=1, args_size=1
    0: aload_0
```

680
1: iconst_3
2: invokevirtual #2 // Method java/lang/String.charAt:(I)C
5: ireturn

The string concatenation is performed using StringBuilder:

```java
public static java.lang.String concat(java.lang.String, java.lang.String);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=2, args_size=2
0: new #3 // class java/lang/StringBuilder
3: dup
4: invokespecial #4 // Method java/lang/StringBuilder."<init>"():V
7: aload_0
8: invokevirtual #5 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
11: aload_1
12: invokevirtual #5 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
15: invokevirtual #6 // Method java/lang/StringBuilder.toString():Ljava/lang/String;
18: areturn

Another example:

```java
public static void main(String[] args)
{
    String s="Hello!";
    int n=123;
    System.out.println("s = " + s + " n = " + n);
}
```

And again, the strings are constructed using the StringBuilder class and its append method, then the constructed string is passed to println:

```java
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=3, args_size=1
0: ldc #2 // String Hello!
2: astore_1
3: bipush 123
5: istore_2
6: getstatic #3 // Field java/lang/System.out:Ljava/io/PrintStream;
9: new #4 // class java/lang/StringBuilder
12: dup
13: invokespecial #5 // Method java/lang/StringBuilder."<init>"():V
16: ldc #6 // String s =
18: invokevirtual #7 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
21: aload_1
25: ldc #8 // String n =
27: invokevirtual #7 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
30: iload_2
31: invokevirtual #9 // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
34: invokevirtual #10 // Method java/lang/StringBuilder.toString():Ljava/lang/String;
37: invokevirtual #11 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
40: return
```

### 4.1.15 Exceptions

Let's rework our Month example (4.1.13 on page 674) a bit:
Listing 4.10: IncorrectMonthException.java

```java
public class IncorrectMonthException extends Exception {
    private int index;

    public IncorrectMonthException(int index) {
        this.index = index;
    }
    public int getIndex() {
        return index;
    }
}
```

Listing 4.11: Month2.java

```java
class Month2 {
    public static String[] months = {
        "January",
        "February",
        "March",
        "April",
        "May",
        "June",
        "July",
        "August",
        "September",
        "October",
        "November",
        "December"
    };

    public static String get_month (int i) throws IncorrectMonthException {
        if (i<0 || i>11)
            throw new IncorrectMonthException(i);
        return months[i];
    }

    public static void main (String[] args) {
        try {
            System.out.println(get_month(100));
        }
        catch(IncorrectMonthException e) {
            System.out.println("incorrect month index: " + e.getIndex());
            e.printStackTrace();
        }
    }
}
```

Essentially, IncorrectMonthException.class has just an object constructor and one getter method. The IncorrectMonthException class is derived from Exception, so the IncorrectMonthException constructor first calls the constructor of the Exception class, then it puts incoming integer value into the sole IncorrectMonthException class field:

```java
public IncorrectMonthException(int);
flags: ACC_PUBLIC
Code:
    stack=2, locals=2, args_size=2
  0: aload_0
  1: invokespecial #1 // Method java/lang/Exception."<init>"():V
  4: aload_0
  5: iload_1
```
getIndex() is just a getter. A reference to IncorrectMonthException is passed in the zeroth LVA slot (this),aload_0 takes it, getfield loads an integer value from the object, ireturn returns it.

```java
public int getIndex();
flags: ACC_PUBLIC
Code:
stack=1, locals=1, args_size=1
0: aload_0
1: getfield #2 // Field index:I
4: ireturn
```

Now let's take a look at get_month() in Month2.class:

```java
public static java.lang.String get_month(int) throws IncorrectMonthException;
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=1, args_size=1
0: iload_0
1: iflt 10
4: iload_0
5: bipush 11
7: if_icmple 19
10: new #2 // class IncorrectMonthException
13: dup
14: iload_0
15: invokespecial #3 // Method IncorrectMonthException."<init>":(I)V
18: athrow
19: getstatic #4 // Field months:[Ljava/lang/String;
22: iload_0
23: aaload
24: areturn
```

iff at offset 1 is if less than.

In case of invalid index, a new object is created using the new instruction at offset 10.
The object's type is passed as an operand to the instruction (which is IncorrectMonthException).
Then its constructor is called, and index is passed via TOS (offset 15).

When the control flow is offset 18, the object is already constructed, so now the athrow instruction takes a reference to the newly constructed object and signals to JVM to find the appropriate exception handler.
The athrow instruction doesn't return the control flow here, so at offset 19 there is another basic block, not related to exceptions business, where we can get from offset 7.

How do handlers work?

```java
public static void main(java.lang.String[]);
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=3, locals=2, args_size=1
0: getstatic #5 // Field java/lang/System.out:Ljava/io/PrintStream;
3: bipush 100
5: invokevirtual #7 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
11: goto 47
14: astore_1
15: getstatic #5 // Field java/lang/System.out:Ljava/io/PrintStream;
18: new #8 // class java/lang/StringBuilder
21: dup
22: invokespecial #9 // Method java/lang/StringBuilder."<init>"():V
25: ldc #10 // String incorrect month index:
```
Here is the Exception table, which defines that from offsets 0 to 11 (inclusive) an exception IncorrectMonthException may happen, and if it does, the control flow is to be passed to offset 14.

Indeed, the main program ends at offset 11.

At offset 14 the handler starts. It's not possible to get here, there are no conditional/unconditional jumps to this area.

But JVM will transfer the execution flow here in case of an exception.

The very first astore_1 (at 14) takes the incoming reference to the exception object and stores it in LVA slot 1.

Later, the getIndex() method (of this exception object) will be called at offset 31.

The reference to the current exception object is passed right before that (offset 30).

The rest of the code is does just string manipulation: first the integer value returned by getIndex() is converted to string by the toString() method, then it's concatenated with the “incorrect month index: ” text string (like we saw before), then println() and printStackTrace() are called.

After printStackTrace() finishes, the exception is handled and we can continue with the normal execution.

At offset 47 there is a return which finishes the main() function, but there could be any other code which would execute as if no exceptions were raised.

Here is an example on how IDA shows exception ranges:

4.1.16 Classes

Simple class:

Listing 4.15: test.java

```java
public class test {
    public static int a;
    private static int b;
```
public test()
{
    a=0;
    b=0;
}
public static void set_a (int input)
{
    a=input;
}
public static int get_a ()
{
    return a;
}
public static void set_b (int input)
{
    b=input;
}
public static int get_b ()
{
    return b;
}

The constructor just sets both fields to zero:

```java
public test();
flags: ACC_PUBLIC
Code:
   stack=1, locals=1, args_size=1
0: aload_0
1: invokespecial #1 // Method java/lang/Object."<init>"():V
4: iconst_0
5: putstatic #2 // Field a:I
8: iconst_0
9: putstatic #3 // Field b:I
12: return
```

Setter of a:

```java
public static void set_a(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
   stack=1, locals=1, args_size=1
0: iload_0
1: putstatic #2 // Field a:I
4: return
```

Getter of a:

```java
public static int get_a();
flags: ACC_PUBLIC, ACC_STATIC
Code:
   stack=1, locals=0, args_size=0
0: getstatic #2 // Field a:I
3: ireturn
```

Setter of b:

```java
public static void set_b(int);
flags: ACC_PUBLIC, ACC_STATIC
Code:
   stack=1, locals=1, args_size=1
0: iload_0
1: putstatic #3 // Field b:I
4: return
```

Getter of b:

```java
public static int get_b();
flags: ACC_PUBLIC, ACC_STATIC
```
There is no difference in the code which works with public and private fields.
But this type information is present in the .class file, and it’s not possible to access private fields from everywhere.
Let’s create an object and call its method:

```
public class ex1 {
    public static void main(String[] args) {
        test obj = new test();
        obj.set_a(1234);
        System.out.println(obj.a);
    }
}
```

The new instruction creates an object, but doesn’t call the constructor (it is called at offset 4).
The set_a() method is called at offset 16.
The a field is accessed using the getstatic instruction at offset 21.

### 4.1.17 Simple patching

**First example**

Let’s proceed with a simple code patching task.

```
public class nag {
    public static void nag_screen() {
        System.out.println("This program is not registered");
    }
    public static void main(String[] args) {
        System.out.println("Greetings from the mega-software");
        nag_screen();
    }
}
```

How would we remove the printing of “This program is not registered” string?
Let’s load the .class file into IDA:
Let’s patch the first byte of the function to 177 (which is the return instruction’s opcode):

```
; Segment type: Pure code
.method public static nag_screen()V
.limit stack 2
.line 4

177 000 002  ; CODE XREF: main+8jP
018 003      ldc "This program is not registered"
182 000 004  invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
.line 5
177  return
```
Perhaps JVM has some other checks related to the stack maps.

OK, let’s patch it differently by removing the call to nag:

```java
public class password {
    public static void main(String[] args) {
        System.out.println("Please enter the password");
        String input = System.console().readLine();
        if (input.equals("secret"))
            System.out.println("password is correct");
        else
            System.out.println("password is not correct");
    }
}
```

Let’s load it in IDA:

Figure 4.3: IDA

0 is the opcode for NOP.

Now that works!

**Second example**

Another simple crackme example:

```java
public class password {
    public static void main(String[] args) {
        System.out.println("Greetings from the mega-software");
        int i = 0;
        System.out.println("hello");
        System.out.println("world");
    }
}
```

Let’s load it in IDA:
We see here the `ifeq` instruction which does the job.

Its name stands for *if equal*, and this is misnomer, a better name would be `ifz` (*if zero*), i.e., if value at TOS is zero, then do the jump.

In our example, it jumps if the password is not correct (the `equals` method returns `False`, which is 0).

The very first idea is to patch this instruction.

There are two bytes in `ifeq` opcode, which encode the jump offset.

To make this instruction a NOP, we must set the 3rd byte to the value of 3 (because by adding 3 to the current address we will always jump to the next instruction, since the `ifeq` instruction’s length is 3 bytes):
That doesn't work (JRE 1.7):

Exception Details:
Location: password.main([Ljava/lang/String;)V @21: ifeq
Reason: Expected stackmap frame at this location.

But it must be mentioned that it worked in JRE 1.6.
We can also try to replace to all 3 ifeq opcode bytes with zero bytes (NOP), and it still won't work.
Seems like there are more stack map checks in JRE 1.7.

OK, we’ll replace the whole call to the equals method with the  

iconst_1 instruction plus a pack of NOPs:
1 needs always to be in the TOS when the ifeq instruction is executed, so ifeq would never jump.
This works.

4.1.18 Summary

What is missing in Java in comparison to C/C++?

- Structures: use classes.
- Unions: use class hierarchies.
- Unsigned data types. By the way, this makes cryptographic algorithms somewhat harder to implement in Java.
- Function pointers.
Chapter 5

Finding important/interesting stuff in the code

Minimalism it is not a prominent feature of modern software.

But not because the programmers are writing a lot, but because a lot of libraries are commonly linked statically to executable files. If all external libraries were shifted into an external DLL files, the world would be different. (Another reason for C++ are the STL and other template libraries.)

Thus, it is very important to determine the origin of a function, if it is from standard library or well-known library (like Boost\(^1\), libpng\(^2\)), or if it is related to what we are trying to find in the code.

It is just absurd to rewrite all code in C/C++ to find what we’re looking for.

One of the primary tasks of a reverse engineer is to find quickly the code he/she needs, and what is not that important.

The IDA disassembler allow us to search among text strings, byte sequences and constants. It is even possible to export the code to .lst or .asm text files and then use grep, awk, etc.

When you try to understand what some code is doing, this easily could be some open-source library like libpng. So when you see some constants or text strings which look familiar, it is always worth to google them. And if you find the opensource project where they are used, then it’s enough just to compare the functions. It may solve some part of the problem.

For example, if a program uses XML files, the first step may be determining which XML library is used for processing, since the standard (or well-known) libraries are usually used instead of self-made one.

For example, the author of these lines once tried to understand how the compression/decompression of network packets works in SAP 6.0. It is a huge software, but a detailed .PDB with debugging information is present, and that is convenient. He finally came to the idea that one of the functions, that was called CsDecomprLZC, was doing the decompression of network packets. Immediately he tried to google its name and he quickly found the function was used in MaxDB (it is an open-source SAP project)\(^3\).

http://www.google.com/search?q=CsDecomprLZC

Astoundingly, MaxDB and SAP 6.0 software shared likewise code for the compression/decompression of network packets.

5.1 Identification of executable files

5.1.1 Microsoft Visual C++

MSVC versions and DLLs that can be imported:

\(^1\)http://go.yurichev.com/17036
\(^2\)http://go.yurichev.com/17037
\(^3\)More about it in relevant section (8.12.1 on page 860)
<table>
<thead>
<tr>
<th>Marketing ver.</th>
<th>Internal ver.</th>
<th>CL.EXE ver.</th>
<th>DLLs imported</th>
<th>Release date</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6.0</td>
<td>12.00</td>
<td>msvcr.dll, msvcp60.dll</td>
<td>June 1998</td>
</tr>
<tr>
<td>.NET (2002)</td>
<td>7.0</td>
<td>13.00</td>
<td>msvcr70.dll, msvcp70.dll</td>
<td>February 13, 2002</td>
</tr>
<tr>
<td>.NET 2003</td>
<td>7.1</td>
<td>13.10</td>
<td>msvcr71.dll, msvcp71.dll</td>
<td>April 24, 2003</td>
</tr>
<tr>
<td>2005</td>
<td>8.0</td>
<td>14.00</td>
<td>msvcr80.dll, msvcp80.dll</td>
<td>November 7, 2005</td>
</tr>
<tr>
<td>2008</td>
<td>9.0</td>
<td>15.00</td>
<td>msvcr90.dll, msvcp90.dll</td>
<td>November 19, 2007</td>
</tr>
<tr>
<td>2010</td>
<td>10.0</td>
<td>16.00</td>
<td>msvcr100.dll, msvcp100.dll</td>
<td>April 12, 2010</td>
</tr>
<tr>
<td>2012</td>
<td>11.0</td>
<td>17.00</td>
<td>msvcr110.dll, msvcp110.dll</td>
<td>September 12, 2012</td>
</tr>
<tr>
<td>2013</td>
<td>12.0</td>
<td>18.00</td>
<td>msvcr120.dll, msvcp120.dll</td>
<td>October 17, 2013</td>
</tr>
</tbody>
</table>

msvcp*.dll has C++-related functions, so if it is imported, this is probably a C++ program.

**Name mangling**

The names usually start with the `?` symbol.

You can read more about MSVC’s name mangling here: 3.21.1 on page 542.

### 5.1.2 GCC

Aside from *NIX targets, GCC is also present in the win32 environment, in the form of Cygwin and MinGW.

**Name mangling**

Names usually start with the `_Z` symbols.

You can read more about GCC’s name mangling here: 3.21.1 on page 542.

**Cygwin**

cygwin1.dll is often imported.

**MinGW**

msvcr.dll may be imported.

### 5.1.3 Intel Fortran

libifcoremd.dll, libifportmd.dll and libiomp5md.dll (OpenMP support) may be imported.

libifcoremd.dll has a lot of functions prefixed with `for_`, which means Fortran.

### 5.1.4 Watcom, OpenWatcom

**Name mangling**

Names usually start with the `W` symbol.

For example, that is how the method named “method” of the class “class” that does not have any arguments and returns `void` is encoded:

```
W?method$_class$n__v
```
5.1.5 Borland

Here is an example of Borland Delphi’s and C++Builder’s name mangling:

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDDAccels$qp6tagMSG
@TModule@sbctr$qpcpvt1
@TModule@sbctr$qv
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiiit1iiiiii
@TrueColorTo16BitN$qpviiiiiit1iiiiii
@DIB24BitTo8BitBitmap$qpviiiiiit1iiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpvl
@TrueBitmap@$bctr$qilll
```

The names always start with the @ symbol, then we have the class name came, method name, and encoded the types of the arguments of the method.

These names can be in the .exe imports, .dll exports, debug data, etc.

Borland Visual Component Libraries (VCL) are stored in .bpl files instead of .dll ones, for example, vcl50.dll, rtl60.dll.

Another DLL that might be imported: BORLNDMM.DLL.

**Delphi**

Almost all Delphi executables has the “Boolean” text string at the beginning of the code segment, along with other type names.

This is a very typical beginning of the CODE segment of a Delphi program, this block came right after the win32 PE file header:

```
00000400 04 10 40 00 03 07 42 6f 6f 6c 65 61 6e 01 00 00 |...Boolean...|
00000410 00 00 01 00 00 00 00 00 10 40 00 05 46 61 6c 73 65 |.........@...False|
00000420 04 54 72 75 65 8d 40 00 2c 10 40 00 09 08 57 69 |.True.@...Wi|
00000430 04 07 09 64 65 74 69 6e 67 69 6e 74 02 00 ff ff |deChar......|
00000440 7f 00 00 00 80 ff ff ff ff ff ff 7f 00 00 00 90 |..........@...Wo|
00000450 01 08 43 61 72 64 69 6e 61 6c 05 00 00 00 00 ff |..Cardinal......|
00000460 14 11 0f 68 6c 79 73 74 65 64 03 00 00 00 00 00 | WideString0.@.|
00000470 00 00 00 00 00 00 00 06 53 79 73 74 65 6d 00 00 |........System..|
00000480 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000490 cc 83 44 24 04 f8 e9 51 6c 00 00 83 44 24 04 f8 |..D$.Object...|
000004a0 a4 11 40 00 04 08 45 73 70 61 74 63 68 02 00 00 |IInterface......|
000004b0 cc 21 12 40 00 03 00 00 00 00 00 00 00 00 00 00 |..System...|
000004c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
000004d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
000004e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
000004f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000500 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000510 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000520 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000530 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000540 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000550 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000560 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000570 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000580 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000590 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
000005a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
000005b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
000005c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
000005d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
000005e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
000005f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000600 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000610 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000620 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000630 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
00000640 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |............|
```

694
The first 4 bytes of the data segment (DATA) can be 00 00 00 00, 32 13 88 C0 or FF FF FF FF.
This information can be useful when dealing with packed/encrypted Delphi executables.

5.1.6 Other known DLLs

- vcomp*.dll—Microsoft’s implementation of OpenMP.

5.2 Communication with outer world (function level)

It’s often advisable to track function arguments and return values in debugger or DBI. For example, the author once tried to understand meaning of some obscure function, which happens to be incorrectly implemented bubble sort⁴. (It worked correctly, but slower.) Meanwhile, watching inputs and outputs of this function helps instantly to understand what it does.

Often, when you see division by multiplication (3.12 on page 496), but forgot all details about its mechanics, you can just observe input and output and quickly find divisor.

5.3 Communication with the outer world (win32)

Sometimes it’s enough to observe some function’s inputs and outputs in order to understand what it does. That way you can save time.

Files and registry access: for the very basic analysis, Process Monitor⁵ utility from SysInternals can help. For the basic analysis of network accesses, Wireshark⁶ can be useful.

But then you will have to look inside anyway.

The first thing to look for is which functions from the OS’s APIs and standard libraries are used.

If the program is divided into a main executable file and a group of DLL files, sometimes the names of the functions in these DLLs can help.

If we are interested in exactly what can lead to a call to MessageBox() with specific text, we can try to find this text in the data segment, find the references to it and find the points from which the control may be passed to the MessageBox() call we’re interested in.

If we are talking about a video game and we’re interested in which events are more or less random in it, we may try to find the rand() function or its replacements (like the Mersenne twister algorithm) and find the places from which those functions are called, and more importantly, how are the results used. One example: 8.3 on page 791.

But if it is not a game, and rand() is still used, it is also interesting to know why. There are cases of unexpected rand() usage in data compression algorithms (for encryption imitation): blog.yurichev.com.

---

⁴https://yurichev.com/blog/weird_sort_KLEE/
⁵http://go.yurichev.com/17301
⁶http://go.yurichev.com/17303
5.3.1 Often used functions in the Windows API

These functions may be among the imported. It is worth to note that not every function might be used in the code that was written by the programmer. A lot of functions might be called from library functions and CRT code.

Some functions may have the -A suffix for the ASCII version and -W for the Unicode version.

- Registry access (advapi32.dll): RegEnumKeyEx, RegEnumValue, RegGetValue, RegOpenKeyEx, RegQueryValueEx.
- Dialog boxes (user32.dll): MessageBox, MessageBoxEx, CreateDialog, SetDlgItemText, GetDlgItemText.
- Resources access (6.5.2 on page 754): (user32.dll): LoadMenu.
- TCP/IP networking (ws2_32.dll): WSARcv, WSASend.
- File access (kernel32.dll): CreateFile, ReadFile, ReadFileEx, WriteFile, WriteFileEx.
- High-level access to the Internet (wininet.dll): WinHttpOpen.
- Checking the digital signature of an executable file (wintrust.dll): WinVerifyTrust.
- The standard MSVC library (if it’s linked dynamically) (msvcr*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

5.3.2 Extending trial period

Registry access functions are frequent targets for those who try to crack trial period of some software, which may save installation date/time into registry.

Another popular target are GetLocalTime() and GetSystemTime() functions: a trial software, at each startup, must check current date/time somehow anyway.

5.3.3 Removing nag dialog box

A popular way to find out what causing popping nag dialog box is intercepting MessageBox(), CreateDialog() and CreateWindow() functions.

5.3.4 tracer: Intercepting all functions in specific module

There are INT3 breakpoints in the tracer, that are triggered only once, however, they can be set for all functions in a specific DLL.

```
--one-time-INT3-bp:somedll.dll!.*
```

Or, let’s set INT3 breakpoints on all functions with the xml prefix in their name:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

On the other side of the coin, such breakpoints are triggered only once. Tracer will show the call of a function, if it happens, but only once. Another drawback—it is impossible to see the function’s arguments.

Nevertheless, this feature is very useful when you know that the program uses a DLL, but you do not know which functions are actually used. And there are a lot of functions.

For example, let’s see, what does the uptime utility from cygwin use:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!* 
```

Thus we may see all that cygwin1.dll library functions that were called at least once, and where from:

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime.exe!OEP+0x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!geteuid32 (called from uptime.exe!OEP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!getuid32 (called from uptime.exe!OEP+0xbaa (0x401bba))
One-time INT3 breakpoint: cygwin1.dll!getegid32 (called from uptime.exe!OEP+0xcb7 (0x401cbb))
One-time INT3 breakpoint: cygwin1.dll!getgid32 (called from uptime.exe!OEP+0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!OEP+0x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!OEP+0x77b2 (0x4017b2))
```
5.4 Strings

5.4.1 Text strings

C/C++

The normal C strings are zero-terminated (ASCIIZ-strings).

The reason why the C string format is as it is (zero-terminated) is apparently historical. In [Dennis M. Ritchie, The Evolution of the Unix Time-sharing System, (1979)] we read:

A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters.

In Hiew or FAR Manager these strings look like this:

```c
int main()
{
    printf("Hello, world!\n");
}
```

![Hiew](image)

Figure 5.1: Hiew

Borland Delphi

The string in Pascal and Borland Delphi is preceded by an 8-bit or 32-bit string length.

For example:

```pascal
Listing 5.1: Delphi
```

<table>
<thead>
<tr>
<th>CODE</th>
<th>ASCII</th>
<th>Hex</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>00518AC8</td>
<td>dd</td>
<td>19h</td>
<td></td>
</tr>
<tr>
<td>00518ACC</td>
<td>aLoading__Plea db 'Loading... , please wait.',0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00518AFC</td>
<td>dd</td>
<td>10h</td>
<td></td>
</tr>
</tbody>
</table>

697
Unicode

Often, what is called Unicode is a methods for encoding strings where each character occupies 2 bytes or 16 bits. This is a common terminological mistake. Unicode is a standard for assigning a number to each character in the many writing systems of the world, but does not describe the encoding method.

The most popular encoding methods are: UTF-8 (is widespread in Internet and *NIX systems) and UTF-16LE (is used in Windows).

UTF-8

UTF-8 is one of the most successful methods for encoding characters. All Latin symbols are encoded just like in ASCII, and the symbols beyond the ASCII table are encoded using several bytes. 0 is encoded as before, so all standard C string functions work with UTF-8 strings just like any other string.

Let's see how the symbols in various languages are encoded in UTF-8 and how it looks like in FAR, using the 437 codepage:

![Figure 5.2: FAR: UTF-8](image)

As you can see, the English language string looks the same as it is in ASCII.

The Hungarian language uses some Latin symbols plus symbols with diacritic marks.

These symbols are encoded using several bytes, these are underscored with red. It's the same story with the Icelandic and Polish languages.

There is also the “Euro” currency symbol at the start, which is encoded with 3 bytes.

The rest of the writing systems here have no connection with Latin.

7The example and translations was taken from here: [http://go.yurichev.com/17304](http://go.yurichev.com/17304)
At least in Russian, Arabic, Hebrew and Hindi we can see some recurring bytes, and that is not surprise: all symbols from a writing system are usually located in the same Unicode table, so their code begins with the same numbers.

At the beginning, before the “How much?” string we see 3 bytes, which are in fact the BOM. The BOM defines the encoding system to be used.

**UTF-16LE**

Many win32 functions in Windows have the suffixes -A and -W. The first type of functions works with normal strings, the other with UTF-16LE strings (*wide*).

In the second case, each symbol is usually stored in a 16-bit value of type `short`.

The Latin symbols in UTF-16 strings look in Hiew or FAR like they are interleaved with zero byte:

```c
int wmain()
{
    wprintf(L"Hello, world!
");  
}
```

![Figure 5.3: Hiew](image)

We can see this often in Windows NT system files:

![Figure 5.4: Hiew](image)

Strings with characters that occupy exactly 2 bytes are called “Unicode” in IDA:

```
.data:0040E000 aHelloWorld:  
.data:0040E000  unicode 0, <Hello, world>
.data:0040E000  dw 0Ah, 0
```

---

8Font Order Mark
Here is how the Russian language string is encoded in UTF-16LE:

![Figure 5.5: Hiew: UTF-16LE](image)

What we can easily spot is that the symbols are interleaved by the diamond character (which has the ASCII code of 4). Indeed, the Cyrillic symbols are located in the fourth Unicode plane. Hence, all Cyrillic symbols in UTF-16LE are located in the 0x400-0x4FF range.

Let's go back to the example with the string written in multiple languages. Here is how it looks like in UTF-16LE:

![Figure 5.6: FAR: UTF-16LE](image)

Here we can also see the BOM at the beginning. All Latin characters are interleaved with a zero byte. Some characters with diacritic marks (Hungarian and Icelandic languages) are also underscored in red.

**Base64**

The base64 encoding is highly popular for the cases when you have to transfer binary data as a text string. In essence, this algorithm encodes 3 binary bytes into 4 printable characters: all 26 Latin letters (both lower and upper case), digits, plus sign (“+”) and slash sign (“/”), 64 characters in total.

700
One distinctive feature of base64 strings is that they often (but not always) end with 1 or 2 padding equality symbol(s) ("="), for example:

```
AVjbbVSfCUmxvaGmpNg==
WVjbbVSfCUmxvaGmpNg==
```

The equality sign ("=") is never encounter in the middle of base64-encoded strings.

Now example of manual encoding. Let’s encode 0x00, 0x11, 0x22, 0x33 hexadecimal bytes into base64 string:

```
$ echo -n "\00\x11\x22\x33" | base64
ABEiMw==
```

Let’s put all 4 bytes in binary form, then regroup them into 6-bit groups:

```
| 00 || 11 || 22 || 33 || || |
00000000000100010010001000110011????????????????
| A || B || E || i || M || w || = || = |
```

Three first bytes (0x00, 0x11, 0x22) can be encoded into 4 base64 characters ("ABEi"), but the last one (0x33) — cannot be, so it’s encoded using two characters ("Mw") and padding symbol ("=") is added twice to pad the last group to 4 characters. Hence, length of all correct base64 strings are always divisible by 4.

Base64 is often used when binary data needs to be stored in XML. “Armored” (i.e., in text form) PGP keys and signatures are encoded using base64.


There are utilities for scanning an arbitrary binary files for base64 strings. One such utility is base64scanner.

Another encoding system which was much more popular in UseNet and FidoNet is Uuencoding. Binary files are still encoded in Uuencode format in Phrack magazine. It offers mostly the same features, but is different from base64 in the sense that file name is also stored in header.

By the way: there is also close sibling to base64: base32, alphabet of which has 10 digits and 26 Latin characters. One well-known usage of it is onion addresses, like: [http://3g2upl4pq6kufc4m.onion/](http://3g2upl4pq6kufc4m.onion/). URL can’t have mixed-case Latin characters, so apparently, this is why Tor developers used base32.

### 5.4.2 Finding strings in binary

Actually, the best form of Unix documentation is frequently running the `strings` command over a program’s object code. Using `strings`, you can get a complete list of the program’s hard-coded file name, environment variables, undocumented options, obscure error messages, and so forth.

The Unix-Haters Handbook

The standard UNIX `strings` utility is quick-n-dirty way to see strings in file. For example, these are some strings from OpenSSH 7.2 sshd executable file:

```
... 0123 0123456789 0123456789abcdefABCDEF.:/% 02x ...
%.100s, line %lu: Bad permitopen specification <%.100s>
```
There are options, error messages, file paths, imported dynamic modules and functions, some other strange strings (keys?) There is also unreadable noise—x86 code sometimes has chunks consisting of printable ASCII characters, up to 8 characters.

Of course, OpenSSH is open-source program. But looking at readable strings inside of some unknown binary is often a first step of analysis.

grep can be applied as well.

Hiew has the same capability (Alt-F6), as well as Sysinternals ProcessMonitor.

### 5.4.3 Error/debug messages

Debugging messages are very helpful if present. In some sense, the debugging messages are reporting what’s going on in the program right now. Often these are printf()-like functions, which write to log-files, or sometimes do not writing anything but the calls are still present since the build is not a debug one but release one.

If local or global variables are dumped in debug messages, it might be helpful as well since it is possible to get at least the variable names. For example, one of such function in Oracle RDBMS is ksdwrt().

Meaningful text strings are often helpful. The IDA disassembler may show from which function and from which point this specific string is used. Funny cases sometimes happen\(^\text{12}\).

The error messages may help us as well. In Oracle RDBMS, errors are reported using a group of functions. You can read more about them here: blog.yurichev.com.

It is possible to find quickly which functions report errors and in which conditions.

By the way, this is often the reason why copy-protection systems use inarticulate cryptic error messages or just error numbers. No software author is happy if the software cracker can quickly understands copy-protection’s inner workings judging by error messages it can produce.

One example of encrypted error messages is here: 8.8.2 on page 826.

\(^{12}\text{blog.yurichev.com}\)
5.4.4 Suspicious magic strings

Some magic strings which are usually used in backdoors look pretty suspicious.

For example, there was a backdoor in the TP-Link WR740 home router. The backdoor can activated using the following URL:

http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html.

Indeed, the “userRpmNatDebugRpm26525557” string is present in the firmware.

This string was not googleable until the wide disclosure of information about the backdoor.

You would not find this in any RFC.

You would not find any computer science algorithm which uses such strange byte sequences. And it doesn’t look like an error or debugging message.

So it’s a good idea to inspect the usage of such weird strings.

Sometimes, such strings are encoded using base64.

So it’s a good idea to decode them all and to scan them visually, even a glance should be enough.

More precise, this method of hiding backdoors is called “security through obscurity”.

5.5 Calls to assert()

Sometimes the presence of the assert() macro is useful too: commonly this macro leaves source file name, line number and condition in the code.

The most useful information is contained in the assert’s condition, we can deduce variable names or structure field names from it. Another useful piece of information are the file names—we can try to deduce what type of code is there. Also it is possible to recognize well-known open-source libraries by the file names.

Listing 5.2: Example of informative assert() calls

```
.text:107D4B29 mov dx, [ecx+42h]
.text:107D4B2D cmp edx, 1
.text:107D4B30 jz short loc_107D4B4A
.text:107D4B32 push 1Ec
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ; "td->td_planarconfig == PLANARCONFIG_CON"
.text:107D4B41 call ds:_assert

... 
.text:107D52CA mov edx, [ebp-4]
.text:107D52CD and edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz short loc_107D52D9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX"
.text:107D52E0 call ds:_assert

...

.text:107D6759 mov cx, [eax+6]
.text:107D675D cmp ecx, 0Ch
.text:107D6760 jle short loc_107D677A
.text:107D6762 push 208h
.text:107D6767 push offset aLzw_c ; "lzw.c"
.text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= BITS_MAX"
.text:107D6771 call ds:_assert
```

\(^{13}\)http://sekurak.pl/tp-link-httptftp-backdoor/

\(^{14}\)Request for Comments
It is advisable to “google” both the conditions and file names, which can lead us to an open-source library. For example, if we “google” “sp->lzw_nbits <= BITS_MAX”, this predictably gives us some open-source code that’s related to the LZW compression.

### 5.6 Constants

Humans, including programmers, often use round numbers like 10, 100, 1000, in real life as well as in the code.

The practicing reverse engineer usually know them well in hexadecimal representation: 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

The constants 0xA8000000 (0b10101001000000000000000000000000) and 0x80000000 (0b01100000000000000000000000000000) are also popular—those are composed of alternating bits.

That may help to distinguish some signal from a signal where all bits are turned on (0b1111 ...) or off (0b0000 ...). For example, the 0x55AA constant is used at least in the boot sector, MBR, and in the ROM of IBM-compatible extension cards.

Some algorithms, especially cryptographical ones use distinct constants, which are easy to find in code using IDA.

For example, the MD5 algorithm initializes its own internal variables like this:

```plaintext
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

If you find these four constants used in the code in a row, it is highly probable that this function is related to MD5.

Another example are the CRC16/CRC32 algorithms, whose calculation algorithms often use precomputed tables like this one:

**Listing 5.3: linux/lib/crc16.c**

```c
/*
 * CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + x^2 + 1) */
const u16 crc16_table[256] = {
   0x0000, 0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007, 0x0008, 0x0009, 0x000A, 0x000B, 0x000C, 0x000D, 0x000E, 0x000F,
   0xA800, 0xA801, 0xA802, 0xA803, 0xA804, 0xA805, 0xA806, 0xA807, 0xA808, 0xA809, 0xA80A, 0xA80B, 0xA80C, 0xA80D, 0xA80E, 0xA80F,
   0x0010, 0x0200, 0x0400, 0x0600, 0x0800, 0x0A00, 0x0C00, 0x0E00, 0x1000, 0x1200, 0x1400, 0x1600, 0x1800, 0x1A00, 0x1C00, 0x1E00,
   0x0020, 0x0210, 0x0410, 0x0610, 0x0810, 0x0A10, 0x0C10, 0x0E10, 0x1020, 0x1220, 0x1420, 0x1620, 0x1820, 0x1A20, 0x1C20, 0x1E20,
   0x0030, 0x0230, 0x0430, 0x0630, 0x0830, 0x0A30, 0x0C30, 0x0E30, 0x1030, 0x1230, 0x1430, 0x1630, 0x1830, 0x1A30, 0x1C30, 0x1E30,
   0x0040, 0x0240, 0x0440, 0x0640, 0x0840, 0x0A40, 0x0C40, 0x0E40, 0x1040, 0x1240, 0x1440, 0x1640, 0x1840, 0x1A40, 0x1C40, 0x1E40,
   0x0050, 0x0250, 0x0450, 0x0650, 0x0850, 0x0A50, 0x0C50, 0x0E50, 0x1050, 0x1250, 0x1450, 0x1650, 0x1850, 0x1A50, 0x1C50, 0x1E50,
   0x0060, 0x0260, 0x0460, 0x0660, 0x0860, 0x0A60, 0x0C60, 0x0E60, 0x1060, 0x1260, 0x1460, 0x1660, 0x1860, 0x1A60, 0x1C60, 0x1E60,
   ...}
```

See also the precomputed table for CRC32: 3.8 on page 482.

In tableless CRC algorithms well-known polynomials are used, for example, 0xEDB88320 for CRC32.

### 5.6.1 Magic numbers

A lot of file formats define a standard file header where a magic number(s) is used, single one or even several.

For example, all Win32 and MS-DOS executables start with the two characters “MZ”.

At the beginning of a MIDI file the “MThd” signature must be present. If we have a program which uses MIDI files for something, it’s very likely that it must check the file for validity by checking at least the first 4 bytes.

This could be done like this: (buf points to the beginning of the loaded file in memory)

```plaintext
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

...or by calling a function for comparing memory blocks like memcmp() or any other equivalent code up to a CMPSB (1.6 on page 1018) instruction.

When you find such point you already can say where the loading of the MIDI file starts, also, we could see the location of the buffer with the contents of the MIDI file, what is used from the buffer, and how.

---

15 Master Boot Record
Dates

Often, one may encounter number like 0x19870116, which is clearly looks like a date (year 1987, 1st month (January), 16th day). This may be someone’s birthday date (a programmer, his/her relative, child), or some other important date. The date may also be written in a reverse order, like 0x16011987. American-style dates are also popular, like 0x01161987.

Well-known example is 0x19540119 (magic number used in UFS2 superblock structure), which is a birthday date of Marshall Kirk McKusick, prominent FreeBSD contributor.

Stuxnet uses the number “19790509” (not as 32-bit number, but as string, though), and this led to speculation that the malware is connected to Israel.\(^{16}\)

Also, numbers like those are very popular in amateur-grade cryptography, for example, excerpt from the secret function internals from HASP3 dongle:\(^{17}\)

```c
void xor_pwd(void)
{
    int i;
    pwd ^= 0x9071966;
    for (i=0; i<8; i++)
    {
        al_buf[i] = pwd & 7; pwd = pwd >> 3;
    }
}

void emulate_func2(unsigned short seed)
{
    int i, j;
    for (i=0; i<8; i++)
    {
        ch[i] = 0;
        for (j=0; j<8; j++)
        {
            seed *= 0x1989;
            seed += 5;
            ch[i] |= (tab[(seed>>9)&0x3f]) << (7-j);
        }
    }
}
```

DHCP

This applies to network protocols as well. For example, the DHCP protocol’s network packets contains the so-called magic cookie: 0x63538263. Any code that generates DHCP packets somewhere must embed this constant into the packet. If we find it in the code we may find where this happens and, not only that. Any program which can receive DHCP packet must verify the magic cookie, comparing it with the constant.

For example, let’s take the dhcpcore.dll file from Windows 7 x64 and search for the constant. And we can find it, twice: it seems that the constant is used in two functions with descriptive names DhcpExtractOptionsForValidation() and DhcpExtractFullOptions():

Listing 5.4: dhcpcore.dll (Windows 7 x64)

```c
.rdata:000007FF6483CBE8 dd 63538263h ; DATA XREF:
    DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBE8 dd 63538263h ; DATA XREF:
    DhcpExtractFullOptions+97`
```

And here are the places where these constants are accessed:

Listing 5.5: dhcpcore.dll (Windows 7 x64)

```c
.text:000007FF6480875F mov eax, [rsi]
.text:000007FF64808761 cmp eax, cs:word_7FF6483CBE8
.text:000007FF64808767 jnz loc_7FF64817179
```

\(^{16}\)This is a date of execution of Habib Elghanian, persian jew.

5.6.2 Specific constants

Sometimes, there is a specific constant for some type of code. For example, the author once dug into a code, where number 12 was encountered suspiciously often. Size of many arrays is 12, or multiple of 12 (24, etc). As it turned out, that code takes 12-channel audio file at input and process it.

And vice versa: for example, if a program works with text field which has length of 120 bytes, there has to be a constant 120 or 119 somewhere in the code. If UTF-16 is used, then $2 \times 120$. If a code works with network packets of fixed size, it’s good idea to search for this constant in the code as well.

This is also true for amateur cryptography (license keys, etc). If encrypted block has size of $n$ bytes, you may want to try to find occurrences of this number throughout the code. Also, if you see a piece of code which is been repeated $n$ times in loop during execution, this may be encryption/decryption routine.

5.6.3 Searching for constants

It is easy in IDA: Alt-B or Alt-I. And for searching for a constant in a big pile of files, or for searching in non-executable files, there is a small utility called binary grep\(^\text{\textsuperscript{18}}\).

5.7 Finding the right instructions

If the program is utilizing FPU instructions and there are very few of them in the code, one can try to check each one manually with a debugger.

For example, we may be interested how Microsoft Excel calculates the formulae entered by user. For example, the division operation.

If we load excel.exe (from Office 2010) version 14.0.4756.1000 into IDA, make a full listing and to find every FDIV instruction (except the ones which use constants as a second operand—obviously, they do not suit us):

```bash
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...then we see that there are 144 of them.

We can enter a string like \(=\left(\frac{1}{3}\right)\) in Excel and check each instruction.

By checking each instruction in a debugger or tracer (one may check 4 instruction at a time), we get lucky and the sought-for instruction is just the 14th:

```plaintext
.text:3011E919 DC 33    fdiv  qword ptr [ebx]
```

ST(0) holds the first argument (1) and second one is in [EBX].

The instruction after FDIV (FSTP) writes the result in memory:

\(^{18}\)GitHub

706
If we set a breakpoint on it, we can see the result:

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

Also as a practical joke, we can modify it on the fly:

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel shows 666 in the cell, finally convincing us that we have found the right point.

![Excel Sheet](image)

Figure 5.7: The practical joke worked

If we try the same Excel version, but in x64, we will find only 12 FDIV instructions there, and the one we looking for is the third one.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

It seems that a lot of division operations of `float` and `double` types, were replaced by the compiler with SSE instructions like DIVSD (DIVSD is present 268 times in total).

### 5.8 Suspicious code patterns

#### 5.8.1 XOR instructions

Instructions like XOR op, op (for example, XOR EAX, EAX) are usually used for setting the register value to zero, but if the operands are different, the “exclusive or” operation is executed.
This operation is rare in common programming, but widespread in cryptography, including amateur one. It’s especially suspicious if the second operand is a big number. This may point to encrypting/decrypting, checksum computing, etc.

One exception to this observation worth noting is the “canary” (1.26.3 on page 279). Its generation and checking are often done using the XOR instruction.

This AWK script can be used for processing IDA listing (.lst) files:

```awk
awk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=$4) if($4!="esp") if ($4!="ebp")
 \ { print $1, $2, tmp, ",", $4 } }' filename.lst
```

It is also worth noting that this kind of script can also match incorrectly disassembled code (5.11.1 on page 719).

### 5.8.2 Hand-written assembly code

Modern compilers do not emit the LOOP and RCL instructions. On the other hand, these instructions are well-known to coders who like to code directly in assembly language. If you spot these, it can be said that there is a high probability that this fragment of code was hand-written. Such instructions are marked as (M) in the instructions list in this appendix: .1.6 on page 1012.

Also the function prologue/epilogue are not commonly present in hand-written assembly.

Commonly there is no fixed system for passing arguments to functions in the hand-written code.

Example from the Windows 2003 kernel (ntoskrnl.exe file):

```asm
MultiplyTest proc near
    ; CODE XREF: Get386Stepping
    xor     cx, cx
loc_620555:
    ; CODE XREF: MultiplyTest+E
    push    cx
    call    Multiply
    pop     cx
    jb      short locret_620563
    loop    loc_620555
    clc
locret_620563:
    ; CODE XREF: MultiplyTest+C
    retn
MultiplyTest endp

Multiply proc near
    ; CODE XREF: MultiplyTest+5
    mov     ecx, 81h
    mov     eax, 417A000h
    mul     ecx
    cmp     edx, 2
    stc
    jnz     short locret_62057F
    cmp     eax, 0FE7A000h
    stc
    jnz     short locret_62057F
    clc
locret_62057F:
    ; CODE XREF: Multiply+10
    ; Multiply+18
    retn
Multiply endp
```

Indeed, if we look in the WRK 19 v1.2 source code, this code can be found easily in file WRK-v1.2\base\ntos\ke\i386\cpu.asm.

As of RCL instruction, I can find it in the ntosrnl.exe file in Windows 2003 x86 (compiled with MS Visual C compiler). It is occurred only once there, in RtlExtendedLargeIntegerDivide() function, and this might be inline assembler code case.

19Windows Research Kernel
5.9 Using magic numbers while tracing

Often, our main goal is to understand how the program uses a value that has been either read from file or received via network. The manual tracing of a value is often a very labor-intensive task. One of the simplest techniques for this (although not 100% reliable) is to use your own magic number.

This resembles X-ray computed tomography in some sense: a radiocontrast agent is injected into the patient’s blood, which is then used to improve the visibility of the patient’s internal structure in to the X-rays. It is well known how the blood of healthy humans percolates in the kidneys and if the agent is in the blood, it can be easily seen on tomography, how blood is percolating, and are there any stones or tumors.

We can take a 32-bit number like 0x0badf00d, or someone’s birth date like 0x11101979 and write this 4-byte number to some point in a file used by the program we investigate.

Then, while tracing this program with tracer in code coverage mode, with the help of grep or just by searching in the text file (of tracing results), we can easily see where the value has been used and how.

Example of grepable tracer results in cc mode:

```
0x150bf66 (kziaia+0x14), e= 1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934
0x150bf69 (kziaia+0x17), e= 1 [MOV EDX, [69AEB08h]] [69AEB08h]=0
0x150bf6f (kziaia+0x1d), e= 1 [FS: MOV EAX, [2Ch]]
0x150bf75 (kziaia+0x23), e= 1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0xf1ac360
0x150bf78 (kziaia+0x26), e= 1 [MOV [EBP-4], ECX] ECX=0xf1ac360
```

This can be used for network packets as well. It is important for the magic number to be unique and not to be present in the program’s code.

Aside of the tracer, DosBox (MS-DOS emulator) in heavydebug mode is able to write information about all registers’ states for each executed instruction of the program to a plain text file, so this technique may be useful for DOS programs as well.

5.10 Loops

Whenever your program works with some kind of file, or buffer of some size, it has to be some kind of decrypting/processing loop inside of the code.

This is a real example of tracer tool output. There was a code which loads some kind of encryted file of 258 bytes. I run it with the intention to get each instruction counts (a DBI tool will serve much better these days). And I quickly found a piece of code, which executed 259/258 times:

```
...  
0x45a6b5 e= 1 [FS: MOV [0], EAX] EAX=0x218fb08
0x45a6bb e= 1 [MOV [EBP-254h], ECX] ECX=0x218fbd8
0x45a6c1 e= 1 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a6c7 e= 1 [CMP [EAX+14h], 0] [EAX+14h]=0x102
0x45a6cb e= 1 [JZ 45A9F2h] ZF=false
0x45a6d1 e= 1 [MOV [EBP-0Dh], 1]
0x45a6d5 e= 1 [XOR ECX, ECX] ECX=0x218fbd8
0x45a6d7 e= 1 [MOV [EBP-14h], CX] CX=0
0x45a6db e= 1 [MOV [EBP-18h], 0]
0x45a6de e= 1 [JMP 45A6EDAh]
0x45a6e4 e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0xfd..0x101
0x45a6e7 e= 258 [ADD EDX, 1] EDX=0..5 (248 items skipped) 0xfd..0x101
0x45a6ea e= 258 [MOV [EBP-18h], EDX] EDX=1..6 (248 items skipped) 0xfe..0x102
0x45a6ed e= 259 [MOV EAX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a6f3 e= 259 [MOV ECX, [EBP-18h]] [EBP-18h]=0..5 (249 items skipped) 0xfe..0x102
0x45a6f6 e= 259 [CMP ECX, [EAX+14h]] ECX=0..5 (249 items skipped) 0xfe..0x102 [EAX+14h]=0x102
0x45a6f9 e= 259 [JNB 45A727h] CF=false,true
0x45a701 e= 258 [MOV EDX, [EBP-254h]] [EBP-254h]=0x218fbd8
0x45a707 e= 258 [ADD ECX, 1] ECX=0..5 (248 items skipped) 0xfd..0x101
0x45a70a e= 258 [IMUL ECX, ECX, 1Fh] ECX=1..6 (248 items skipped) 0xfe..0x102
0x45a70d e= 258 [MOV EDX, [EBP-18h]] [EBP-18h]=0..5 (248 items skipped) 0xfd..0x101
```

20See also my blog post about this DosBox feature: blog.yurichev.com
As it turns out, this is the decrypting loop.

### 5.10.1 Some binary file patterns

All examples here were prepared on the Windows with active code page 437 in console. Binary files internally may look visually different if another code page is set.
Arrays

Sometimes, we can clearly spot an array of 16/32/64-bit values visually, in hex editor. Here is an example of array of 16-bit values. We see that the first byte in pair is 7 or 8, and the second looks random:

![Figure 5.8: FAR: array of 16-bit values](image)

I used a file containing 12-channel signal digitized using 16-bit ADC\(^\text{21}\).
And here is an example of very typical MIPS code.

As we may recall, every MIPS (and also ARM in ARM mode or ARM64) instruction has size of 32 bits (or 4 bytes), so such code is array of 32-bit values.

By looking at this screenshot, we may see some kind of pattern.

Vertical red lines are added for clarity.

![Hiew: very typical MIPS code](image)

Figure 5.9: Hiew: very typical MIPS code

Another example of such pattern here is book: [9.5 on page 954](#).
Sparse files

This is sparse file with data scattered amidst almost empty file. Each space character here is in fact zero byte (which looks like space). This is a file to program FPGA (Altera Stratix GX device). Of course, files like these can be compressed easily, but formats like this one are very popular in scientific and engineering software where efficient access is important while compactness is not.

Figure 5.10: FAR: Sparse file
Compressed file

This file is just some compressed archive. It has relatively high entropy and visually looks just chaotic. This is how compressed and/or encrypted files looks like.

Figure 5.11: FAR: Compressed file
OS installations are usually distributed as ISO files which are copies of CD/DVD discs. Filesystem used is named CDFS, here is you see file names mixed with some additional data. This can be file sizes, pointers to another directories, file attributes, etc. This is how typical filesystems may look internally.

Figure 5.12: FAR: ISO file: Ubuntu 15 installation

22Compact Disc File System
32-bit x86 executable code

This is how 32-bit x86 executable code looks like. It has not very high entropy, because some bytes occurred more often than others.

Figure 5.13: FAR: Executable 32-bit x86 code
BMP graphics files

BMP files are not compressed, so each byte (or group of bytes) describes each pixel. I’ve found this picture somewhere inside my installed Windows 8.1:

You see that this picture has some pixels which unlikely can be compressed very good (around center), but there are long one-color lines at top and bottom. Indeed, lines like these also looks as lines during viewing the file:

5.10.2 Memory “snapshots” comparing

The technique of the straightforward comparison of two memory snapshots in order to see changes was often used to hack 8-bit computer games and for hacking “high score” files.

For example, if you had a loaded game on an 8-bit computer (there isn’t much memory on these, but the game usually consumes even less memory) and you know that you have now, let’s say, 100 bullets,
can do a “snapshot” of all memory and back it up to some place. Then shoot once, the bullet count goes to 99, do a second “snapshot” and then compare both: it must be a byte somewhere which has been 100 at the beginning, and now it is 99.

Considering the fact that these 8-bit games were often written in assembly language and such variables were global, it can be said for sure which address in memory has holding the bullet count. If you searched for all references to the address in the disassembled game code, it was not very hard to find a piece of code decrementing the bullet count, then to write a NOP instruction there, or a couple of NOP-s, and then have a game with 100 bullets forever. Games on these 8-bit computers were commonly loaded at the constant address, also, there were not much different versions of each game (commonly just one version was popular for a long span of time), so enthusiastic gamers knew which bytes must be overwritten (using the BASIC’s instruction POKE) at which address in order to hack it. This led to “cheat” lists that contained POKE instructions, published in magazines related to 8-bit games.

Likewise, it is easy to modify “high score” files, this does not work with just 8-bit games. Notice your score count and back up the file somewhere. When the “high score” count gets different, just compare the two files, it can even be done with the DOS utility FC ("high score" files are often in binary form).

There will be a point where a couple of bytes are different and it is easy to see which ones are holding the score number. However, game developers are fully aware of such tricks and may defend the program against it.

Somewhat similar example in this book is: 9.3 on page 942.

A real story from 1999

There was a time of ICQ messenger’s popularity, at least in ex-USSR countries. The messenger had a peculiarity — some users didn’t want to share their online status with everyone. And you had to ask an authorization from that user. That user could allow you seeing his/her status, or maybe not.

This is what the author of these lines did:

• Added a user.
  • A user appeared in a contact-list, in a “wait for authorization” section.
  • Closed ICQ.
  • Backed up the ICQ database.
  • Loaded ICQ again.
  • User authorized.
  • Closed ICQ and compared two databases.

It turned out: two database differed by only one byte. In the first version: RESU\x03, in the second: RESU\x02. ("RESU", presumably, means “USER”, i.e., a header of a structure where all the information about user was stored.) That means the information about authorization was stored not at the server, but at the client. Presumably, 2/3 value reflected authorization status.

Windows registry

It is also possible to compare the Windows registry before and after a program installation.

It is a very popular method of finding which registry elements are used by the program. Perhaps, this is the reason why the “windows registry cleaner” shareware is so popular.

By the way, this is how to dump Windows registry to text files:

```
reg export HKLM HKLM.reg
reg export HKCU HKCU.reg
reg export HKCR HKCR.reg
reg export HKU HKU.reg
reg export HKCC HKCC.reg
```

They can be compared using diff...

---

24MS-DOS utility for comparing binary files
Engineering software, CADs, etc

If a software uses proprietary files, you can also investigate something here as well. You save file. Then you add a dot or line or another primitive. Save file, compare. Or move dot, save file, compare.

Blink-comparator

Comparison of files or memory snapshots remind us blink-comparator \(^{25}\): a device used by astronomers in past, intended to find moving celestial objects.

Blink-comparator allows to switch quickly between two photographies shot in different time, so astronomer would spot the difference visually.

By the way, Pluto was discovered by blink-comparator in 1930.

5.11 ISA detection

Often, you can deal with a binary file for an unknown ISA. Perhaps, easiest way to detect ISA is to try various ones in IDA, objdump or another disassembler.

To achieve this, one should understand a difference between incorrectly disassembled code and correctly one.

5.11.1 Incorrectly disassembled code

Practicing reverse engineers often have to deal with incorrectly disassembled code.

Disassembling from an incorrect start (x86)

Unlike ARM and MIPS (where any instruction has a length of 2 or 4 bytes), x86 instructions have variable size, so any disassembler that starts in the middle of a x86 instruction may produce incorrect results.

As an example:

```
add [ebp-31F7Bh], cl
dec dword ptr [ecx-3277Bh]
dec dword ptr [ebp-2CF7Bh]
ing dword ptr [ebx-7A76F33Ch]
fdiv st(4), st
db 0FFh
dec dword ptr [ecx-21F7Bh]
dec dword ptr [ecx-22373h]
dec dword ptr [ecx-2276Bh]
dec dword ptr [ecx-22B63h]
dec dword ptr [ecx-22F4Bh]
dec dword ptr [ecx-23343h]
jmp dword ptr [esi-74h]
xchg eax, ebp
clc
std
db 0FFh
db 0FFh
mov word ptr [ebp-214h], cs ; <-- disassembler finally found right track here
mov word ptr [ebp-238h], ds
mov word ptr [ebp-23Ch], es
mov word ptr [ebp-240h], fs
mov word ptr [ebp-244h], gs
pushf
pop dword ptr [ebp-210h]
mov eax, [ebp+4]
mov [ebp-218h], eax
lea eax, [ebp+4]
mov [ebp-20Ch], eax
mov dword ptr [ebp-200h], 10001h
mov eax, [eax-4]
mov [ebp-21Ch], eax
```

\(^{25}\)http://go.yurichev.com/17348
There are incorrectly disassembled instructions at the beginning, but eventually the disassembler gets on the right track.

**How does random noise looks disassembled?**

Common properties that can be spotted easily are:

- Unusually big instruction dispersion. The most frequent x86 instructions are PUSH, MOV, CALL, but here we see instructions from all instruction groups: FPU instructions, IN/OUT instructions, rare and system instructions, everything mixed up in one single place.
- Big and random values, offsets and immediates.
- Jumps having incorrect offsets, often jumping in the middle of another instructions.

Listing 5.7: random noise (x86)
Listing 5.8: random noise (x86-64)

```
lea    esi, [rax+rdx*4+43558D29h]

loc_AF3: ; CODE XREF: seg000:0000000000000B46
    rcl    byte ptr [rsi+rax*8+29BB423Ah], 1
    lea    ecx, cs:0FFFFFFFFFB2A6780Fh
    mov    al, 96h
    mov    ah, 0CEh
    push   rsp
    lods   byte ptr [esi]
    db     2Fh ; /
    pop    rsp
    db     64h
    retf   0E9993h
    cmp    ah, [rax+4Ah]
    movzx   rsi, dword ptr [rbp-25h]
    push    4Ah
    movzx   rdi, dword ptr [rdi+rdx*8]
    db     9Ah
    rcr    byte ptr [rax+1Dh], cl
    lodsd
    xor    [rbp+6CF20173h], edx
    xor    [rbp+66F8B593h], edx
    push   rbx
    sbb    ch, [rbx-0Fh]
    stosd
    int    87h
    db     46h, 4Ch
    out    33h, rax
    xchg   eax, ebp
    test   ecx, ebp
    movsd
```
leave
push rsp
db 16h

xchg eax, esi
pop rdi

loc_B3D: ; CODE XREF: seg000:0000000000000B5F
mov ds:39C685DF9B9A9F9h, eax
jnz short near ptr loc_AF3+6
out dx, eax
cwde
mov bh, 5Dh ; '}
movsb
pop rbp

Listing 5.9: random noise (ARM (ARM mode))
Listing 5.10: random noise (ARM (Thumb mode))

```
LSRS R3, R6, #0x12
LDRH R1, [R7,#0x2C]
SUBS R0, #0x55, 'U'
ADR R1, loc_3C
LDR R2, [SP,#0x218]
CMP R4, #0x86
SXTB R7, R4
LDR R1, [R7,#0x2C]
SUBS R0, #0x55, 'U'
ADR R1, loc_3C
LDR R2, [SP,#0x218]
CMP R4, #0x86
SXTB R7, R4
LDR R4, [R1,#0x2C4]
```

Listing 5.11: random noise (MIPS little endian)

```
lw  $t9, 0xCB3($t5)
sb  $t5, 0x3855($t0)
sltiu $a2, $a0, -0x657A
ldr  $t4, -0x4D99($a2)
daddi $s0, $s1, 0
lw  $s7, -0x2353($s4)
bgtzl $a1, 0

.loc_3C ; DATA XREF: ROM:00000006
   B 0xFFFFFFF98

.lw  $t8, -0x2CD4($s2)  # K
 .byte 0x72
 .byte 0x17
 .byte 0xED
 .byte 0x4B
 .byte 0x54

.lwc2 $31, 0x66C5($sp)
lwu  $s1, 0x10D3($a1)
ldr  $t6, -0x204B($zero)
lwc1 $f30, 0x4DBE($s2)
daddiu $t1, $s1, 0x60D9
lwu  $s5, -0x2C64($v1)
cop0 0x13D642D
bne  $gp, $t4, 0xFFFF9EF0
lh  $ra, 0x1819($s1)
sdl  $fp, -0x6474($t8)
jal  0x78C0050
ori  $v0, $s2, 0xC634
blez  $gp, 0xFFFFFEA904
swl  $t8, -0x2CD4($s2)
sltiu $a1, $k0, 0x685
sdc1 $f15, 0x5964($at)
sw  $s0, -0x19A6($a1)
```
It is also important to keep in mind that cleverly constructed unpacking and decryption code (including self-modifying) may looks like noise as well, but still execute correctly.

5.11.2 Correctly disassembled code

Each ISA has a dozen of a most used instructions, all the rest are used much less often.

As of x86, it is interesting to know that the fact that function calls (PUSH/CALL/ADD) and MOV instructions are the most frequently executed pieces of code in almost all programs we use. In other words, CPU is very busy passing information between levels of abstractions, or, it can be said, it’s very busy switching between these levels. Regardless type of ISA. This is a cost of splitting problems into several levels of abstractions (so humans could work with them easier).

5.12 Other things

5.12.1 General idea

A reverse engineer should try to be in programmer’s shoes as often as possible. To take his/her viewpoint and ask himself, how would one solve some task the specific case.

5.12.2 Order of functions in binary code

All functions located in a single .c or .cpp-file are compiled into corresponding object (.o) file. Later, a linker puts all object files it needs together, not changing order of functions in them. As a consequence, if you see two or more consecutive functions, it means, that they were placed together in a single source code file (unless you’re on border of two object files, of course.) This means these functions have something in common, that they are from the same API level, from the same library, etc.

This is a real story from practice: once upon a time, the author searched for Twofish-related functions in a program with CryptoPP library linked, especially encryption/decryption functions. I found the Twofish::Base::UncheckedSetKey() function, but not others. After peeking into the twofish.cpp source code, it became clear that all functions are located in one module (twofish.cpp). So I tried all function that followed Twofish::Base::UncheckedSetKey() — as it happened, one was Twofish::Enc::ProcessAndXorBlock(), another — Twofish::Dec::ProcessAndXorBlock().

5.12.3 Tiny functions

Tiny functions like empty functions (1.3 on page 5) or function which returns just “true” (1) or “false” (0) (1.4 on page 7) are very common, and almost all decent compilers tend to put only one such function into resulting executable code even if there were several similar functions in source code. So, whenever you see a tiny function consisting just of mov eax, 1 / ret which is referenced (and can be called) from many places, which are seems unconnected to each other, this may be a result of such optimization.

5.12.4 C++

RTTI (3.21.1 on page 557)-data may be also useful for C++ class identification.

5.12.5 Crash on purpose

Often you need to know, which function has been executed, and which is not. You can use a debugger, but on exotic architectures there may not be the one, so easiest way is to put there an invalid opcode, or something like INT3 (0xCC). The crash would signal about the very fact this instruction has been executed.

Another example of crashing on purpose: 3.23.4 on page 606.

---

26 https://github.com/weidai11/cryptopp/blob/b613522794a7633aa2bd81932a98a0b0a51bc04f/twofish.cpp
Chapter 6

OS-specific

6.1 Arguments passing methods (calling conventions)

6.1.1 cdecl

This is the most popular method for passing arguments to functions in the C/C++ languages. The glscaller also must return the value of the stack pointer (ESP) to its initial state after the callee function exits.

Listing 6.1: cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; returns ESP
```

6.1.2 stdcall

It’s almost the same as cdecl, with the exception that the callee must set ESP to the initial state by executing the RET x instruction instead of RET, where x = arguments number * sizeof(int). The caller is not adjusting the stack pointer, there are no add esp, x instruction.

Listing 6.2: stdcall

```
push arg3
push arg2
push arg1
call function

function:
  ... do something ...
ret 12
```

The method is ubiquitous in win32 standard libraries, but not in win64 (see below about win64).

For example, we can take the function from 1.90 on page 97 and change it slightly by adding the __stdcall modifier:

```
int __stdcall f2 (int a, int b, int c)
{
    return a*b+c;
};
```

It is to be compiled in almost the same way as 1.91 on page 97, but you will see RET 12 instead of RET. SP is not updated in the caller.

As a consequence, the number of function arguments can be easily deduced from the RETN n instruction: just divide n by 4.

\footnote{The size of an \textit{int} type variable is 4 in x86 systems and 8 in x64 systems}
Functions with variable number of arguments

printf()-like functions are, probably, the only case of functions with a variable number of arguments in C/C++, but it is easy to illustrate an important difference between cdecl and stdcall with their help. Let’s start with the idea that the compiler knows the argument count of each printf() function call.

However, the called printf(), which is already compiled and located in MSVCR71.DLL (if we talk about Windows), does not have any information about how much arguments were passed, however it can determine it from the format string.

Thus, if printf() would be a stdcall function and restored stack pointer to its initial state by counting the number of arguments in the format string, this could be a dangerous situation, when one programmer’s typo can provoke a sudden program crash. Thus it is not suitable for such functions to use stdcall, cdecl is better.

6.1.3fastcall

That’s the general naming for the method of passing some arguments via registers and the rest via the stack. It worked faster than cdecl/stdcall on older CPUs (because of smaller stack pressure). It may not help to gain any significant performance on latest (much more complex) CPUs, however.

It is not standardized, so the various compilers can do it differently. It’s a well known caveat: if you have two DLLs and the one uses another one, and they are built by different compilers with differentfastcall calling conventions, you can expect problems.

Both MSVC and GCC pass the first and second arguments via ECX and EDX and the rest of the arguments via the stack.

The stack pointer must be restored to its initial state by the callee (like in stdcall).
```c
int __fastcall f3 (int a, int b, int c)
{
    return a*b+c;
}
```

Here is how it is to be compiled:

Listing 6.5: Optimizing MSVC 2010 /Ob0

```assembly
; c$ = 8 ; size = 4
@f3@12 PROC
; _a$ = ecx
; _b$ = edx
    mov     eax, ecx
    imul    eax, edx
    add     eax, DWORD PTR _c$[esp-4]
    ret
@f3@12 ENDP
; ...
    mov     edx, 2
    push    3
    lea     ecx, DWORD PTR [edx-1]
    call    @f3@12
    push    eax
    push    OFFSET $SG81390
    call    _printf
    add     esp, 8
```

We see that the callee returns SP by using the RETN instruction with an operand.
Which implies that the number of arguments can be deduced easily here as well.

**GCC regparm**

It is the evolution offastcall\(^2\) in some sense. With the -mregparm option it is possible to set how many arguments are to be passed via registers (3 is the maximum). Thus, the EAX, EDX and ECX registers are to be used.

Of course, if the number of arguments is less than 3, not all 3 registers are to be used.
The caller restores the stack pointer to its initial state.
For example, see (1.28.1 on page 305).

**Watcom/OpenWatcom**

Here it is called “register calling convention”. The first 4 arguments are passed via the EAX, EDX, EBX and ECX registers. All the rest—via the stack.
These functions have an underscore appended to the function name in order to distinguish them from those having a different calling convention.

### 6.1.4 thiscall

This is passing the object’s this pointer to the function-method, in C++.
In MSVC, this is usually passed in the ECX register.
In GCC, the this pointer is passed as the first function-method argument. Thus it will be visible that all functions in assembly code have an extra argument, in comparison with the source code.
For an example, see (3.21.1 on page 542).

\(^2\)http://go.yurichev.com/17040
6.1.5 x86-64

Windows x64

The method of for passing arguments in Win64 somewhat resembles fastcall. The first 4 arguments are passed via RCX, RDX, R8 and R9, the rest—via the stack. The caller also must prepare space for 32 bytes or 4 64-bit values, so then the callee can save there the first 4 arguments. Short functions may use the arguments' values just from the registers, but larger ones may save their values for further use.

The caller also must return the stack pointer into its initial state.

This calling convention is also used in Windows x86-64 system DLLs (instead of stdcall in win32).

Example:

```c
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
    printf("%d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};

int main()
{
    f1(1,2,3,4,5,6,7);
};
```

Listing 6.6: MSVC 2012 /0b

```assembly
S5G2937 DB '%d %d %d %d %d %d', 0aH, 00H

main PROC
    sub rsp, 72
    mov DWORD PTR [rsp+48], 7
    mov DWORD PTR [rsp+40], 6
    mov DWORD PTR [rsp+32], 5
    mov r9d, 4
    mov r8d, 3
    mov edx, 2
    mov ecx, 1
    call f1
    xor eax, eax
    add rsp, 72
    ret 0

main ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1 PROC
$LN3:
    mov DWORD PTR [rsp+32], r9d
    mov DWORD PTR [rsp+24], r8d
    mov DWORD PTR [rsp+16], edx
    mov DWORD PTR [rsp+8], ecx
    sub rsp, 72
    mov eax, DWORD PTR g$[rsp]
    mov DWORD PTR [rsp+56], eax
    mov eax, DWORD PTR f$[rsp]
    mov DWORD PTR [rsp+48], eax
    mov eax, DWORD PTR e$[rsp]
    mov DWORD PTR [rsp+40], eax
    mov eax, DWORD PTR d$[rsp]
    mov DWORD PTR [rsp+32], eax
    mov r9d, DWORD PTR c$[rsp]
```

728
Here we clearly see how 7 arguments are passed: 4 via registers and the remaining 3 via the stack. The code of the f1() function’s prologue saves the arguments in the “scratch space”—a space in the stack intended exactly for this purpose.

This is arranged so because the compiler cannot be sure that there will be enough registers to use without these 4, which will otherwise be occupied by the arguments until the function’s execution end.

The “scratch space” allocation in the stack is the caller’s duty.

If we compile the example with optimizations, it is to be almost the same, but the “scratch space” will not be used, because it won’t be needed.
Also take a look on how MSVC 2012 optimizes the loading of primitive values into registers by using LEA (.1.6 on page 1014). MOV would be 1 byte longer here (5 instead of 4).

Another example of such thing is: 8.2.1 on page 789.

**Windows x64: Passing this (C/C++)**

The this pointer is passed in RCX, the first argument of the method is in RDX, etc. For an example see: 3.21.1 on page 544.

**Linux x64**

The way arguments are passed in Linux for x86-64 is almost the same as in Windows, but 6 registers are used instead of 4 (RDI, RSI, RDX, RCX, R8, R9) and there is no “scratch space”, although the callee may save the register values in the stack, if it needs/wants to.

**Listing 6.8: Optimizing GCC 4.7.3**

```c
.LC0:
   .string "%d %d %d %d %d %d\n"
f1:
   sub   rsp, 40
   mov   eax, DWORD PTR [rsp+48]
   mov   DWORD PTR [rsp+8], r9d
   mov   r9d, ecx
   mov   DWORD PTR [rsp], r8d
   mov   ecx, esi
   mov   r8d, edx
   mov   esi, OFFSET FLAT:.LC0
   mov   edx, edi
   mov   edi, 1
   mov   DWORD PTR [rsp+16], eax
   xor   eax, eax
   call  __printf_chk
   add   rsp, 40
   ret
main:
   sub   rsp, 24
   mov   r9d, 6
   mov   r8d, 5
   mov   DWORD PTR [rsp], 7
   mov   ecx, 4
   mov   edx, 3
   mov   esi, 2
   mov   edi, 1
   call  f1
   add   rsp, 24
   ret
```

N.B.: here the values are written into the 32-bit parts of the registers (e.g., EAX) but not in the whole 64-bit register (RAX). This is because each write to the low 32-bit part of a register automatically clears the high 32 bits. Supposedly, it was decided in AMD to do so to simplify porting code to x86-64.

**6.1.6 Return values of float and double type**

In all conventions except in Win64, the values of type float or double are returned via the FPU register ST(0).

In Win64, the values of float and double types are returned in the low 32 or 64 bits of the XMM0 register.

**6.1.7 Modifying arguments**

Sometimes, C/C++ programmers (not limited to these PLs, though), may ask, what can happen if they modify the arguments?

The answer is simple: the arguments are stored in the stack, that is where the modification takes place.
The calling functions is not using them after the callee’s exit (the author of these lines has never seen any such case in his practice).

```c
#include <stdio.h>

void f(int a, int b)
{
    a=a+b;
    printf ("%d\n", a);
}
```

Listing 6.9: MSVC 2012

| a$ = 8 | ; size = 4 |
| b$ = 12| ; size = 4 |

```assembly
    _f PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$ [ebp]
    add    eax, DWORD PTR _b$ [ebp]
    mov    DWORD PTR _a$ [ebp], eax
    mov    ecx, DWORD PTR _a$ [ebp]
    push   ecx
    push OFFSET $SG2938 ; '\%d', 0aH
    call   _printf
    add    esp, 8
    pop    ebp
    ret 0
_f ENDP
```

So yes, one can modify the arguments easily. Of course, if it is not references in C++ (3.21.3 on page 558), and if you don’t modify data to which a pointer points to, then the effect will not propagate outside the current function.

Theoretically, after the callee’s return, the caller could get the modified argument and use it somehow. Maybe if it is written directly in assembly language.

For example, code like this will be generated by usual C/C++ compiler:

```c
push 456 ; will be b
push 123 ; will be a
call f ; f() modifies its first argument
add esp, 2*4
```

We can rewrite this code like:

```c
push 456 ; will be b
push 123 ; will be a
call f ; f() modifies its first argument
pop eax
add esp, 4
; EAX=1st argument of f() modified in f()
```

Hard to imagine, why anyone would need this, but this is possible in practice. Nevertheless, the C/C++ languages standards don’t offer any way to do so.

### 6.1.8 Taking a pointer to function argument

...even more than that, it’s possible to take a pointer to the function’s argument and pass it to another function:

```c
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
    modify_a (&a);
    printf ("%d\n", a);
}
```
It’s hard to understand how it works until we can see the code:

### Listing 6.10: Optimizing MSVC 2010

```assembly
$SG2796 DB 'd', 0AH, 00H
_a$ = 8
_f PROC
  lea eax, DWORD PTR _a$[esp-4] ; just get the address of value in local stack
  push eax ; and pass it to modify_a()
  call _modify_a
  mov ecx, DWORD PTR _a$[esp] ; reload it from the local stack
  push ecx ; and pass it to printf()
  push OFFSET $SG2796 ; '%'d'
  call _printf
  add esp, 12
  ret 0
_f ENDP
```

The address of the place in the stack where \( a \) has been passed is just passed to another function. It modifies the value addressed by the pointer and then printf() prints the modified value.

The observant reader might ask, what about calling conventions where the function’s arguments are passed in registers?

That’s a situation where the **Shadow Space** is used.

The input value is copied from the register to the Shadow Space in the local stack, and then this address is passed to the other function:

### Listing 6.11: Optimizing MSVC 2012 x64

```assembly
$SG2994 DB 'd', 0AH, 00H
a$ = 48
_f PROC
  mov DWORD PTR [rsp+8], ecx ; save input value in Shadow Space
  sub rsp, 40
  lea rcx, DWORD PTR a$[rsp] ; get address of value and pass it to modify_a()
  call modify_a
  mov edx, DWORD PTR a$[rsp] ; reload value from Shadow Space and pass it to
  printf()
  lea rcx, OFFSET FLAT:$SG2994 ; '%'d'
  call _printf
  add rsp, 40
  ret 0
_f ENDP
```

GCC also stores the input value in the local stack:

### Listing 6.12: Optimizing GCC 4.9.1 x64

```assembly
.LC0:
.string "%d\n"
_f:
  sub rsp, 24
  mov DWORD PTR [rsp+12], edi ; store input value to the local stack
  lea rdi, [rsp+12] ; take an address of the value and pass it to
modify_a()
  call modify_a
  mov edx, DWORD PTR [rsp+12] ; reload value from the local stack and pass it to
printf()
  mov esi, OFFSET FLAT:.LC0 ; '%'d'
  mov edi, 1
  xor eax, eax
  call __printf_chk
  add rsp, 24
  ret
```

GCC for ARM64 does the same, but this space is called **Register Save Area** here:
Listing 6.13: Optimizing GCC 4.9.1 ARM64

f:
  stp  x29, x30, [sp, -32]!
  add  x29, sp, 0       ; setup FP
  add  x1, x29, 32      ; calculate address of variable in Register Save Area
  str  w0, [x1, -4]!    ; store input value there
  mov  x0, x1           ; pass address of variable to the modify_a()
  bl   modify_a
  ldr  w1, [x29, 28]     ; load value from the variable and pass it to printf()
  adrp x0, .LC0         ; '%d'
  add  x0, x0, :lo12:.LC0
  bl   printf           ; call printf()
  ldp  x29, x30, [sp], 32
  ret

.LC0:
  .string "%d\n"

By the way, a similar usage of the Shadow Space is also considered here: 3.17.1 on page 519.

6.1.9 Python ctypes problem (x86 assembly homework)

A Python ctypes module can call external functions in DLLs, .so’s, etc. But calling convention (for 32-bit environment) must be specified explicitly:

"ctypes" exports the *cdll*, and on Windows *windll* and *oledll* objects, for loading dynamic link libraries.

You load libraries by accessing them as attributes of these objects. *cdll* loads libraries which export functions using the standard "cdecl" calling convention, while *windll* libraries call functions using the "stdcall" calling convention.

(https://docs.python.org/3/library/ctypes.html)

In fact, we can modify ctypes module (or any other caller code), so that it will successfully call external cdecl or stdcall functions, without knowledge, which is where. (Number of arguments, however, is to be specified).

This is possible to solve using maybe 5-10 x86 assembly instructions in caller. Try to find out these.

6.1.10 Cdecl example: a DLL

Let’s back to the fact that this is not very important how to declare the main() function: 1.9.2 on page 32.

This is a real story: once upon a time I wanted to replace an original DLL file in some software by mine. First I enumerated names of all DLL exports and made a function in my own replacement DLL for each function in the original DLL, like:

```c
void function1 ()
{
    write_to_log ("function1() called\n");
};
```

I wanted to see, which functions are called during run, and when. However, I was in hurry and had no time to deduce arguments count for each function, let alone data types. So each function in my replacement DLL had no argument whatsoever. But everything worked, because all functions had cdecl calling convention. (It wouldn’t work if functions had stdcall calling convention.) It also worked for x64 version.

And then I did a next step: I deduced argument types for some functions. But I made several mistakes, for example, the original function took 3 arguments, but I knew only about 2, etc.

Still, it worked. At the beginning, my replacement DLL just ignored all arguments. Then, it ignored the 3rd argument.
### 6.2 Thread Local Storage

TLS is a data area, specific to each thread. Every thread can store what it needs there. One well-known example is the C standard global variable `errno`.

Multiple threads may simultaneously call functions which return an error code in `errno`, so a global variable will not work correctly here for multi-threaded programs, so `errno` must be stored in the TLS.

In the C++11 standard, a new `thread_local` modifier was added, showing that each thread has its own version of the variable, it can be initialized, and it is located in the TLS.

```cpp
#include <iostream>
#include <thread>

thread_local int tmp = 3;

int main() {
    std::cout << tmp << std::endl;
}
```

Compiled in MinGW GCC 4.8.1, but not in MSVC 2012.

If we talk about PE files, in the resulting executable file, the `tmp` variable is to be allocated in the section devoted to the TLS.

#### 6.2.1 Linear congruential generator revisited

The pseudorandom number generator we considered earlier 1.29 on page 336 has a flaw: it's not thread-safe, because it has an internal state variable which can be read and/or modified in different threads simultaneously.

**Win32**

**Uninitialized TLS data**

One solution is to add `__declspec( thread )` modifier to the global variable, then it will be allocated in the TLS (line 9):

```cpp
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
define RNG_a 1664525
define RNG_c 1013904223
__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init) {
    rand_state=init;
}

int my_rand () {
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main() {
    my_srand(0x12345678);
    printf("%d\n", my_rand());
}
```

\(^3\) C11 also has thread support, optional though.
Hiew shows us that there is a new PE section in the executable file: `.tls`.

Listings 6.15: Optimizing MSVC 2013 x86

```assembly
_TLS SEGMENT
_rand_state DD 01H DUP (?)
_TLS ENDS
_DATA SEGMENT
$SG84851 DB 'd', 0aH, 00H
_DATA ENDS
_TEXT SEGMENT

_my_srand PROC ; FS:0=address of TIB
mov eax, DWORD PTR fs:_tls_array ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
mov ecx, DWORD PTR _tls_index
mov ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
mov eax, DWORD PTR _init$[esp-4]
mov DWORD PTR _rand_state[ecx], eax
ret 0
_my_srand ENDP

_my_rand PROC ; FS:0=address of TIB
mov eax, DWORD PTR fs:_tls_array ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
mov ecx, DWORD PTR _tls_index
mov ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
imul eax, DWORD PTR _rand_state[ecx], 1664525
add eax, 1013904223 ; 3c6ef35fH
mov DWORD PTR _rand_state[ecx], eax
and eax, 32767 ; 00007fffH
ret 0
_my_rand ENDP

_TEXT ENDS
```

`rand_state` is now in the TLS segment, and each thread has its own version of this variable.

Here is how it's accessed: load the address of the TIB from FS:2Ch, then add an additional index (if needed), then calculate the address of the TLS segment.

Then it's possible to access the `rand_state` variable through the ECX register, which points to an unique area in each thread.

The FS selector is familiar to every reverse engineer, it is specially used to always point to TIB, so it would be fast to load the thread-specific data.

The GS selector is used in Win64 and the address of the TLS is 0x58:

Listings 6.16: Optimizing MSVC 2013 x64

```assembly
_TLS SEGMENT
_rand_state DD 01H DUP (?)
_TLS ENDS
_DATA SEGMENT
$SG85451 DB 'd', 0aH, 00H
_DATA ENDS
_TEXT SEGMENT

_my_srand PROC
mov edx, DWORD PTR _tls_index
```
Initialized TLS data

Let's say, we want to set some fixed value to rand_state, so in case the programmer forgets to, the rand_state variable would be initialized to some constant anyway (line 9):

```c
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec(thread) uint32_t rand_state = 1234;

void my_srand(uint32_t init)
{
    rand_state = init;
}

int my_rand()
{
    rand_state = rand_state*RNG_a;
    rand_state = rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    printf("%d\n", my_rand());
}
```

The code is not different from what we already saw, but in IDA we see:

```
...736
```

### TLS Segment

- **.tls:** 00404000
  - Segment type: Pure data
  - Segment permissions: Read/Write
  - _tls_ segment para public 'DATA' use32
  - assume cs: _tls
  - ;org 404000h
  - TlsStart db 0 ; DATA XREF: .rdata:TlsDirectory
  - TlsEnd db 0
  - TlsEnd_ptr db 0
  - ...
1234 is there and every time a new thread starts, a new TLS is allocated for it, and all this data, including 1234, will be copied there.

This is a typical scenario:

• Thread A is started. A TLS is created for it, 1234 is copied to rand_state.
  • The my_rand() function is called several times in thread A. rand_state is different from 1234.
• Thread B is started. A TLS is created for it, 1234 is copied to rand_state, while thread A has a different value in the same variable.

**TLS callbacks**

But what if the variables in the TLS have to be filled with some data that must be prepared in some unusual way?

Let’s say, we’ve got the following task: the programmer can forget to call the my_srand() function to initialize the PRNG, but the generator has to be initialized at start with something truly random, instead of 1234. This is a case in which TLS callbacks can be used.

The following code is not very portable due to the hack, but nevertheless, you get the idea.

What we do here is define a function (tls_callback()) which is to be called before the process and/or thread start.

The function initializes the PRNG with the value returned by GetTickCount() function.

```c
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223
__declspec(thread) uint32_t rand_state;

void my_srand(uint32_t init)
{
    rand_state = init;
}

void WINAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
    my_srand (GetTickCount());
}

#pragma data SEG(\".CRT$XLB\")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data SEG()

int my_rand()
{
    rand_state = rand_state*RNG_a;
    rand_state = rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    // rand_state is already initialized at the moment (using GetTickCount())
    printf ("%d\n", my_rand());
}
```

Let’s see it in IDA:

```plaintext
Listing 6.17: Optimizing MSVC 2013
.text:00401020 TlsCallback_0 proc near ; DATA XREF: .rdata:TlsCallbacks
.text:00401020 call ds:GetTickCount
```
TLS callback functions are sometimes used in unpacking routines to obscure their processing. Some people may be confused and be in the dark that some code executed right before the OEP⁴.

**Linux**

Here is how a thread-local global variable is declared in GCC:

```c
__thread uint32_t rand_state=1234;
```

This is not the standard C/C++ modifier, but a rather GCC-specific one⁵.

The GS: selector is also used to access the TLS, but in a somewhat different way:

```
Listing 6.18: Optimizing GCC 4.8.1 x86
```

```
.text:00401026 push eax
.text:00401027 call my_srand
.text:0040102C pop ecx
.text:0040102D retn 0Ch
.text:0040102D TlsCallback_0 endp

... rdata:004020C0 TlsCallbacks dd offset TlsCallback_0 ; DATA XREF: .rdata:TlsCallbacks_ptr
... rdata:00402118 TlsDirectory dd offset TlsStart
.rdata:0040211C TlsEnd_ptr dd offset TlsEnd
.rdata:00402120 TlsIndex_ptr dd offset TlsIndex
.rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata:00402128 TlsSizeOfZeroFill dd 0
.rdata:0040212C TlsCharacteristics dd 300000h
```


### 6.3 System calls (syscall-s)

As we know, all running processes inside an OS are divided into two categories: those having full access to the hardware (“kernel space”) and those that do not (“user space”).

The OS kernel and usually the drivers are in the first category.

All applications are usually in the second category.

For example, Linux kernel is in kernel space, but Glibc in user space.

---

⁴Original Entry Point
⁵[http://go.yurichev.com/17062](http://go.yurichev.com/17062)
⁶Also available as [http://go.yurichev.com/17272](http://go.yurichev.com/17272)
This separation is crucial for the safety of the OS: it is very important not to give to any process the possibility to screw up something in other processes or even in the OS’s kernel. On the other hand, a failing driver or error inside the OS’s kernel usually leads to a kernel panic or BSOD\(^7\).

The protection in the x86 processors allows to separate everything into 4 levels of protection (rings), but both in Linux and in Windows only two are used: ring0 (“kernel space”) and ring3 (“user space”).

System calls (syscall-s) are a point where these two areas are connected.

It can be said that this is the main API provided to applications.

As in Windows NT, the sycalls table resides in the SSDT\(^8\).

The usage of sycalls is very popular among shellcode and computer viruses authors, because it is hard to determine the addresses of needed functions in the system libraries, but it is easier to use sycalls. However, much more code has to be written due to the lower level of abstraction of the API.

It is also worth noting that the syscall numbers may be different in various OS versions.

### 6.3.1 Linux

In Linux, a syscall is usually called via \texttt{int 0x80}. The call’s number is passed in the EAX register, and any other parameters —in the other registers.

#### Listing 6.19: A simple example of the usage of two sycalls

```
section .text
global _start

_start:
    mov edx,len ; buffer len
    mov ecx,msg ; buffer
    mov ebx,1 ; file descriptor. 1 is for stdout
    mov eax,4 ; syscall number. 4 is for sys_write
    int 0x80

    mov eax,1 ; syscall number. 1 is for sys_exit
    int 0x80

section .data

msg     db 'Hello, world!',0xa
len     equ $ - msg
```

Compilation:
```
nasm -f elf32 1.s
ld 1.o
```

The full list of sycalls in Linux: \url{http://go.yurichev.com/17319}.

For system calls interception and tracing in Linux, strace(\textit{7.2.3 on page 781}) can be used.

### 6.3.2 Windows

Here they are called via \texttt{int 0x2e} or using the special x86 instruction SYSENTER.

The full list of sycalls in Windows: \url{http://go.yurichev.com/17320}.

Further reading:
“Windows Syscall Shellcode” by Piotr Bania: \url{http://go.yurichev.com/17321}.

### 6.4 Linux

#### 6.4.1 Position-independent code

While analyzing Linux shared (.so) libraries, one may frequently spot this code pattern:

\(^7\)Blue Screen of Death
\(^8\)System Service Dispatch Table
Listing 6.20: libc-2.17.so x86

.text:0012D5E3 __x86_get_pc_thunk_bx proc near ; CODE XREF: sub_17350+3
.text:0012D5E3 ; sub_173CC+4 ...
.text:0012D5E3 mov ebx, [esp+0]
.text:0012D5E6 retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp

... .text:000576C0 sub_576C0 proc near ; CODE XREF: tmpfile+73 ...
.text:000576C0 push ebp
.text:000576C1 mov ecx, large gs:0
.text:000576C8 push edi
.text:000576C9 push esi
.text:000576CA push ebx
.text:000576CB call __x86_get_pc_thunk_bx
.text:000576D0 add ebx, 157930h
.text:000576D6 sub esp, 9Ch

... .text:000579F0 lea eax, (a__gen_tempname - 1AF000h)[ebx]; "__gen_tempname"
.text:000579F6 mov [esp+OAH+var_A0], eax
.text:000579FA lea eax, (a_SysdepsPosix - 1AF000h)[ebx]; "!/sysdeps/posix/tempname.c"
.text:00057A00 mov [esp+OAH+var_A8], eax
.text:00057A04 lea eax, (aInvalidKindIn_ - 1AF000h)[ebx]; "! "invalid KIND in __gen_tempname]\"
.text:00057A0A mov [esp+OAH+var_A4], 14Ah
.text:00057A12 mov [esp+OAH+var_AC], eax
.text:00057A15 call __assert_fail

All pointers to strings are corrected by some constants and the value in EBX, which is calculated at the beginning of each function.

This is the so-called PIC, it is intended to be executable if placed at any random point of memory, that is why it cannot contain any absolute memory addresses.

PIC was crucial in early computer systems and is still crucial today in embedded systems without virtual memory support (where all processes are placed in a single continuous memory block).

It is also still used in *NIX systems for shared libraries, since they are shared across many processes while loaded in memory only once. But all these processes can map the same shared library at different addresses, so that is why a shared library has to work correctly without using any absolute addresses.

Let's do a simple experiment:

#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf("returning %d\n", rt);
    return rt;
}

Let's compile it in GCC 4.7.3 and see the resulting .so file in IDA:

gcc -fPIC -shared -O3 -o 1.so 1.c

Listing 6.21: GCC 4.7.3

.text:00000440 public __x86_get_pc_thunk_bx
.text:00000440 __x86_get_pc_thunk_bx proc near ; CODE XREF: _init proc+4
.text:00000440 ; deregister_tm_clones+4 ...
.text:00000440 mov ebx, [esp+0]
That's a simple way to get the value of the program counter (\texttt{\_\_x86\_get\_pc\_thunk\_bx})

That's it: the pointers to \texttt{\_\_x86\_get\_pc\_thunk\_bx} are to be corrected at each function execution.

The \texttt{\_\_x86\_get\_pc\_thunk\_bx()} function returns in EBX the address of the point after a call to itself (0x57C here).

That's a simple way to get the value of the program counter (EIP) at some point. The 0x1A84 constant is related to the difference between this function's start and the so-called \texttt{Global Offset Table Procedure Linkage Table} (GOT PLT), the section right after the \texttt{Global Offset Table} (GOT), where the pointer to \texttt{global\_variable} is. IDA shows these offsets in their processed form to make them easier to understand, but in fact the code is:

```
.call \_\_x86\_get\_pc\_thunk\_bx
(add ebx, 1A84h)
.call \_\_x86\_get\_pc\_thunk\_bx
```

Here EBX points to the GOT PLT section and to calculate a pointer to \texttt{global\_variable} (which is stored in the GOT), 0xC must be subtracted.

To calculate pointer to the \texttt{\_\_x86\_get\_pc\_thunk\_bx} string, 0x1A30 must be subtracted.

By the way, that is the reason why the AMD64 instruction set supports RIP\textsuperscript{9}-relative addressing — to simplify PIC-code.

Let's compile the same C code using the same GCC version, but for x64.

IDA would simplify the resulting code but would suppress the RIP-relative addressing details, so we are going to use \texttt{objdump} instead of IDA to see everything:

```
0000000000000720 <f1>:
   720: 48 8b 05 b9 f8 20 00  mov  rax, QWORD PTR [rip+0x2008b9]    
   727: 53                  push rbx
   728: 89 fb               mov  ebx, edi
```

\textsuperscript{9}program counter in AMD64
0x2008b9 is the difference between the address of the instruction at 0x720 and \textit{global\_variable}, and 0x20 is the difference between the address of the instruction at 0x72A and the \textit{«returning \%d\n»} string.

As you might see, the need to recalculate addresses frequently makes execution slower (it is better in x64, though).

So it is probably better to link statically if you care about performance [see: Agner Fog, \textit{Optimizing software in C++} (2015)].

\textbf{Windows}

The PIC mechanism is not used in Windows DLLs. If the Windows loader needs to load DLL on another base address, it “patches” the DLL in memory (at the \textit{FIXUP} places) in order to correct all addresses.

This implies that several Windows processes cannot share an once loaded DLL at different addresses in different process’ memory blocks — since each instance that’s loaded in memory is \textit{fixed} to work only at these addresses..

\textbf{6.4.2 \textit{LD\_PRELOAD} hack in Linux}

This allows us to load our own dynamic libraries before others, even before system ones, like libc.so.6.

This, in turn, allows us to “substitute” our written functions before the original ones in the system libraries. For example, it is easy to intercept all calls to time(), read(), write(), etc.

Let’s see if we can fool the \textit{uptime} utility. As we know, it tells how long the computer has been working. With the help of \texttt{strace (7.2.3 on page 781)}, it is possible to see that the utility takes this information the \texttt{/proc/uptime} file:

\begin{verbatim}
$ strace uptime
...
open("/proc/uptime", O_RDONLY) = 3
lseek(3, 0, SEEK_SET) = 0
read(3, "416166.86 414629.38\n", 2047) = 20
...
\end{verbatim}

It is not a real file on disk, it is a virtual one and its contents are generated on fly in the Linux kernel. There are just two numbers:

\begin{verbatim}
$ cat /proc/uptime
416690.91 415152.03
\end{verbatim}

What we can learn from Wikipedia \textsuperscript{10}:

\begin{center}
The first number is the total number of seconds the system has been up. The second number is how much of that time the machine has spent idle, in seconds.
\end{center}

Let’s try to write our own dynamic library with the \texttt{open()}, \texttt{read()}, \texttt{close()} functions working as we need. At first, our \texttt{open()} will compare the name of the file to be opened with what we need and if it is so, it will write down the descriptor of the file opened.

Second, \texttt{read()}, if called for this file descriptor, will substitute the output, and in the rest of the cases will call the original \texttt{read()} from libc.so.6. And also \texttt{close()}, will note if the file we are currently following is to be closed.

\textsuperscript{10}https://en.wikipedia.org/wiki/Uptime
We are going to use the dlopen() and dlsym() functions to determine the original function addresses in libc.so.6.

We need them because we must pass control to the "real" functions.

On the other hand, if we intercepted strcmp() and monitored each string comparisons in the program, then we would have to implement our version of strcmp(), and not use the original function 11, that would be easier.

```c
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void *, size_t) = NULL;

bool inited = false;

_Noreturn void die(const char *fmt, ...) {
    va_list va;
    va_start(va, fmt);
    vprintf(fmt, va);
    exit(0);
};

static void find_original_functions () {
    if (inited)
        return;

    libc_handle = dlopen("libc.so.6", RTLD_LAZY);
    if (libc_handle==NULL)
        die("can't open libc.so.6\n");

    open_ptr = dlsym(libc_handle, "open");
    if (open_ptr==NULL)
        die("can't find open()\n");

    close_ptr = dlsym(libc_handle, "close");
    if (close_ptr==NULL)
        die("can't find close()\n");

    read_ptr = dlsym(libc_handle, "read");
    if (read_ptr==NULL)
        die("can't find read()\n");

    inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags) {
    find_original_functions();

    int fd=(*open_ptr)(pathname, flags);
    if (strcmp(pathname, "/proc/uptime")==0)
        opened_fd=fd; // that's our file! record its file descriptor
    else
        opened_fd=0;
    return fd;
}
```

11For example, here is how simple strcmp() interception works in this article 12 written by Yong Huang
int close(int fd)
{
    find_original_functions();

    if (fd==opened_fd)
        opened_fd=0;  // the file is not opened anymore
    return (*close_ptr)(fd);
};

ssize_t read(int fd, void *buf, size_t count)
{
    find_original_functions();

    if (opened_fd!=0 & & fd==opened_fd)
    {
        // that's our file!
        return snprintf(buf, count, "%d %d", 0x7fffffff, 0x7fffffff)+1;
    }
    // not our file, go to real read() function
    return (*read_ptr)(fd, buf, count);
};

(Source code)
Let's compile it as common dynamic library:
gcc -fpic -shared -Wall -o fool uptime.so fool uptime.c -ldl

Let's run uptime while loading our library before the others:
LD_PRELOAD=`pwd`/fool uptime.so uptime

And we see:

```
01:23:02 up 24855 days, 3:14, 3 users, load average: 0.00, 0.01, 0.05
```

If the LD_PRELOAD
environment variable always points to the filename and path of our library, it is to be loaded for all starting programs.

More examples:

- Very simple interception of the strcmp() (Yong Huang) [http://go.yurichev.com/17143](http://go.yurichev.com/17143)
- Kevin Pulo—Fun with LD_PRELOAD. A lot of examples and ideas. [yurichev.com](http://go.yurichev.com)

6.5 Windows NT

6.5.1 CRT (win32)

Does the program execution start right at the main() function? No, it does not.

If we would open any executable file in IDA or HIEW, we can see OEP pointing to some another code block. This code is doing some maintenance and preparations before passing control flow to our code. It is called startup-code or CRT code (C RunTime).

The main() function takes an array of the arguments passed on the command line, and also one with environment variables. But in fact a generic string is passed to the program, the CRT code finds the spaces in it and cuts it in parts. The CRT code also prepares the environment variables array envp.

As for GUI\(^{13}\) win32 applications, WinMain is used instead of main(), having its own arguments:

\(^{13}\)Graphical User Interface
The CRT code prepares them as well.

Also, the number returned by the main() function is the exit code.

It may be passed in CRT to the ExitProcess() function, which takes the exit code as an argument.

Usually, each compiler has its own CRT code.

Here is a typical CRT code for MSVC 2008.

```asm
__tmainCRTStartup proc near

var_24 = dword ptr -24h
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
ms_exc = CPPEH_RECORD ptr -18h

push 14h
push offset stru_4092D0
call __SEH_prolog4
mov eax, 5A4Dh
cmp ds:400000h, ax
jnz short loc_401096
mov eax, ds:40003Ch
cmp dword ptr [eax+400000h], 4550h
jnz short loc_401096
mov ecx, 10Bh
cmp [eax+400018h], cx
jnz short loc_401096
cmp dword ptr [eax+400074h], 0Eh
jbe short loc_401096
xor ecx, ecx
setnz cl
mov [ebp+var_1C], ecx
jmp short loc_40109A

loc_401096: ; CODE XREF: __tmainCRTStartup+18
            ; __tmainCRTStartup+29 ...
            and [ebp+var_1C], 0

loc_40109A: ; CODE XREF: __tmainCRTStartup+50
            push 1
            call __heap_init
            pop ecx
test eax, eax
jnz short loc_4010AE
push 1Ch
call __fast_error_exit
pop ecx

loc_4010AE: ; CODE XREF: __tmainCRTStartup+60
call __mtinit
test eax, eax
jnz short loc_4010BF
push 10h
call __fast_error_exit
pop ecx

loc_4010BF: ; CODE XREF: __tmainCRTStartup+71
call sub_401F2B
            and [ebp+ms_exc.disabled], 0

loc_7B1E: mov [ebp+ms_exc.disabled], 0
            ret

call sub_401F2B
            ret

__tmainCRTStartup endp

745
```
call _ioinit
test eax, eax
jge short loc_4010D9
push 1Bh
call __amsg_exit
pop ecx

loc_4010D9: ; CODE XREF: __tmainCRTStartup+8B
call ds:GetCommandLineA
mov dword_40B7F8, eax
call __crtGetEnvironmentStringsA
mov dword_40AC60, eax
call __setargv
test eax, eax
jge short loc_4010FF
push 8
call __amsg_exit
pop ecx

loc_4010FF: ; CODE XREF: __tmainCRTStartup+B1
call __setenvp
test eax, eax
jge short loc_401110
push 9
call __amsg_exit
pop ecx

loc_401110: ; CODE XREF: __tmainCRTStartup+C2
push 1
call __cinit
pop ecx
test eax, eax
jz short loc_401123
push eax
call __amsg_exit
pop ecx

loc_401123: ; CODE XREF: __tmainCRTStartup+D6
mov eax, envp
mov dword_40AC80, eax
push eax ; envp
push argv ; argv
push argc ; argc
call __main
add esp, 0Ch
mov [ebp+var_20], eax
cmp [ebp+var_1C], 0
jnz short $LN28
push eax ; uExitCode
call $LN32
$LN28: ; CODE XREF: __tmainCRTStartup+105
call __cexit
jmp short loc_401186

$LN27: ; DATA XREF: .rdata:stru_4092D0
mov eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 for function 401044
mov ecx, [eax]
mov ecx, [ecx]
mov [ebp+var_24], ecx
push eax
push ecx
call __XcptFilter
pop ecx
pop ecx
$LN24:
retn

746
Here we can see calls to GetCommandLineA() (line 62), then to setargv() (line 66) and setenvp() (line 74), which apparently fill the global variables argc, argv, envp.

Finally, main() is called with these arguments (line 97).

There are also calls to functions with self-describing names like heap_init() (line 35), ioinit() (line 54).

The heap is indeed initialized in the CRT. If you try to use malloc() in a program without CRT, it will exit abnormally with the following error:

**runtime error R6030**
- CRT not initialized

Global object initializations in C++ is also occur in the CRT before the execution of main(): 3.21.4 on page 564.

The value that main() returns is passed to cexit(), or in $LN32, which in turn calls doexit().

Is it possible to get rid of the CRT? Yes, if you know what you are doing.

The MSVC’s linker has the /ENTRY option for setting an entry point.

```c
#include <windows.h>

int main()
{
    MessageBox(NULL, "hello, world", "caption", MB_OK);
};
```

Let's compile it in MSVC 2008.

```shell
c /link /entry:main
```

We are getting a runnable .exe with size 2560 bytes, that has a PE header in it, instructions calling MessageBox, two strings in the data segment, the MessageBox function imported from user32.dll and nothing else.

This works, but you cannot write WinMain with its 4 arguments instead of main().

To be precise, you can, but the arguments are not prepared at the moment of execution.

By the way, it is possible to make the .exe even shorter by aligning the PE sections at less than the default 4096 bytes.

```shell
c /link /entry:main /align:16
```

Linker says:

```text
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run
```
We get an .exe that’s 720 bytes. It can be executed in Windows 7 x86, but not in x64 (an error message will be shown when you try to execute it).

With even more efforts, it is possible to make the executable even shorter, but as you can see, compatibility problems arise quickly.

### 6.5.2 Win32 PE

**PE** is an executable file format used in Windows. The difference between .exe, .dll and .sys is that .exe and .sys usually do not have exports, only imports.

A DLL, just like any other PE-file, has an entry point (OEP) (the function DllMain() is located there) but this function usually does nothing. .sys is usually a device driver. As of drivers, Windows requires the checksum to be present in the PE file and for it to be correct.

Starting at Windows Vista, a driver’s files must also be signed with a digital signature. It will fail to load otherwise.

Every PE file begins with tiny DOS program that prints a message like “This program cannot be run in DOS mode.”—if you run this program in DOS or Windows 3.1 (OS-es which are not aware of the PE format), this message will be printed.

**Terminology**

- **Module**—a separate file, .exe or .dll.
- **Process**—a program loaded into memory and currently running. Commonly consists of one .exe file and bunch of .dll files.
- **Process memory**—the memory a process works with. Each process has its own. There usually are loaded modules, memory of the stack, heap(s), etc.
- **VA**—an address which is to be used in program while runtime.
- **Base address (of module)**—the address within the process memory at which the module is to be loaded. OS loader may change it, if the base address is already occupied by another module just loaded before.
- **RVA**—the VA-address minus the base address.

Many addresses in PE-file tables use RVA-addresses.

- **IAT**—an array of addresses of imported symbols. Sometimes, the IMAGE_DIRECTORY_ENTRY_IAT data directory points at the IAT. It is worth noting that IDA (as of 6.1) may allocate a pseudo-section named .idata for IAT, even if the IAT is a part of another section!
- **INT**—an array of names of symbols to be imported.

**Base address**

The problem is that several module authors can prepare DLL files for others to use and it is not possible to reach an agreement which addresses is to be assigned to whose modules.

So that is why if two necessary DLLs for a process have the same base address, one of them will be loaded at this base address, and the other—at some other free space in process memory, and each virtual addresses in the second DLL will be corrected.

With MSVC the linker often generates the .exe files with a base address of 0x400000, and with the code section starting at 0x401000. This means that the RVA of the start of the code section is 0x1000.

DLLs are often generated by MSVC’s linker with a base address of 0x1000000.  

---

14 Dynamic-Link Library  
15 For example, Hiew (7.1 on page 780) can calculate it  
16 Virtual Address  
17 Relative Virtual Address  
18 Import Address Table  
20 Import Name Table  
22 The origin of this address choice is described here: MSDN  
23 This can be changed by the /BASE linker option
There is also another reason to load modules at various base addresses, in this case random ones. It is ASLR.

A shellcode trying to get executed on a compromised system must call system functions, hence, know their addresses.

In older OS (in Windows NT line: before Windows Vista), system DLL (like kernel32.dll, user32.dll) were always loaded at known addresses, and if we also recall that their versions rarely changed, the addresses of functions were fixed and shellcode could call them directly.

In order to avoid this, the ASLR method loads your program and all modules it needs at random base addresses, different every time.

ASLR support is denoted in a PE file by setting the flag

IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE [see Mark Russinovich, *Microsoft Windows Internals*].

**Subsystem**

There is also a subsystem field, usually it is:

- native24 (.sys-driver),
- console (console application) or
- GUI (non-console).

**OS version**

A PE file also specifies the minimal Windows version it needs in order to be loadable.

The table of version numbers stored in the PE file and corresponding Windows codenames is here.

For example, MSVC 2005 compiles .exe files for running on Windows NT4 (version 4.00), but MSVC 2008 does not (the generated files have a version of 5.00, at least Windows 2000 is needed to run them).

MSVC 2012 generates .exe files of version 6.00 by default, targeting at least Windows Vista. However, by changing the compiler’s options25, it is possible to force it to compile for Windows XP.

**Sections**

Division in sections, as it seems, is present in all executable file formats.

It is devised in order to separate code from data, and data—from constant data.

- Either the IMAGE_SCN_CNT_CODE or IMAGE_SCN_MEM_EXECUTE flags will be set on the code section—this is executable code.
- On data section—IMAGE_SCN_CNT_INITIALIZED_DATA, IMAGE_SCN_MEM_READ and IMAGE_SCN_MEM_WRITE flags.
- On an empty section with uninitialized data—IMAGE_SCN_CNT_UNINITIALIZED_DATA, IMAGE_SCN_MEM_READ and IMAGE_SCN_MEM_WRITE.
- On a constant data section (one that’s protected from writing), the flags IMAGE_SCN_CNT_INITIALIZED_DATA and IMAGE_SCN_MEM_READ can be set, but not IMAGE_SCN_MEM_WRITE. A process going to crash if it tries to write to this section.

Each section in PE-file may have a name, however, it is not very important. Often (but not always) the code section is named .text, the data section—.data, the constant data section — .rdata (readable data) (perhaps, .rdata means read-only-data). Other popular section names are:

- .idata—imports section. IDA may create a pseudo-section named like this: 6.5.2 on the preceding page.
- .edata—exports section (rare)
- .pdata—section holding all information about exceptions in Windows NT for MIPS, IA64 and x64: 6.5.3 on page 773
- .reloc—relocs section

---

24 Meaning, the module use Native API instead of Win32
25 MSDN
• `.bss`—uninitialized data (BSS)
• `.tls`—thread local storage (TLS)
• `.rsrc`—resources
• `.CRT`—may present in binary files compiled by ancient MSVC versions

PE file packers/encryptors often garble section names or replace the names with their own. MSVC allows you to declare data in arbitrarily named section \(^26\).

Some compilers and linkers can add a section with debugging symbols and other debugging information (MinGW for instance). However it is not so in latest versions of MSVC (separate PDB files are used there for this purpose).

That is how a PE section is described in the file:

```c
typedef struct _IMAGE_SECTION_HEADER {
    BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD PhysicalAddress;
        DWORD VirtualSize;
    } Misc;
    DWORD VirtualAddress;
    DWORD SizeOfRawData;
    DWORD PointerToRawData;
    DWORD PointerToRelocations;
    DWORD PointerToLineNumbers;
    WORD NumberOfRelocations;
    WORD NumberOfLineNumbers;
    DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

\(^27\)

A word about terminology: `PointerToRawData` is called “Offset” in Hiew and `VirtualAddress` is called “RVA” there.

### Data section

Data section in file can be smaller than in memory. For example, some variables can be initialized, some are not. Compiler and linker will collect them all into one section, but the first part of it is initialized and allocated in file, while another is absent in file (of course, to make it smaller). `VirtualSize` will be equal to the size of section in memory, and `SizeOfRawData` — to size of section in file.

IDA can show the border between initialized and not initialized parts like that:

```plaintext
... 
.data:10017FFA    db  0  
data:10017FFB    db  0  
data:10017FFC    db  0  
data:10017FFD    db  0  
data:10017FFE    db  0  
data:10017FFF    db  0  
data:10018000    db  ?;  
data:10018001    db  ?;  
data:10018002    db  ?;  
data:10018003    db  ?;  
data:10018004    db  ?;  
data:10018005    db  ?;  
... 
```

\(^26\)MSDN
\(^27\)MSDN
Strings are usually located here (because they have `const char*` type), other variables marked as `const`, imported function names.

See also: 3.3 on page 468.

**Relocations (relocs)**

AKA FIXUP-s (at least in Hiew).

They are also present in almost all executable file formats. Exceptions are shared dynamic libraries compiled with PIC, or any other PIC-code.

What are they for?

Obviously, modules can be loaded on various base addresses, but how to deal with global variables, for example? They must be accessed by address. One solution is position-independent code (6.4.1 on page 739). But it is not always convenient.

That is why a relocations table is present. There the addresses of points that must be corrected are enumerated, in case of loading at a different base address.

For example, there is a global variable at address `0x410000` and this is how it is accessed:

```
A1 00 00 41 00  mov  eax, [000410000]
```

The base address of the module is `0x400000`, the RVA of the global variable is `0x10000`.

If the module is loaded at base address `0x500000`, the real address of the global variable must be `0x510000`.

As we can see, the address of variable is encoded in the instruction `MOV`, after the byte `0xA1`.

That is why the address of the 4 bytes after `0xA1`, is written in the relocations table.

If the module is loaded at a different base address, the OS loader enumerates all addresses in the table, finds each 32-bit word the address points to, subtracts the original base address from it (we get the RVA here), and adds the new base address to it.

If a module is loaded at its original base address, nothing happens.

All global variables can be treated like that.

Relocs may have various types, however, in Windows for x86 processors, the type is usually `IMAGE_REL_BASED_HIGHLOW`.

By the way, relocations are darkened in Hiew, for example: fig.1.22. (You have to circumvent these bytes during patching.)

OllyDbg underlines the places in memory to which relocations are to be applied, for example: fig.1.53.

**Exports and imports**

As we all know, any executable program must use the OS’s services and other DLL-libraries somehow.

It can be said that functions from one module (usually DLL) must be connected somehow to the points of their calls in other modules (.exe-file or another DLL).

For this, each DLL has an “exports” table, which consists of functions plus their addresses in a module.

And every .exe file or DLL has “imports”, a table of functions it needs for execution including list of DLL filenames.

After loading the main .exe-file, the OS loader processes imports table: it loads the additional DLL-files, finds function names among the DLL exports and writes their addresses down in the IAT of the main .exe-module.

As we can see, during loading the loader must compare a lot of function names, but string comparison is not a very fast procedure, so there is a support for “ordinals” or “hints”, which are function numbers stored in the table, instead of their names.

That is how they can be located faster when loading a DLL. Ordinals are always present in the “export” table.

---

28 Even in .exe files for MS-DOS
For example, a program using the MFC library usually loads mfc*.dll by ordinals, and in such programs there are no MFC function names in INT.

When loading such programs in IDA, it will ask for a path to the mfc*.dll files in order to determine the function names.

If you don’t tell IDA the path to these DLLs, there will be mfc80_123 instead of function names.

**Imports section**

Often a separate section is allocated for the imports table and everything related to it (with name like .idata), however, this is not a strict rule.

Imports are also a confusing subject because of the terminological mess. Let’s try to collect all information in one place.

---

**Figure 6.1:** A scheme that unites all PE-file structures related to imports

---

29 Microsoft Foundation Classes
The main structure is the array \texttt{IMAGE\_IMPORT\_DESCRIPTOR}. Each element for each DLL being imported. Each element holds the RVA address of the text string (DLL name) \textit{(Name)}.

\textit{OriginalFirstThunk} is the RVA address of the INT table. This is an array of RVA addresses, each of which points to a text string with a function name. Each string is prefixed by a 16-bit integer (“hint”)—“ordinal” of function.

While loading, if it is possible to find a function by ordinal, then the strings comparison will not occur. The array is terminated by zero.

There is also a pointer to the IAT table named \textit{FirstThunk}, it is just the RVA address of the place where the loader writes the addresses of the resolved functions.

The points where the loader writes addresses are marked by IDA like this: \_\_imp\_CreateFileA, etc.

There are at least two ways to use the addresses written by the loader.

- The code will have instructions like \texttt{call \_\_imp\_CreateFileA}, and since the field with the address of the imported function is a global variable in some sense, the address of the call instruction (plus 1 or 2) is to be added to the relocs table, for the case when the module is loaded at a different base address.

  But, obviously, this may enlarge relocs table significantly.

  Because there are might be a lot of calls to imported functions in the module.

  Furthermore, large relocs table slows down the process of loading modules.

- For each imported function, there is only one jump allocated, using the JMP instruction plus a reloc to it. Such points are also called “thunks”.

  All calls to the imported functions are just CALL instructions to the corresponding “thunk”. In this case, additional relocs are not necessary because these CALL-s have relative addresses and do not need to be corrected.

These two methods can be combined.

Possible, the linker creates individual “thunk”s if there are too many calls to the function, but not done by default.

By the way, the array of function addresses to which FirstThunk is pointing is not necessary to be located in the IAT section. For example, the author of these lines once wrote the PE\_add\_import\textsuperscript{30} utility for adding imports to an existing .exe-file.

Some time earlier, in the previous versions of the utility, at the place of the function you want to substitute with a call to another DLL, my utility wrote the following code:

\begin{verbatim}
MOV EAX, [yourdll.dll!function]
JMP EAX
\end{verbatim}

FirstThunk points to the first instruction. In other words, when loading yourdll.dll, the loader writes the address of the function right in the code.

It also worth noting that a code section is usually write-protected, so my utility adds the \texttt{IMAGE\_SCN\_MEM\_WRITE} flag for code section. Otherwise, the program to crash while loading with error code 5 (access denied).

One might ask: what if I supply a program with a set of DLL files which is not supposed to change (including addresses of all DLL functions), is it possible to speed up the loading process?

Yes, it is possible to write the addresses of the functions to be imported into the FirstThunk arrays in advance. The \texttt{Timestamp} field is present in the \texttt{IMAGE\_IMPORT\_DESCRIPTOR} structure.

If a value is present there, then the loader compares this value with the date-time of the DLL file.

If the values are equal, then the loader does not do anything, and the loading of the process can be faster. This is called “old-style binding” \textsuperscript{31}.

The BIND.EXE utility in Windows SDK is for this. For speeding up the loading of your program, Matt Pietrek in Matt Pietrek, \textit{An In-Depth Look into the Win32 Portable Executable File Format}, (2002)\textsuperscript{32}, suggests to

\textsuperscript{30}yurichev.com
\textsuperscript{31}MSDN. There is also the “new-style binding”.
\textsuperscript{32}Also available as http://go.yurichev.com/17318
do the binding shortly after your program installation on the computer of the end user.

PE-files packers/encryptors may also compress/encrypt imports table.

In this case, the Windows loader, of course, will not load all necessary DLLs.

Therefore, the packer/encryptor does this on its own, with the help of \textit{LoadLibrary()} and the \textit{GetProcAddress()} functions.

That is why these two functions are often present in \textit{IAT} in packed files.

In the standard DLLs from the Windows installation, \textit{IAT} often is located right at the beginning of the PE file. Supposedly, it is made so for optimization.

While loading, the .exe file is not loaded into memory as a whole (recall huge install programs which are started suspiciously fast), it is “mapped”, and loaded into memory in parts as they are accessed.

Probably, Microsoft developers decided it will be faster.

\textbf{Resources}

Resources in a PE file are just a set of icons, pictures, text strings, dialog descriptions.

Perhaps they were separated from the main code, so all these things could be multilingual, and it would be simpler to pick text or picture for the language that is currently set in the \textit{OS}.

As a side effect, they can be edited easily and saved back to the executable file, even if one does not have special knowledge, by using the ResHack editor, for example (6.5.2).

\textbf{.NET}

.NET programs are not compiled into machine code but into a special bytecode. Strictly speaking, there is bytecode instead of the usual x86 code in the .exe file, however, the entry point (OEP) points to this tiny fragment of x86 code:

\begin{verbatim}
jmp mscoree.dll!_CorExeMain
\end{verbatim}

The .NET loader is located in mscoree.dll, which processes the PE file.

It was so in all pre-Windows XP OSes. Starting from XP, the OS loader is able to detect the .NET file and run it without executing that JMP instruction $^{33}$.

\textbf{TLS}

This section holds initialized data for the TLS (6.2 on page 734) (if needed). When a new thread start, its TLS data is initialized using the data from this section.

Aside from that, the PE file specification also provides initialization of the TLS section, the so-called TLS callbacks.

If they are present, they are to be called before the control is passed to the main entry point (OEP).

This is used widely in the PE file packers/encryptors.

\textbf{Tools}

- objdump (present in cygwin) for dumping all PE-file structures.
- Hiew (7.1 on page 780) as editor.
- pefile—Python-library for PE-file processing $^{34}$.
- ResHack AKA Resource Hacker—resources editor $^{35}$.
- PE_add_import$^{36}$— simple tool for adding symbol(s) to PE executable import table.

\footnotesize
$^{33}$MSDN
$^{34}$http://go.yurichev.com/17052
$^{35}$http://go.yurichev.com/17052
$^{36}$http://go.yurichev.com/17049
• PE_patcher\textsuperscript{37}—simple tool for patching PE executables.
• PE_search_str_refs\textsuperscript{38}—simple tool for searching for a function in PE executables which use some text string.

Further reading
• Daniel Pistelli—The .NET File Format \textsuperscript{39}

6.5.3 Windows SEH

Let’s forget about MSVC

In Windows, the SEH is intended for exceptions handling, nevertheless, it is language-agnostic, not related to C++ or OOP in any way.

Here we are going to take a look at SEH in its isolated (from C++ and MSVC extensions) form.

Each running process has a chain of SEH handlers, each TIB has the address of the most recently defined handler.

When an exception occurs (division by zero, incorrect address access, user exception triggered by calling the \texttt{RaiseException()} function), the OS finds the last handler in the TIB and calls it, passing exception kind and all information about the CPU state (register values, etc.) at the moment of the exception.

The exception handler considering the exception, does it see something familiar? If so, it handles the exception.

If not, it signals to the OS that it cannot handle it and the OS calls the next handler in the chain, until a handler which is able to handle the exception is be found.

At the very end of the chain there a standard handler that shows the well-known dialog box, informing the user about a process crash, some technical information about the CPU state at the time of the crash, and offering to collect all information and send it to developers in Microsoft.

\begin{center}
\begin{tabular}{|p{\textwidth}|}
\hline
\texttt{crash.exe}\textsuperscript{\textsuperscript{37}}
\hline
\texttt{crash.exe} has encountered a problem and needs to close. We are sorry for the inconvenience.
\hline
If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.
\ \ \ Send Error Report \ \ Don't Send
\hline
\end{tabular}
\end{center}

Figure 6.2: Windows XP

\textsuperscript{37}yurichev.com
\textsuperscript{38}yurichev.com
\textsuperscript{39}http://go.yurichev.com/17056
Figure 6.3: Windows XP

Figure 6.4: Windows 7
Earlier, this handler was called Dr. Watson.

By the way, some developers make their own handler that sends information about the program crash to themselves. It is registered with the help of `SetUnhandledExceptionFilter()` and to be called if the OS does not have any other way to handle the exception. An example is Oracle RDBMS—it saves huge dumps reporting all possible information about the CPU and memory state.

Let's write our own primitive exception handler. This example is based on the example from [Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling, (1997)*]({http://go.yurichev.com/17293}). It must be compiled with the SAFESEH option: `cl seh1.cpp /link /safeseh:no`. More about SAFESEH here: [MSDN](http://go.yurichev.com/17293).

```c
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void *EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void *DispatcherContext )
{
    unsigned i;

    printf ("%s\n", __FUNCTION__);
    printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->ExceptionCode);
    printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->ExceptionFlags);
    printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ExceptionRecord->ExceptionAddress);

    if (ExceptionRecord->ExceptionCode==0xE1223344)
    {
        printf ("That's for us\n");
        // yes, we "handled" the exception
        return ExceptionContinueExecution;
    }
    else if (ExceptionRecord->ExceptionCode==EXCEPTION_ACCESS_VIOLATION)
    {
        printf ("ContextRecord->Eax=0x%08X\n", ContextRecord->Eax);
        // will it be possible to 'fix' it?
        printf ("Trying to fix wrong pointer address\n");
        ContextRecord->Eax=(DWORD)&new_value;
        // yes, we "handled" the exception
        return ExceptionContinueExecution;
    }
    else
    {
        printf ("We do not handle this\n");
        // someone else's problem
        return ExceptionContinueSearch;
    }
}
```

---

40 Also available as [http://go.yurichev.com/17293](http://go.yurichev.com/17293)
int main()
{
    DWORD handler = (DWORD)except_handler; // take a pointer to our handler

    // install exception handler
    __asm
    {
        push handler // make EXCEPTION_REGISTRATION record:
        push FS:[0] // address of handler function
        mov FS:[0],ESP // address of previous handler
        // add new EXCEPTION_REGISTRATION
    }

    RaiseException (0xE123344, 0, 0, NULL);

    // now do something very bad
    int *ptr=NULL;
    int val=0;
    val=*ptr;
    printf ("val=%d\n", val);

    // deinstall exception handler
    __asm
    {
        mov eax,[ESP] // remove our EXCEPTION_REGISTRATION record
        mov FS:[0], EAX // get pointer to previous record
        add esp, 8 // install previous record
        // clean our EXCEPTION_REGISTRATION off stack
    }

    return 0;
}

The FS: segment register is pointing to the TIB in win32.

The very first element in the TIB is a pointer to the last handler in the chain. We save it in the stack and store the address of our handler there. The structure is named _EXCEPTION_REGISTRATION, it is a simple singly-linked list and its elements are stored right in the stack.

```
LISTING 6.22: MSVC/VC/crt/src/exsup.inc

_EXCEPTION_REGISTRATION struc
    prev dd ?,
    handler dd ?,
_EXCEPTION_REGISTRATION ends
```

So each “handler” field points to a handler and an each “prev” field points to the previous record in the chain of exception handlers. The last record has 0xFFFFFFFF (-1) in the “prev” field.
After our handler is installed, we call RaiseException() \(^{41}\). This is an user exception. The handler checks the code. If the code is \(0x12345678\), it returning ExceptionContinueExecution, which means that handler corrected the CPU state (it is usually a correction of the EIP/ESP registers) and the OS can resume the execution of the thread. If you alter slightly the code so the handler returns ExceptionContinueSearch, then the OS will call the other handlers, and it’s unlikely that one who can handle it will be found, since no one will have any information about it (rather about its code). You will see the standard Windows dialog about a process crash.

What is the difference between a system exceptions and a user one? Here are the system ones:

<table>
<thead>
<tr>
<th>as defined in WinBase.h</th>
<th>as defined in ntstatus.h</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCEPTION_ACCESS_VIOLATION</td>
<td>STATUS_ACCESS_VIOLATION</td>
<td>0xC0000005</td>
</tr>
<tr>
<td>EXCEPTION_DATATYPE_MISALIGNMENT</td>
<td>STATUS_DATATYPE_MISALIGNMENT</td>
<td>0x80000002</td>
</tr>
<tr>
<td>EXCEPTION_BREAKPOINT</td>
<td>STATUS_BREAKPOINT</td>
<td>0x80000003</td>
</tr>
<tr>
<td>EXCEPTION_SINGLE_STEP</td>
<td>STATUS_SINGLE_STEP</td>
<td>0x80000004</td>
</tr>
<tr>
<td>EXCEPTION_ARRAY_BOUNDS_EXCEEDED</td>
<td>STATUS_ARRAY_BOUNDS_EXCEEDED</td>
<td>0xC0000008C</td>
</tr>
<tr>
<td>EXCEPTION_FLT_DENORMAL_OPERAND</td>
<td>STATUS_FLOAT_DENORMAL_OPERAND</td>
<td>0xC0000008D</td>
</tr>
<tr>
<td>EXCEPTION_FLT_DIVIDE_BY_ZERO</td>
<td>STATUS_FLOAT_DIVIDE_BY_ZERO</td>
<td>0xC0000008E</td>
</tr>
<tr>
<td>EXCEPTION_FLT_INEXACT_RESULT</td>
<td>STATUS_FLOAT_INEXACT_RESULT</td>
<td>0xC0000008F</td>
</tr>
<tr>
<td>EXCEPTION_FLT_INVALID_OPERATION</td>
<td>STATUS_FLOAT_INVALID_OPERATION</td>
<td>0xC00000090</td>
</tr>
<tr>
<td>EXCEPTION_FLT_OVERFLOW</td>
<td>STATUS_FLOAT_OVERFLOW</td>
<td>0xC00000091</td>
</tr>
<tr>
<td>EXCEPTION_FLT_STACK_CHECK</td>
<td>STATUS_FLOAT_STACK_CHECK</td>
<td>0xC00000092</td>
</tr>
<tr>
<td>EXCEPTION_FLT_UNDERFLOW</td>
<td>STATUS_FLOAT_UNDERFLOW</td>
<td>0xC00000093</td>
</tr>
<tr>
<td>EXCEPTION_INT_DIVIDE_BY_ZERO</td>
<td>STATUS_INTEGER_DIVIDE_BY_ZERO</td>
<td>0xC00000094</td>
</tr>
<tr>
<td>EXCEPTION_INT_OVERFLOW</td>
<td>STATUS_INTEGER_OVERFLOW</td>
<td>0xC00000095</td>
</tr>
<tr>
<td>EXCEPTION_PRIV_INSTRUCTION</td>
<td>STATUS_PRIVILEGED_INSTRUCTION</td>
<td>0xC00000096</td>
</tr>
<tr>
<td>EXCEPTION_IN_PAGE_ERROR</td>
<td>STATUS_IN_PAGE_ERROR</td>
<td>0xC00000097</td>
</tr>
<tr>
<td>EXCEPTION_ILLEGAL_INSTRUCTION</td>
<td>STATUS_ILLEGAL_INSTRUCTION</td>
<td>0xC000001D</td>
</tr>
<tr>
<td>EXCEPTION_NONCONTINUABLE_EXCEPTION</td>
<td>STATUS_NONCONTINUABLE_EXCEPTION</td>
<td>0xC0000025</td>
</tr>
<tr>
<td>EXCEPTION_STACK_OVERFLOW</td>
<td>STATUS_STACK_OVERFLOW</td>
<td>0xC00000FD</td>
</tr>
<tr>
<td>EXCEPTION_INVALID_DISPOSITION</td>
<td>STATUS_INVALID_DISPOSITION</td>
<td>0xC0000026</td>
</tr>
<tr>
<td>EXCEPTION_GUARD_PAGE</td>
<td>STATUS_GUARD_PAGE_VIOLATION</td>
<td>0x80000001</td>
</tr>
<tr>
<td>EXCEPTION_INVALID_HANDLE</td>
<td>STATUS_INVALID_HANDLE</td>
<td>0xC0000008</td>
</tr>
<tr>
<td>EXCEPTION_POSSIBLE_DEADLOCK</td>
<td>STATUS_POSSIBLE_DEADLOCK</td>
<td>0xC0000194</td>
</tr>
<tr>
<td>CONTROL_C_EXIT</td>
<td>STATUS_CONTROL_C_EXIT</td>
<td>0xC000013A</td>
</tr>
</tbody>
</table>

That is how the code is defined:

<table>
<thead>
<tr>
<th>S</th>
<th>U</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>31 29 28 27</td>
<td>16 15</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Facility code</th>
<th>Error code</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>U</td>
</tr>
</tbody>
</table>

S is a basic status code: 11—error; 10—warning; 01—informational; 00—success. U—whether the code is user code.

\(^{41}\)MSDN
That is why we chose 0xE1223344—E₁₆ (1110₂) 0xE (1110b) means that it is 1) user exception; 2) error. But to be honest, this example works fine without these high bits.

Then we try to read a value from memory at address 0.

Of course, there is nothing at this address in win32, so an exception is raised.

The very first handler is to be called—yours, and it will know about it first, by checking the code if it’s equal to the EXCEPTION_ACCESS_VIOLATION constant.

The code that’s reading from memory at address 0 is looks like this:

```
Listing 6.23: MSVC 2010
...
xor eax, eax
mov eax, DWORD PTR [eax]; exception will occur here
push eax
push OFFSET msg
call _printf
add esp, 8
...
```

Will it be possible to fix this error “on the fly” and to continue with program execution?

Yes, our exception handler can fix the EAX value and let the OS execute this instruction once again. So that is what we do. printf() prints 1234, because after the execution of our handler EAX is not 0, but contains the address of the global variable new_value. The execution will resume.

That is what is going on: the memory manager in the CPU signals about an error, the CPU suspends the thread, finds the exception handler in the Windows kernel, which, in turn, starts to call all handlers in the SEH chain, one by one.

We use MSVC 2010 here, but of course, there is no any guarantee that EAX will be used for this pointer.

This address replacement trick is showy, and we considering it here as an illustration of SEH’s internals. Nevertheless, it’s hard to recall any case where it is used for “on-the-fly” error fixing.

Why SEH-related records are stored right in the stack instead of some other place?
Supposedly because the OS is not needing to care about freeing this information, these records are simply disposed when the function finishes its execution. This is somewhat like alloca(): (1.9.2 on page 34).

**Now let’s get back to MSVC**

Supposedly, Microsoft programmers needed exceptions in C, but not in C++ (for use in Windows NT kernel, which is written in C), so they added a non-standard C extension to MSVC. It is not related to C++ PL exceptions.

```
__try
{
    ...
}
_except(filter code)
{
    handler code
}
```

“Finally” block may be instead of handler code:

```
__try
{
    ...
}
__finally
{
    ...
}
```

---

42 MSDN
The filter code is an expression, telling whether this handler code corresponds to the exception raised. If your code is too big and cannot fit into one expression, a separate filter function can be defined.

There are a lot of such constructs in the Windows kernel. Here are a couple of examples from there (WRK):

### Listing 6.24: WRK-v1.2/base/ntos/ob/obwait.c

```c
try {
    KeReleaseMutant( (PKMUTANT)SignalObject, 
    MUTANT_INCREMENT, 
    FALSE, 
    TRUE );
} except((GetExceptionCode () == STATUS_ABANDONED || 
    GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?) {
    EXCEPTION_EXECUTE_HANDLER : 
    EXCEPTION_CONTINUE_SEARCH) {
        Status = GetExceptionCode();
        goto WaitExit;
}
```

### Listing 6.25: WRK-v1.2/base/ntos/cache/cachesub.c

```c
try {
    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset), 
        UserBuffer, 
        MorePages ? 
            (PAGE_SIZE - PageOffset) : 
            (ReceivedLength - PageOffset) );
} except( CcCopyReadExceptionFilter( GetExceptionInformation(), 
        &Status ) ) {
```

Here is also a filter code example:

### Listing 6.26: WRK-v1.2/base/ntos/cache/copysup.c

```c
LONG CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer, 
    IN PNTSTATUS ExceptionCode 
)

/**+
Routine Description:

This routine serves as an exception filter and has the special job of
extracting the "real" I/O error when Mm raises STATUS_IN_PAGE_ERROR
beneath us.

Arguments:

    ExceptionPointer - A pointer to the exception record that contains
        the real Io Status.

    ExceptionCode - A pointer to an NTSTATUS that is to receive the real
        status.

Return Value:

    EXCEPTION_EXECUTE_HANDLER

--*/
```
*ExceptionCode = ExceptionPointer-&gt;ExceptionRecord-&gt;ExceptionCode;

if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
    (ExceptionPointer-&gt;ExceptionRecord-&gt;NumberParameters >= 3) ) {
    *ExceptionCode = (NTSTATUS) ExceptionPointer-&gt;ExceptionRecord-&gt;ExceptionInformation[2];
}

ASSERT( !NT_SUCCESS(*ExceptionCode) );

return EXCEPTION_EXECUTE_HANDLER;
}

Internally, SEH is an extension of the OS-supported exceptions. But the handler function is _except_handler3 (for SEH3) or _except_handler4 (for SEH4).
The code of this handler is MSVC-related, it is located in its libraries, or in msvcr*.dll. It is very important to know that SEH is a MSVC thing.
Other win32-compilers may offer something completely different.

**SEH3**

SEH3 has _except_handler3 as a handler function, and extends the _EXCEPTION_REGISTRATION table, adding a pointer to the scope table and previous try level variable. SEH4 extends the scope table by 4 values for buffer overflow protection.

The scope table is a table that consists of pointers to the filter and handler code blocks, for each nested level of try/except.
Again, it is very important to understand that the OS takes care only of the \textit{prev/handle} fields, and nothing more. It is the job of the \texttt{_except_handler3} function to read the other fields and \texttt{scope table}, and decide which handler to execute and when.

The source code of the \texttt{_except_handler3} function is closed. However, Sanos OS, which has a win32 compatibility layer, has the same functions reimplemented, which are somewhat equivalent to those in Windows\textsuperscript{43}. Another reimplementation is present in Wine\textsuperscript{44} and ReactOS\textsuperscript{45}.

If the \texttt{filter} pointer is NULL, the \texttt{handler} pointer is the pointer to the \texttt{finally} code block.

During execution, the \texttt{previous try level} value in the stack changes, so \texttt{_except_handler3} can get information about the current level of nestedness, in order to know which \texttt{scope table} entry to use.

**SEH3: one try/except block example**

```c
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
    try
    {
        printf("hello #1!\n");
        *p = 13; // causes an access violation exception;
        printf("hello #2!\n");
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
    {
        printf("access violation, can't recover\n");
    }
}
```

Listing 6.27: MSVC 2003

\texttt{$SG74605 \text{DB 'hello}\ #1\!',\ 0AH,\ 00H}$
\texttt{$SG74606 \text{DB 'hello}\ #2\!',\ 0AH,\ 00H}$
\texttt{$SG74608 \text{DB 'access violation, can't recover'},\ 0AH,\ 00H}$

\texttt{DATA \text{ENDS}}

\begin{verbatim}
; scope table:
CONST SEGMENT
$T74622 DD 0ffffffffH ; previous try level
DD FLAT:$L74617 ; filter
DD FLAT:$L74618 ; handler

CONST ENDS
TEXT SEGMENT
$L74621 = -32 ; size = 4
_p$ = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24
_main PROC NEAR
push ebp
mov ebp, esp
push -1
push OFFSET FLAT:$T74622
push OFFSET FLAT:__except_handler3
mov eax, DWORD PTR fs:__except_list
push eax

\end{verbatim}

\textsuperscript{43}http://go.yurichev.com/17058
\textsuperscript{44}GitHub
\textsuperscript{45}http://go.yurichev.com/17060
Here we see how the SEH frame is constructed in the stack. The scope table is located in the CONST segment—indeed, these fields are not to be changed. An interesting thing is how the previous try level variable has changed. The initial value is 0xFFFFFFFF (−1). The moment when the body of the try statement is opened is marked with an instruction that writes 0 to the variable. The moment when the body of the try statement is closed, −1 is written back to it. We also see the addresses of filter and handler code.

Thus we can easily see the structure of the try/except constructs in the function.

Since the SEH setup code in the function prologue may be shared between many functions, sometimes the compiler inserts a call to the SEH_prolog() function in the prologue, which does just that.

The SEH cleanup code is in the SEH_epilog() function.
Let's try to run this example in tracer:

tracer.exe -l:2.exe --dump-seh

Listing 6.28: tracer.exe output

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0000e3c8
ESP=0x00000000 EBP=0x0018feac EDI=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF

* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!_except_handler3)
  scopetable entry[0]. previous try level=-1, filter=0x401070 (2.exe!main+0x60) handler=0x401088
  handler=0x40104f (2.exe!mainCRTStartup+0x18d)

* SEH frame at 0x18ff78 prev=0x18ff4c handler=0x401204 (2.exe!_except_handler3)
  SEH3 frame. previous try level=0
  scopetable entry[0]. previous try level=-1, filter=0x401531 (2.exe!mainCRTStartup+0x18d)
  handler=0x401545 (2.exe!mainCRTStartup+0x1a1)

* SEH frame at 0x18ff4c prev=0x18ffe4 handler=0x771f71f5 (ntdll.dll!__exception_handler4)
  SEH4 frame. previous try level=0
  SEH4 header: GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
  EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
  scopetable entry[0]. previous try level=-2, filter=0x771f4d0 (ntdll.dll!_safe_se_handler_table+0x20)
  handler=0x771f0eb (ntdll.dll! TppTerminateProcess@4+0x43)

* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16)
```

We see that the SEH chain consists of 4 handlers.

The first two are located in our example. Two? But we made only one? Yes, another one has been set up in the CRT function _mainCRTStartup(), and as it seems that it handles at least FPU exceptions. Its source code can be found in the MSVC installation: crtsrc/winxfltr.c.

The third is the SEH4 one in ntdll.dll, and the fourth handler is not MSVC-related and is located in ntdll.dll, and has a self-describing function name.

As you can see, there are 3 types of handlers in one chain:

one is not related to MSVC at all (the last one) and two MSVC-related: SEH3 and SEH4.

**SEH3: two try/except blocks example**

```c
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions (unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    }
}

int main()
{
    int* p = NULL;
    __try
    {
        __try
        {
```
printf ("hello!\n");
RaiseException (0x112233, 0, 0, NULL);
printf ("0x112233 raised. now let's crash\n");
*p = 13;  // causes an access violation exception;
}
_except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
{
    printf("access violation, can't recover\n");
}
_except(filter_user_exceptions(GetExceptionCode(), GetExceptionInformation()))
{
    // the filter_user_exceptions() function answering to the question
    // "is this exception belongs to this block?"
    // if yes, do the follow:
    printf("user exception caught\n");
}

Now there are two try blocks. So the scope table now has two entries, one for each block. Previous try level changes as execution flow enters or exits the try block.

Listing 6.29: MSVC 2003

 FILTER_USER_EXCEPTIONS PROC NEAR
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push OFFSET FLAT:$SG74606  ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867; 00112233H
    jne    SHORT $L74607
    push OFFSET FLAT:$SG74608  ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $L74605
$L74607:
    push OFFSET FLAT:$SG74610  ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$L74605:
    pop    ebp
    ret    0
_FILTER_USER_EXCEPTIONS ENDP

; scope table:
CONST SEGMENT
T74644  DD  0ffffffff ; previous try level for outer block
T74634  DD  FLAT:$L74634  ; outer block filter
T74635  DD  FLAT:$L74635  ; outer block handler
00H     DD  FLAT:$L74638  ; previous try level for inner block
T74638  DD  FLAT:$L74638  ; inner block filter
T74639  DD  FLAT:$L74639  ; inner block handler
CONST ENDS

766
If we set a breakpoint on the \texttt{printf()} function, which is called from the handler, we can also see how yet another SEH handler is added.

Perhaps it's another machinery inside the SEH handling process. Here we also see our \textit{scope table} consisting of 2 entries.

tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh

\begin{verbatim}
Listing 6.30: tracer.exe output
(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x008e3c8 
ESI=0x00000000 EDI=0x00000000 EBP=0x18f840 ESP=0x18f838 
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.dll!ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.exe!main+0xb0) handler=0x40113b
\texttt{/uni2926/}
/rcurvearrowse
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.exe!main+0x78) handler=0x401100
\texttt{/uni2926/}
/rcurvearrowse
* SEH frame at 0x18ff78 prev=0x18fffc4 handler=0x4012e0 (3.exe!_except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.exe!mainCRTStartup+0x18d) \texttt{/uni2926/}
\texttt{/rcurvearrowse} handler=0x401621 (3.exe!mainCRTStartup+0x1a1)
* SEH frame at 0x18fffc4 prev=0x18ffe4 handler=0x771f7f15 (ntdll.dll!__safe_se_handler_table+0x20)
SEH4 frame. previous trylevel=0
SEH4 header: GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0 \texttt{/uni2926/}
\texttt{/rcurvearrowse} EHCookieOffset=0xffffffffc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll!\texttt{/uni2926/}
\texttt{/rcurvearrowse} _safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll! _TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16)
\end{verbatim}

\end{verbatim}
During a buffer overflow (1.26.2 on page 271) attack, the address of the scope table can be rewritten, so starting from MSVC 2005, SEH3 was upgraded to SEH4 in order to have buffer overflow protection. The pointer to the scope table is now xorred with a security cookie. The scope table was extended to have a header consisting of two pointers to security cookies.

Each element has an offset inside the stack of another value: the address of the stack frame (EBP) xorred with the security_cookie, placed in the stack.

This value will be read during exception handling and checked for correctness. The security_cookie in the stack is random each time, so hopefully a remote attacker can’t predict it.

The initial previous try level is −2 in SEH4 instead of −1.

Here are both examples compiled in MSVC 2012 with SEH4:

Listing 6.31: MSVC 2012: one try block example

```plaintext
SG85485 DB 'hello #1!', 0AH, 00H
SG85486 DB 'hello #2!', 0AH, 00H
```
$SG85488 DB 'access violation, can't recover', 0AH, 00H

; scope table:
 xdata$x SEGMENT
 __sehtables$main DD 0fffffff7EH ; GS Cookie Offset
   DD 00H ; GS Cookie XOR Offset
   DD 0fffffffccH ; EH Cookie Offset
   DD 00H ; EH Cookie XOR Offset
   DD FLAT:$LN12@main ; previous try level
   DD FLAT:$LN8@main ; handler
xdata$x ENDS

$T2 = -36 ; size = 4
_p$ = -32 ; size = 4
tv68 = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24

_main PROC
  push ebp
  mov ebp, esp
  push -2
  push OFFSET __sehtables$main
  push OFFSET _except_handler4
  mov eax, DWORD PTR fs:0
  push eax
  add esp, -20
  push ebx
  push esi
  push edi
  mov eax, DWORD PTR __security_cookie
  xor DWORD PTR _$SEHRec$[ebp+16], eax ; xored pointer to scope table
  xor eax, ebp
  push eax
  lea eax, DWORD PTR _$SEHRec$[ebp+8] ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
  mov DWORD PTR fs:0, eax
  mov DWORD PTR _$SEHRec$[ebp], esp
  mov DWORD PTR _p$_[ebp], 0
  mov DWORD PTR _$SEHRec$[ebp+20], 0 ; previous try level
  push OFFSET $SG85485 ; 'hello #1!'
  call _printf
  add esp, 4
  mov eax, DWORD PTR _p$_[ebp]
  mov DWORD PTR [eax], 13
  push OFFSET $SG85486 ; 'hello #2!'
  call _printf
  add esp, 4
  mov DWORD PTR _$SEHRec$[ebp+20], -2 ; previous try level
  jmp SHORT $LN6@main;

; handler:
$LN7@main:
$LN12@main:
  mov ecx, DWORD PTR _$SEHRec$[ebp+4]
  mov edx, DWORD PTR [ecx]
  mov eax, DWORD PTR [edx]
  mov DWORD PTR $T2[ebp], eax
  cmp DWORD PTR $T2[ebp], -1073741819 ; c0000005H
  jne SHORT $LN4@main
  mov DWORD PTR tv68[ebp], 1
  jmp SHORT $LN5@main
$LN4@main:
  mov DWORD PTR tv68[ebp], 0
$LN5@main:
  mov eax, DWORD PTR tv68[ebp]
$LN9@main:
$LN11@main:
  ret 0

; filter:
$LN7@main:
Listing 6.32: MSVC 2012: two try blocks example

```
mov esp, DWORD PTR __SEHRec$[ebp]
push OFFSET $SG5488 ; 'access violation, can''t recover'
call printf
add esp, 4
mov DWORD PTR __SEHRec$[ebp+20], -2 ; previous try level

$LN6@main:
xor eax, eax
mov ecx, DWORD PTR __SEHRec$[ebp+8]
add esp, 4
push eax
pop ebp
ret 0

_main ENDP
```

```
$SG85486 DB 'in filter. code=0x%08X', 0AH, 00H
$SG85488 DB 'yes, that is our exception', 0AH, 00H
$SG85490 DB 'not our exception', 0AH, 00H
$SG85497 DB 'hello!', 0AH, 00H
$SG85499 DB 'access violation, can''t recover', 0AH, 00H
$SG85501 DB 'user exception caught', 0AH, 00H

xdata$x SEGMENT
__sehtable$main DD 0fffffffeH ; GS Cookie Offset
    DD 00H ; GS Cookie XOR Offset
    DD 0ffffffc8H ; EH Cookie Offset
    DD 00H ; EH Cookie Offset
    DD 0fffffffH ; previous try level for outer block
    DD FLAT:$LN19@main ; outer block filter
    DD FLAT:$LN9@main ; outer block handler
    DD 00H ; previous try level for inner block
    DD FLAT:$LN18@main ; inner block filter
    DD FLAT:$LN13@main ; inner block handler

xdata$x ENDS

$T2 = -40 ; size = 4
$T3 = -36 ; size = 4
_p$ = -32 ; size = 4
tv72 = -28 ; size = 4
__$SEHRec$ = -24 ; size = 24

_main PROC
    push ebp
    mov ebp, esp
    push -2 ; initial previous try level
    push OFFSET __sehtable$main
    push OFFSET __except_handler4
    mov eax, DWORD PTR fs:0
    push eax ; prev
    add esp, -24
    push ebx
    push esi
    push edi
    mov eax, DWORD PTR __security_cookie
    xor eax, DWORD PTR __SEHRec$[ebp+16], eax ; xored pointer to scope table
    xor eax, ebp ; ebp ^ security_cookie
    push eax
    lea eax, DWORD PTR __SEHRec$[ebp+8] ; pointer to VC_EXCEPTION_REGISTRATION_RECORD
    mov DWORD PTR fs:0, eax
    mov DWORD PTR __SEHRec$[ebp], esp
    mov DWORD PTR _p$[ebp], 0
    mov DWORD PTR __SEHRec$[ebp+20], 0 ; entering outer try block, setting previous try level=0
    mov DWORD PTR __SEHRec$[ebp+20], 1 ; entering inner try block, setting previous try level=1
    push OFFSET $SG85497 ; 'hello!'`````
call _printf
add esp, 4
push 0
push 0
push 0
push 1122867 ; 00112233H
call DWORD PTR __imp__RaiseException@16
push OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
call _printf
add esp, 4
mov eax, DWORD PTR __p$[ebp]
add esp, 4
mov DWORD PTR [edx], 13
mov DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, set previous try level back to 0
jmp SHORT $LN2$@main

; inner block filter:
$LN12$@main:
$LN18$@main:
    mov ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov edx, DWORD PTR [ecx]
    mov eax, DWORD PTR [edx]
    mov DWORD PTR $T3[ebp], eax
cmp DWORD PTR $T3[ebp], -1073741819 ; c0000005H
    jne SHORT $LN5$@main
    mov DWORD PTR tv72[ebp], 1
    jmp SHORT $LN6$@main
$LN5$@main:
    mov DWORD PTR tv72[ebp], 0
$LN6$@main:
    mov eax, DWORD PTR tv72[ebp]
$LN14$@main:
$LN16$@main:
    ret 0

; inner block handler:
$LN13$@main:
$LN19$@main:
    mov esp, DWORD PTR __$SEHRec$[ebp]
push OFFSET $SG85501 ; 'access violation, can''t recover'
call _printf
add esp, 4
mov DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try block, setting previous try level back to 0
$LN2$@main:
    mov DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both blocks, setting previous try level back to -2
    jmp SHORT $LN7$@main

; outer block filter:
$LN8$@main:
$LN19$@main:
    mov ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov edx, DWORD PTR [ecx]
    mov eax, DWORD PTR [edx]
    mov DWORD PTR $T2[ebp], eax
    mov ecx, DWORD PTR __$SEHRec$[ebp+4]
push ecx
    mov edx, DWORD PTR $T2[ebp]
push edx
call _filter_user_exceptions
add esp, 8
$LN10$@main:
$LN17$@main:
    ret 0

; outer block handler:
$LN9$@main:
$LN19$@main:
    mov esp, DWORD PTR __$SEHRec$[ebp]
push OFFSET $SG85503 ; 'user exception caught'
call _printf
add esp, 4
Here is the meaning of the **cookies**: Cookie Offset is the difference between the address of the saved EBP value in the stack and the EBP@security_cookie value in the stack. Cookie XOR Offset is an additional difference between the EBP@security_cookie value and what is stored in the stack.

If this equation is not true, the process is to halt due to stack corruption:

```
security_cookie@(CookieXOROffset + address_of_saved_EBP) == stack[address_of_saved_EBP + CookieOffset]
```

If Cookie Offset is −2, this implies that it is not present.

**Cookies** checking is also implemented in my tracer, see GitHub for details.

It is still possible to fall back to SEH3 in the compilers after (and including) MSVC 2005 by setting the /GS- option, however, the CRT code use SEH4 anyway.

**Windows x64**

As you might think, it is not very fast to set up the SEH frame at each function prologue. Another performance problem is changing the previous try level value many times during the function's execution.

So things are changed completely in x64: now all pointers to try blocks, filter and handler functions are stored in another PE segment .pdata, and from there the OS's exception handler takes all the information.

Here are the two examples from the previous section compiled for x64:

```
Listing 6.33: MSVC 2012

$SG86276 DB  'hello #1!', 0AH, 00H
$SG86277 DB  'hello #2!', 0AH, 00H
```
Listing 6.34: MSVC 2012

SG86279 DB 'access violation, can't recover', 0aH, 00H

pdata SEGMENT
$pdata$main DD imagerel $LN9
   DD imagerel $LN9+61
   DD imagerel $unwind$main
pdata ENDS

pdata SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
   DD imagerel main$filt$0+32
   DD imagerel $unwind$main$filt$0
pdata ENDS

xdata SEGMENT
$xdata$main DD 020609H
   DD 030023206H
   DD imagerel _C_specific_handler
   DD 01H
   DD imagerel $LN9+8
   DD imagerel $LN9+40
   DD imagerel main$filt$0
   DD imagerel $LN9+40
$xdata ENDS

_TEXT SEGMENT
main PROC
$LN9:
    push rbx
    sub rsp, 32
    xor ebx, ebx
    lea rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
    call printf
    mov DWORD PTR [rbx], 13
    lea rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
    call printf
    jmp SHORT $LN8@main
$LN6@main:
    lea rcx, OFFSET FLAT:$SG86279 ; 'access violation, can't recover'
    call printf
    npad 1 ; align next label
$LN8@main:
    xor eax, eax
    add rsp, 32
    pop rbx
    ret 0
main ENDP

_TEXT ENDS

text$x SEGMENT
main$filt$0 PROC
    push rbp
    sub rsp, 32
    mov rbp, rdx
$LN5@main$filt$0:
    mov rax, QWORD PTR [rcx]
    xor ecx, ecx
    cmp DWORD PTR [rax], -1073741819; c0000005H
    sete cl
    mov eax, ecx
$LN7@main$filt$0:
    add rsp, 32
    pop rbp
    ret 0
    int 3
main$filt$0 ENDP
text$x ENDS
$SG86277 DB 'in filter. code=0x%08X', 0aH, 00H
$SG86279 DB 'yes, that is our exception', 0aH, 00H
$SG86281 DB 'not our exception', 0aH, 00H
$SG86288 DB 'hello!', 0aH, 00H
$SG86290 DB '0x112233 raised. now let''s crash', 0aH, 00H
$SG86292 DB 'access violation, can''t recover', 0aH, 00H
$SG86294 DB 'user exception caught', 0aH, 00H

pdata SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
   DD      imagerel $LN6+73
   DD      imagerel $unwind$filter_user_exceptions
$pdata$main DD imagerel $LN14
   DD      imagerel $LN14+95
   DD      imagerel $unwind$main
pdata ENDS

xdata SEGMENT
$xdata$filter_user_exceptions DD 020601H
   DD 030023206H
$xdata$main DD 020609H
   DD 030023206H
   DD imagerel __C_specific_handler
   DD 02H
   DD imagerel $LN14+8
   DD imagerel $LN14+59
   DD imagerel main$filt$0
   DD imagerel $LN14+59
   DD imagerel $LN14+8
   DD imagerel $LN14+74
   DD imagerel main$filt$1
   DD imagerel $LN14+74
$xdata ENDS

_TEXT SEGMENT
main PROC
$LN14:
   push rbx
   sub rsp, 32
   xor ebx, ebx
   lea rcx, OFFSET FLAT:$SG86288 ; 'hello'
   call printf
   xor r9d, r9d
   xor r8d, r8d
   xor edx, edx
   mov ecx, 1122867 ; 00112233H
   call QWORD PTR __imp_RaiseException
   lea rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. now let''s crash'
   call printf
   mov DWORD PTR [rbx], 13
   jmp SHORT $LN13@main
$LN11@main:
   lea rcx, OFFSET FLAT:$SG86292 ; 'access violation, can''t recover'
   call printf
   npad 1 ; align next label
$LN13@main:
   jmp SHORT $LN9@main
$LN7@main:
lea   rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
call  printf
npad  1 ; align next label

$LN9@main:
xor   eax, eax
add   rsp, 32
pop   rbx
ret   0

main ENDP

text$x SEGMENT
main$filt$0 PROC
push  rbp
sub   rsp, 32
mov   rbp, rdx
$LN10@main$filt$:
    mov   rax, QWORD PTR [rcx]
xor   ecx, ecx
cmp   DWORD PTR [rax], -1073741819; c0000005H
sete  cl
mov   eax, ecx
$LN12@main$filt$:
    add   rsp, 32
pop   rbp
ret   0
int  3

main$filt$0 ENDP

main$filt$1 PROC
push  rbp
sub   rsp, 32
mov   rbp, rdx
$LN6@main$filt$:
    mov   rax, QWORD PTR [rcx]
    mov   rdx, rcx
    mov   ecx, DWORD PTR [rax]
call  filter_user_exceptions
npad  1 ; align next label
$LN8@main$filt$:
    add   rsp, 32
pop   rbp
ret   0
int  3

main$filt$1 ENDP

text$x ENDS

_CODE_ SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6:
    push  rbx
    sub   rsp, 32
    mov   ebx, ecx
    mov   edx, ecx
    lea   rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x%08X'
call  printf
cmp   ebx, 1122867; 00112233H
jne   SHORT $LN2@filter_use
    lea   rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our exception'
call  printf
    mov   eax, 1
    add   rsp, 32
    pop   rbx
    ret   0
$LN2@filter_use:
    lea   rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
call  printf
    xor   eax, eax
    add   rsp, 32

_TEXT SEGMENT

Aside from exception information, .pdata is a section that contains the addresses of almost all function starts and ends, hence it may be useful for a tools targeted at automated analysis.

**Read more about SEH**


### 6.5.4 Windows NT: Critical section

Critical sections in any OS are very important in multithreaded environment, mostly for giving a guarantee that only one thread can access some data in a single moment of time, while blocking other threads and interrupts.

That is how a CRITICAL_SECTION structure is declared in Windows NT line OS:

```c
typedef struct RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;
    // The following three fields control entering and exiting the critical
    // section for the resource
    //
    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;       // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;       // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

That's is how EnterCriticalSection() function works:

```assembly
_loc_7DE922DD:
```

---

46Also available as [http://go.yurichev.com/17294](http://go.yurichev.com/17294)

47Also available as [http://go.yurichev.com/17293](http://go.yurichev.com/17293)

48Also available as [http://go.yurichev.com/17294](http://go.yurichev.com/17294)
The most important instruction in this code fragment is BTR (prefixed with LOCK): the zeroth bit is stored in the CF flag and cleared in memory. This is an atomic operation, blocking all other CPUs’ access to this piece of memory (see the LOCK prefix before the BTR instruction). If the bit at LockCount is 1, fine, reset it and return from the function: we are in a critical section.
If not—the critical section is already occupied by other thread, so wait. The wait is performed there using WaitForSingleObject().

And here is how the LeaveCriticalSection() function works:

```
Listing 6.37: Windows 2008/ntdll.dll/x86 (begin)

_RtlLeaveCriticalSection@4 proc near
  arg_0  = dword ptr 8
  mov    edi, edi
  push   ebp
  mov    ebp, esp
  push   esi
  mov    esi, [ebp+arg_0]
  add    dword ptr [esi+8], 0FFFFFFFh ; RecursionCount
  jnz    short loc_7DE922B2
  push   ebx
  push   edi
  lea    edi, [esi+4] ; LockCount
  mov    dword ptr [esi+0Ch], 0
  mov    ebx, 1
  mov    eax, edi
  lock   xadd [eax], ebx
  inc    ebx
  cmp    ebx, 0FFFFFFFh
  jnz    loc_7DEA8EB7

loc_7DE922B0:
  pop    edi
  pop    ebx

loc_7DE922B2:
  xor    eax, eax
  pop    esi
  pop    ebp
  retn   4

... skipped
```

XADD is “exchange and add”.
In this case, it adds 1 to LockCount, meanwhile saves initial value of LockCount in the EBX register. However, value in EBX is to incremented with a help of subsequent INC EBX, and it also will be equal to the updated value of LockCount.

This operation is atomic since it is prefixed by LOCK as well, meaning that all other CPUs or CPU cores in system are blocked from accessing this point in memory.
The LOCK prefix is very important:
without it two threads, each of which works on separate CPU or CPU core can try to enter a critical section and to modify the value in memory, which will result in non-deterministic behavior.
Chapter 7

Tools

Now that Dennis Yurichev has made this book free (libre), it is a contribution to the world of free knowledge and free education. However, for our freedom’s sake, we need free (libre) reverse engineering tools to replace the proprietary tools described in this book.

Richard M. Stallman

7.1 Binary analysis

Tools you use when you don’t run any process.

- (Free, open-source) **ent**\(^1\): entropy analyzing tool. Read more about entropy: 9.2 on page 929.
- **Hiew**\(^2\): for small modifications of code in binary files. Has assembler/disassembler.
- (Free, open-source) **GHex**\(^3\): simple hexadecimal editor for Linux.
- (Free, open-source) **xxd** and **od**: standard UNIX utilities for dumping.
- (Free, open-source) **strings**: **NIX** tool for searching for ASCII strings in binary files, including executable ones. Sysinternals has alternative\(^4\) supporting wide char strings (UTF-16, widely used in Windows).
- (Free, open-source) **Binwalk**\(^5\): analyzing firmware images.
- (Free, open-source) **binary grep**: a small utility for searching any byte sequence in a big pile of files, including non-executable ones: [GitHub](https://github.com/). There is also rafind2 in rada.re for the same purpose.

7.1.1 Disassemblers

- **IDA**. An older freeware version is available for download\(^6\). Hot-keys cheatsheet: .6.1 on page 1029
- **Ghidra**\(^7\) — free alternative to IDA from [NSA](http://www.nsa.gov).
- **Binary Ninja**\(^8\)
- (Free, open-source) **zynamics BinNavi**\(^9\)
- (Free, open-source) **objdump**: simple command-line utility for dumping and disassembling.
- (Free, open-source) **readelf**\(^10\): dump information about ELF file.

\(^1\) [http://www.fourmilab.ch/random/](http://www.fourmilab.ch/random/)
\(^2\) [hiew.ru](http://hiew.ru)
\(^3\) [https://wiki.gnome.org/Apps/Ghex](https://wiki.gnome.org/Apps/Ghex)
\(^5\) [http://binwalk.org/](http://binwalk.org/)
\(^6\) [hex-rays.com/products/ida/support/download_freeware.shtml](http://hex-rays.com/products/ida/support/download_freeware.shtml)
\(^7\) [https://ghidra-sre.org/](https://ghidra-sre.org/)
\(^8\) [http://binary.ninja/](http://binary.ninja/)
\(^9\) [https://www.zynamics.com/binnavi.html](https://www.zynamics.com/binnavi.html)
\(^10\) [https://sourceware.org/binutils/docs/binutils/readelf.html](https://sourceware.org/binutils/docs/binutils/readelf.html)
7.1.2 Decompilers

The best known is Hex-Rays: http://hex-rays.com/products/decompiler/. Read more about it: 11.8 on page 988.

There is also a free alternative from NSA: Ghidra\(^{11}\).

7.1.3 Patch comparison/diffing

You may want to use it when you compare original version of some executable and patched one, in order to find what has been patched and why.

- (Free) zynamics BinDiff\(^{12}\)
- (Free, open-source) Diaphora\(^{13}\)

7.2 Live analysis

Tools you use on a live system or during running of a process.

7.2.1 Debuggers

- (Free) OllyDbg. Very popular user-mode win32 debugger\(^{14}\). Hot-keys cheatsheet: 6.2 on page 1030
- (Free, open-source) GDB. Not quite popular debugger among reverse engineers, because it’s intended mostly for programmers. Some commands: 6.5 on page 1031. There is a visual interface for GDB, “GDB dashboard”\(^{15}\).
- (Free, open-source) LLDB\(^{16}\).
- WinDbg\(^{17}\): kernel debugger for Windows.
- IDA has internal debugger.
- (Free, open-source) Radare AKA rada.re AKA r2\(^{18}\). A GUI also exists: ragui\(^{19}\).
- (Free, open-source) tracer. The author often uses tracer\(^{20}\) instead of a debugger.

The author of these lines stopped using a debugger eventually, since all he needs from it is to spot function arguments while executing, or registers state at some point. Loading a debugger each time is too much, so a small utility called tracer was born. It works from command line, allows intercepting function execution, setting breakpoints at arbitrary places, reading and changing registers state, etc.

N.B.: the tracer isn’t evolving, because it was developed as a demonstration tool for this book, not as everyday tool.

7.2.2 Library calls tracing

ltrace\(^{21}\).

7.2.3 System calls tracing

strace / dtruss

It shows which system calls (syscalls( 6.3 on page 738)) are called by a process right now.

For example:

---

\(^{11}\)https://ghidra-sre.org/
\(^{12}\)https://www.zynamics.com/software.html
\(^{13}\)https://github.com/joxeankoret/diaphora
\(^{14}\)ollydbg.de
\(^{15}\)https://github.com/cyrus-and/gdb-dashboard
\(^{16}\)http://lldb.llvm.org/
\(^{17}\)https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit
\(^{18}\)http://rada.re/r/
\(^{19}\)http://radare.org/ragui/
\(^{20}\)yurichev.com
\(^{21}\)http://www.ltrace.org/
Mac OS X has dtruss for doing the same.

Cygwin also has strace, but as far as it’s known, it works only for .exe-files compiled for the cygwin environment itself.

### 7.2.4 Network sniffing

**Sniffing** is intercepting some information you may be interested in.

(Free, open-source) **Wireshark**\(^{22}\) for network sniffing. It has also capability for USB sniffing\(^{23}\).

Wireshark has a younger (or older) brother **tcpdump**\(^{24}\), simpler command-line tool.

### 7.2.5 Sysinternals

(Free) Sysinternals (developed by Mark Russinovich) \(^{25}\). At least these tools are important and worth studying: Process Explorer, Handle, VMMap, TCPView, Process Monitor.

### 7.2.6 Valgrind

(Free, open-source) a powerful tool for detecting memory leaks: [http://valgrind.org/](http://valgrind.org/). Due to its powerful **JIT** mechanism, Valgrind is used as a framework for other tools.

### 7.2.7 Emulators

- (Free, open-source) **QEMU**\(^{26}\): emulator for various CPUs and architectures.
- (Free, open-source) **DosBox**\(^{27}\): MS-DOS emulator, mostly used for retrogaming.
- (Free, open-source) **SimH**\(^{28}\): emulator of ancient computers, mainframes, etc.

### 7.3 Other tools

**Microsoft Visual Studio Express** \(^{29}\): Stripped-down free version of Visual Studio, convenient for simple experiments.

Some useful options: [6.3 on page 1030](#).

There is a website named “Compiler Explorer”, allowing to compile small code snippets and see output in various GCC versions and architectures (at least x86, ARM, MIPS): [http://godbolt.org/](http://godbolt.org/)—I would have used it myself for the book if I would know about it!

### 7.3.1 SMT solvers

From the reverse engineer’s perspective, SMT solvers are used when dealing with amateur cryptography, symbolic/concolic execution, ROP chains generation.

For more information, read: [https://yurichev.com/writings/SAT_SMT_by_example.pdf](https://yurichev.com/writings/SAT_SMT_by_example.pdf).

---

\(^{22}\)[https://www.wireshark.org/]

\(^{23}\)[https://wiki.wireshark.org/CaptureSetup/USB]

\(^{24}\)[http://www.tcpdump.org/]


\(^{26}\)[http://qemu.org]

\(^{27}\)[https://www.dosbox.com/]

\(^{28}\)[http://simh.trailing-edge.com/]

7.3.2 Calculators

Good calculator for reverse engineer’s needs should support at least decimal, hexadecimal and binary bases, as well as many important operations like XOR and shifts.

- IDA has built-in calculator (“?”).
- rada.re has rax2.
- https://yurichev.com/progcalc/
- As a last resort, standard calculator in Windows has programmer’s mode.

7.4 Do You Think Something Is Missing Here?

If you know a great tool not listed here, please drop a note: <first_name @ last_name . com> or <first_name . last_name @ gmail . com>.
Chapter 8

Case studies

Instead of epigraph:

Peter Seibel: How do you tackle reading source code? Even reading something in a programming language you already know is a tricky problem.  
Donald Knuth: But it's really worth it for what it builds in your brain. So how do I do it? There was a machine called the Bunker Ramo 300 and somebody told me that the Fortran compiler for this machine was really amazingly fast, but nobody had any idea why it worked. I got a copy of the source-code listing for it. I didn't have a manual for the machine, so I wasn't even sure what the machine language was.

But I took it as an interesting challenge. I could figure out BEGIN and then I would start to decode. The operation codes had some two-letter mnemonics and so I could start to figure out “This probably was a load instruction, this probably was a branch.” And I knew it was a Fortran compiler, so at some point it looked at column seven of a card, and that was where it would tell if it was a comment or not.

After three hours I had figured out a little bit about the machine. Then I found these big, branching tables. So it was a puzzle and I kept just making little charts like I'm working at a security agency trying to decode a secret code. But I knew it worked and I knew it was a Fortran compiler—it wasn't encrypted in the sense that it was intentionally obscure; it was only in code because I hadn't gotten the manual for the machine.

Eventually I was able to figure out why this compiler was so fast. Unfortunately it wasn't because the algorithms were brilliant; it was just because they had used unstructured programming and hand optimized the code to the hilt.

It was just basically the way you solve some kind of an unknown puzzle— make tables and charts and get a little more information here and make a hypothesis. In general when I'm reading a technical paper, it's the same challenge. I'm trying to get into the author's mind, trying to figure out what the concept is. The more you learn to read other people's stuff, the more able you are to invent your own in the future, it seems to me.

(Peter Seibel — Coders at Work: Reflections on the Craft of Programming)

8.1 Mahjong solitaire prank (Windows 7)

Mahjong solitaire is a great game, but can we make it harder, by disabling the Hint menu item?

In my Windows 7, I can find Mahjong.dll and Mahjong.exe in:
C:|Windows\winsxs\x86_microsoft-windows-s..inboxgames-shanghai_31bf3856ad364e35_6.1.7600.16385_none|c07a51d9507d9398.

Also, the Mahjong.exe.mui file in:
C:|Windows\winsxs\x86_microsoft-windows-s..shanghai.resources_31bf3856ad364e35_6.1.7600.16385_en-us_c430954533c66bf3

and
C:|Windows\winsxs\x86_microsoft-windows-s..shanghai.resources_31bf3856ad364e35_6.1.7600.16385_ru-ru
I'm using English Windows, but with Russian language support, so there might be resource files for two languages. By opening Mahjong.exe.mui in Resource Hacker, there we can see a menu definition:

```
Listing 8.1: Menu resources from Mahjong.exe.mui
103 MENU
  LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
  {
    POPUP "&Game"
    {
      MENUITEM "&New Game	F2", 40000
      MENUITEM SEPARATOR
      MENUITEM "&Undo	Ctrl+Z", 40001
      MENUITEM "&Hint	H", 40002
      MENUITEM SEPARATOR
      MENUITEM "&Statistics	F4", 40003
      MENUITEM "&Options	F5", 40004
      MENUITEM "Change &Appearance	F7", 40005
      MENUITEM SEPARATOR
      MENUITEM "E&xit", 40006
    }
    POPUP "&Help"
    {
      MENUITEM "&View Help	F1", 40015
      MENUITEM "&About Mahjong Titans", 40016
      MENUITEM SEPARATOR
      MENUITEM "Get &More Games Online", 40020
    }
  }
```

The Hint submenu has the 40002 code. Now I'm opening Mahjong.exe in IDA and find the 40002 value.

(I'm writing this in November 2019. Somehow, IDA can't get PDBs from Microsoft servers. Maybe Windows 7 is unsupported anymore? Anyway, I can't get function names...)

```
Listing 8.2: Mahjong.exe
.text:010205C8 6A 03  push 3
.text:010205CA 85 FF  test  edi, edi
.text:010205CC 5B  pop  ebx

.text:01020625 57  push edi ; uIDEnableItem
.text:01020626 FF 35 C8 97 00 00  push hmenu ; hMenu
.text:01020622 CC 42 9C 00 00  mov edi, 40002
.text:01020637 75 18  jnz short loc_1020651 ; must be always
.text:01020639 6A 00  push 0 ; uEnable
.text:0102063B 57  push edi ; uIDEnableItem
.text:0102063C FF 35 B4 88 08 01  push hMenu ; hMenu
.text:01020642 FF 66  call esi ; EnableMenuItem
.text:01020644 6A 00  push 0 ; uEnable
.text:01020646 57  push edi ; uIDEnableItem
.text:01020647 FF 35 C8 97 00 01  push hmenu ; hMenu
.text:0102064D FF 66  call esi ; EnableMenuItem
.text:0102064F EB 1A  jmp short loc_102066B
.text:01020651.................................................................
.text:01020651 785
```

785
This piece of code enables or disables the Hint menu item.
And according to MSDN¹:

\[ \text{MF\_DISABLED} \mid \text{MF\_GRAYED} = 3 \] and \[ \text{MF\_ENABLED} = 0. \]

I think, this function enables or disables several menu items (Hint, Undo, etc), according to the value in \( \text{arg\_0} \). Because at start, when a user choose solitaire type, Hint and Undo are disabled. They are enabled when the game has begun.

So I’m patching the Mahjong.exe file at 0x01020637 by replacing 0x75 with 0xEB byte, making this JNZ jump working always. Effectively, this will make calling EnableMenuItem(\ldots, \ldots, 3) always. Now the Hint submenu is always disabled.

Also, somehow, EnableMenuItem() called twice, for hMenu and for hMenu. Perhaps, the program has two menus, and maybe switching them?

As a homework, try to disable Undo menu item, to make the game even harder.

### 8.2 Task manager practical joke (Windows Vista)

Let’s see if it’s possible to hack Task Manager slightly so it would detect more CPU cores.

Let us first think, how does the Task Manager know the number of cores?

There is the GetSystemInfo() win32 function present in win32 userspace which can tell us this. But it’s not imported in taskmgr.exe.

There is, however, another one in NTAPI, NtQuerySystemInformation(), which is used in taskmgr.exe in several places.

To get the number of cores, one has to call this function with the SystemBasicInformation constant as a first argument (which is zero ²).

The second argument has to point to the buffer which is getting all the information.

So we have to find all calls to the NtQuerySystemInformation(0, ?, ?, ?) function. Let’s open taskmgr.exe in IDA.

What is always good about Microsoft executables is that IDA can download the corresponding PDB file for this executable and show all function names.

It is visible that Task Manager is written in C++ and some of the function names and classes are really speaking for themselves. There are classes CAdapter, CNetPage, CPerfPage, CProcInfo, CProcPage, CSvcPage, CTaskPage, CUserPage.

Apparently, each class corresponds to each tab in Task Manager.

Let’s visit each call and add comment with the value which is passed as the first function argument. We will write “not zero” at some places, because the value there was clearly not zero, but something really different (more about this in the second part of this chapter).

And we are looking for zero passed as argument, after all.

---


²MSDN
Yes, the names are really speaking for themselves.

When we closely investigate each place where `NtQuerySystemInformation(0, ?, ?, ?)` is called, we quickly find what we need in the `InitPerfInfo()` function:

Listing 8.3: taskmgr.exe (Windows Vista)

```assembly
.text:10000B4B3 xor r9d, r9d
.text:10000B4B6 lea rdx, [rsp+0C78h+var_C58]; buffer
.text:10000B4BB xor ecx, ecx
.text:10000B4BD lea ebp, [r9+40h]
.text:10000B4C1 mov r8d, ebp
.text:10000B4C4 call cs:_imp_NtQuerySystemInformation; 0
.text:10000B4CA xor ebx, ebx
.text:10000B4CC cmp eax, ebx
.text:10000B4CE jge short loc_10000B4D7
.text:10000B4D0 ; CODE XREF: InitPerfInfo(void)+97
.text:10000B4D0 xor al, al
.text:10000B4D2 jmp loc_10000B5EA
.text:10000B4D7 ; CODE XREF: InitPerfInfo(void)+36
.text:10000B4D7 loc_10000B4D7:
.text:10000B4D7 move eax, [rsp+0C78h+var_C50]
.text:10000B4DB mov esi, ebx
.text:10000B4DD mov r12d, 3E80h
.text:10000B4E3 mov cs:?g_PageSize@3KA, eax; ulong g_PageSize
.text:10000B4E9 shr eax, 0Ah
.text:10000B4EC lea r13, ImageBase
.text:10000B4F3 imul eax, [rsp+0C78h+var_C4C]
.text:10000B4F8 cmp [rsp+0C78h+var_C20], bpl
.text:10000B4FD mov cs:?g_MEMMax@3 JA, rax; __int64 g_MEMMax
.text:10000B504 movzx eax, [rsp+0C78h+var_C20]; number of CPUs
.text:10000B509 cmova eax, ebp
.text:10000B50C cmp al, bl
.text:10000B50E mov cs:?g_cProcessors@3EA, al; uchar g_cProcessors
```

g_cProcessors is a global variable, and this name has been assigned by IDA according to the PDB loaded from Microsoft's symbol server.
The byte is taken from var_C20. And var_C58 is passed to NtQuerySystemInformation() as a pointer to the receiving buffer. The difference between 0xC20 and 0xC58 is 0x38 (56).

Let's take a look at format of the return structure, which we can find in MSDN:

```c
typedef struct _SYSTEM_BASIC_INFORMATION {
    BYTE Reserved1[24];
    PVOID Reserved2[4];
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION;
```

This is a x64 system, so each PVOID takes 8 bytes.

All reserved fields in the structure take $24 + 4 \times 8 = 56$ bytes.

Oh yes, this implies that var_C20 is the local stack is exactly the NumberOfProcessors field of the SYSTEM_BASIC_INFORMATION structure.

Let's check our guess. Copy taskmgr.exe from C:\Windows\System32 to some other folder (so the Windows Resource Protection will not try to restore the patched taskmgr.exe).

Let's open it in Hiew and find the place:

![Figure 8.2: Hiew: find the place to be patched](image)

Let's replace the MOVZX instruction with ours. Let's pretend we've got 64 CPU cores.

Add one additional NOP (because our instruction is shorter than the original one):

![Figure 8.3: Hiew: patch it](image)

And it works! Of course, the data in the graphs is not correct.

At times, Task Manager even shows an overall CPU load of more than 100%.
The biggest number Task Manager does not crash with is 64. Apparently, Task Manager in Windows Vista was not tested on computers with a large number of cores. So there are probably some static data structure(s) inside it limited to 64 cores.

8.2.1 Using LEA to load values

Sometimes, LEA is used in taskmgr.exe instead of MOV to set the first argument of NtQuerySystemInformation:

```
xor r9d, r9d
div dword ptr [rsp+4C8h+WndClass.lpfWndProc]
lea rdx, [rsp+4C8h+VersionInformation]
lea ecx, [r9+2] ; put 2 to ECX
mov r8d, 138h
mov ebx, eax
; ECX=SystemPerformanceInformation
    call cs:__imp_NtQuerySystemInformation ; 2
...

mov r8d, 30h
lea r9, [rsp+298h+var_268]
lea rdx, [rsp+298h+var_258]
lea ecx, [r8-20h] ; put 3 to ECX
; ECX=SystemTimeOfDayInformation
    call cs:__imp_NtQuerySystemInformation ; not zero
```
mov rbp, [rsi+8]
mov r8d, 20h
lea r9, [rsp+98h+arg_0]
lea rdx, [rsp+98h+var_78]
lea ecx, [r8+2Fh] ; put 0x4F to ECX
mov [rsp+98h+var_60], ebx
mov [rsp+98h+var_68], rbp
; ECX=SystemSuperfetchInformation
call cs:__imp_NtQuerySystemInformation ; not zero

Perhaps MSVC did so because machine code of LEA is shorter than MOV REG, 5 (would be 5 instead of 4). LEA with offset in -128..127 range (offset will occupy 1 byte in opcode) with 32-bit registers is even shorter (for lack of REX prefix)—3 bytes.

Another example of such thing is: 6.1.5 on page 729.
8.3 Color Lines game practical joke

This is a very popular game with several implementations in existence. We can take one of them, called BallTriX, from 1997, available freely at http://go.yurichev.com/17311. Here is how it looks:

Figure 8.5: This is how the game is usually looks like

\(^3\)Or at http://go.yurichev.com/17365 or http://go.yurichev.com/17366.
So let's see, is it be possible to find the random generator and do some trick with it. IDA quickly recognize
the standard _rand function in balltrix.exe at 0x00403DA0. IDA also shows that it is called only from one place:

We’ll call it “random”. Let’s not to dive into this function’s code yet.

This function is referred from 3 places.

Here are the first two:

Here is the third one:

So the function has only one argument.

10 is passed in first two cases and 5 in third. We can also notice that the board has a size of 10*10 and
there are 5 possible colors. This is it! The standard rand() function returns a number in the 0..0x7FFF
range and this is often inconvenient, so many programmers implement their own random functions which
returns a random number in a specified range. In our case, the range is 0..n−1 and n is passed as the sole
argument of the function. We can quickly check this in any debugger.

So let’s fix the third function call to always return zero. First, we will replace three instructions (PUSH/CALL/ADD)
by NOPs. Then we’ll add XOR EAX, EAX instruction, to clear the EAX register.
So what we did is we replaced a call to the `random()` function by a code which always returns zero.
Let's run it now:

![Figure 8.6: Practical joke works](image)

Oh yes, it works\(^4\).

But why are the arguments to the `random()` functions global variables? That's just because it's possible to change the board size in the game's settings, so these values are not hardcoded. The 10 and 5 values are just defaults.

### 8.4 Minesweeper (Windows XP)

For those who are not very good at playing Minesweeper, we could try to reveal the hidden mines in the debugger.

As we know, Minesweeper places mines randomly, so there has to be some kind of random number generator or a call to the standard `rand()` C-function.

What is really cool about reversing Microsoft products is that there are PDB file with symbols (function names, etc). When we load `winmine.exe` into IDA, it downloads the PDB file exactly for this executable and shows all names.

So here it is, the only call to `rand()` is this function:

```assembly
.text:01003940 ; _stdcall Rnd(x)
.text:01003940 _Rnd@4 proc near
.text:01003940 arg_0 = dword ptr 4
.text:01003940 arg_0 proc near
.text:01003940 arg_0 = dword ptr 4
.text:01003940 cdq
.text:01003940 idiv [esp+arg_0]
.text:01003940 mov eax, edx
.text:01003940 retn 4
.text:01003940 _Rnd@4 endp
```

IDA named it so, and it was the name given to it by Minesweeper's developers.

The function is very simple:

```c
int Rnd(int limit) {
```

\(^4\)Author of this book once did this as a joke for his coworkers with the hope that they would stop playing. They didn't.
return rand() % limit;
};

(There is no “limit” name in the PDB file; we manually named this argument like this.)

So it returns a random value from 0 to a specified limit.

Rnd() is called only from one place, a function called StartGame(), and as it seems, this is exactly the code which place the mines:

Minesweeper allows you to set the board size, so the X (xBoxMac) and Y (yBoxMac) of the board are global variables. They are passed to Rnd() and random coordinates are generated. A mine is placed by the OR instruction at 0x010036FA. And if it has been placed before (it’s possible if the pair of Rnd() generates a coordinates pair which has already been generated), then TEST and JNZ at 0x010036E6 jumps to the generation routine again.

cBombStart is the global variable containing total number of mines. So this is loop.

The width of the array is 32 (we can conclude this by looking at the SHL instruction, which multiplies one of the coordinates by 32).

The size of the rgBlk global array can be easily determined by the difference between the rgBlk label in the data segment and the next known one. It is 0x360 (864):

864/32 = 27.

So the array size is 27 * 32? It is close to what we know: when we try to set board size to 100 * 100 in Minesweeper settings, it fallbacks to a board of size 24 * 30. So this is the maximal board size here. And the array has a fixed size for any board size.

So let’s see all this in OllyDbg. We will ran Minesweeper, attaching OllyDbg to it and now we can see the memory dump at the address of the rgBlk array (0x01005340)².

So we got this memory dump of the array:

²All addresses here are for Minesweeper for Windows XP SP3 English. They may differ for other service packs.
We will click “Run” in OllyDbg to unfreeze the Minesweeper process, then we’ll clicked randomly at the Minesweeper window and trapped into mine, but now all mines are visible:

OllyDbg, like any other hexadecimal editor, shows 16 bytes per line. So each 32-byte array row occupies exactly 2 lines here.

This is beginner level (9*9 board).

There is some square structure can be seen visually (0x10 bytes).

We will click “Run” in OllyDbg to unfreeze the Minesweeper process, then we’ll clicked randomly at the Minesweeper window and trapped into mine, but now all mines are visible:

![Minesweeper Window](image)

Figure 8.7: Mines

By comparing the mine places and the dump, we can conclude that 0x10 stands for border, 0x0F—empty block, 0x8F—mine. Perhaps, 0x10 is just a *sentinel value*.

Now we’ll add comments and also enclose all 0x8F bytes into square brackets:

```
border:
01005340 10 0F 0F 8F|0F 0F 0F|0F 0F 0F|0F 0F 0F 8F|0F 0F 0F 0F
01005350 0F 0F 0F 0F|0F 0F 0F|0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F
01005360 10 0F 0F 0F|0F 0F 0F|0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F
01005370 0F 0F 0F 0F|0F 0F 0F|0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F
01005380 10 0F 0F 0F|0F 0F 0F|0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F
01005390 0F 0F 0F 0F|0F 0F 0F|0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F
010053A0 10 0F 0F 0F|0F 0F 0F|0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F
010053B0 0F 0F 0F 0F|0F 0F 0F|0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F
010053C0 10 0F 0F 0F|0F 0F 0F|0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F
010053D0 0F 0F 0F 0F|0F 0F 0F|0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F
010053E0 10 0F 0F 0F|0F 0F 0F|0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F
```
Now we’ll remove all *border* *bytes* (0x10) and what’s beyond those:

```
OF OF OF OF OF OF OF OF OF OF
OF OF OF OF OF OF OF OF OF OF
OF OF OF OF OF OF OF OF OF OF
OF OF OF OF OF OF OF OF OF OF
OF OF OF OF OF OF OF OF OF OF
OF OF OF OF OF OF OF OF OF OF
OF OF OF OF OF OF OF OF OF OF
OF OF OF OF OF OF OF OF OF OF
```

Yes, these are mines, now it can be clearly seen and compared with the screenshot.
What is interesting is that we can modify the array right in OllyDbg. We can remove all mines by changing all 0x8F bytes by 0x0F, and here is what we’ll get in Minesweeper:

![Figure 8.8: All mines are removed in debugger](image)

We can also move all of them to the first line:

![Figure 8.9: Mines set in debugger](image)

Well, the debugger is not very convenient for eavesdropping (which is our goal anyway), so we’ll write a small utility to dump the contents of the board:

```c
#include <windows.h>
#include <assert.h>
#include <stdio.h>

int main (int argc, char * argv[])
{
    int i, j;
    HANDLE h;
    DWORD PID, address, rd;
    BYTE board[27][32];
```
if (argc!=3) {
    printf ("Usage: %s <PID> <address>\n", argv[0]);
    return 0;
}
assert (argv[1]!=NULL);
assert (argv[2]!=NULL);
assert (sscanf (argv[1], "%d", &PID)==1);
assert (sscanf (argv[2], "%x", &address)==1);
h=OpenProcess (PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE, FALSE, PID);
if (h==NULL) {
    DWORD e=GetLastError();
    printf ("OpenProcess error: %08X\n", e);
    return 0;
};
if (ReadProcessMemory (h, (LPVOID)address, board, sizeof(board), &rd)!=TRUE) {
    printf ("ReadProcessMemory() failed\n");
    return 0;
};
for (i=1; i<26; i++) {
    if (board[i][0]==0x10 && board[i][1]==0x10) break; // end of board
    for (j=1; j<31; j++) {
        if (board[i][j]==0x10) break; // board border
        if (board[i][j]==0x8F) printf ("**");
        else printf (" ");
    }
    printf ("\n");
}
CloseHandle (h);

Just set the PID\(^6^\)\(^7^\) and the address of the array (0x01005340 for Windows XP SP3 English) and it will dump it\(^8^\).

It attaches itself to a win32 process by PID and just reads process memory at the address.

**8.4.1 Finding grid automatically**

This is kind of nuisance to set address each time when we run our utility. Also, various Minesweeper versions may have the array on different address. Knowing the fact that there is always a border (0x10 bytes), we can just find it in memory:

```c
// find frame to determine the address
process_mem=(BYTE*)malloc(process_mem_size);
assert (process_mem!=NULL);

if (ReadProcessMemory (h, (LPVOID)start_addr, process_mem, process_mem_size, &rd)!=TRUE) {
```

\(^6^\)Program/process ID  
\(^7^\)PID it can be seen in Task Manager (enable it in "View → Select Columns")  
\(^8^\)The compiled executable is here: beginners.re
printf ("ReadProcessMemory() failed\n");
return 0;
}

// for 9*9 grid.
// FIXME: slow!
for (i=0; i<process_mem_size; i++)
{
    if (memcmp(process_mem+i, "\x10\x10\x10\x10\x10\x10\x10\x10\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x0F\x10", 32) \n
        ==0)
        {
            // found
            address=start_addr+i;
            break;
        }
    }
if (address==0)
{
    printf ("Can't determine address of frame (and grid)\n");
    return 0;
}
else
{
    printf ("Found frame and grid at 0x%\x\n", address);
}

Full source code: https://beginners.re/current-tree/examples/minesweeper/minesweeper_cheater2.c.

8.4.2 Exercises

• Why do the border bytes (or sentinel values) (0x10) exist in the array?
  What they are for if they are not visible in Minesweeper's interface? How could it work without them?
• As it turns out, there are more values possible (for open blocks, for flagged by user, etc). Try to find the meaning of each one.
• Modify my utility so it can remove all mines or set them in a fixed pattern that you want in the Minesweeper process currently running.

8.5 Hacking Windows clock

Sometimes I do some kind of first April prank for my coworkers.
Let's find, if we could do something with Windows clock? Can we force to go clock hands backwards?
First of all, when you click on date/time in status bar, a C:\WINDOWS\SYSTEM32\TIMEDATE.CPL module gets executed, which is usual executable PE-file.
Let's see, how it draw hands? When I open the file (from Windows 7) in Resource Hacker, there are clock faces, but with no hands:
OK, what we know? How to draw a clock hand? All they are started at the middle of circle, ending with its border. Hence, we must calculate coordinates of a point on circle's border. From school-level mathematics we may recall that we have to use sine/cosine functions to draw circle, or at least square root. There are no such things in `TIMEDATE.CPL`, at least at first glance. But, thanks to Microsoft debugging PDB files, I can find a function named `CAnalogClock::DrawHand()`, which calls `Gdiplus::Graphics::DrawLine()` at least twice.

Here is its code:

```plaintext
.text:6E9BC7 ; private: enum Gdiplus::Status __thiscall CAnalogClock::DrawHand(class Gdiplus::Graphics *, int, struct ClockHand const &, class Gdiplus::Pen *)
.text:6E9BC7 ? DrawHand@CAnalogClock@@AAE?
.AWStatus@gdiplus@PAVGraphics@3@HABUClockHand@PAVPen@3@Z proc near
.text:6E9BC7 ; CODE XREF: CAnalogClock::ClockPaint(HDC *)+163
.text:6E9BC7 ; CAnalogClock::ClockPaint(HDC *)+18B
.text:6E9BC7
.text:6E9BC7 var_10 = dword ptr -10h
.text:6E9BC7 var_C = dword ptr -0Ch
.text:6E9BC7 var_8 = dword ptr -8
.text:6E9BC7 var_4 = dword ptr -4
.text:6E9BC7 arg_0 = dword ptr 8
.text:6E9BC7 arg_4 = dword ptr 0Ch
.text:6E9BC7 arg_8 = dword ptr 10h
.text:6E9BC7 arg_C = dword ptr 14h
.text:6E9BC7
.text:6E9BC7
.text:6E9BC7 mov edi, edi
.text:6E9BC7 push ebp
.text:6E9BC7 mov ebp, esp
.text:6E9BC7 sub esp, 10h
.text:6E9BC7 mov eax, [ebp+arg_4]
.text:6E9BC7 push ebx
.text:6E9BC7 push esi
.text:6E9BC7 push edi
```

Figure 8.10: Resource Hacker
.text:6E9BD05       cdq
.text:6E9BD06       push 3Ch
.text:6E9BD08       mov esi, ecx
.text:6E9BD0A       pop ecx
.text:6E9BD0B       cdq
.text:6E9BD0D       push 2
.text:6E9BD0F       lea ebx, table[edx*8]
.text:6E9BD0E       lea eax, [edx+1Eh]
.text:6E9BD09       cdq
.text:6E9BD0A       mov ecx, [ebp+arg_0]
.text:6E9BD0F       mov [ebp+var_4], ebx
.text:6E9BD03       lea eax, table[edx*8]
.text:6E9BD09       mov [ebp+arg_4], eax
.text:6E9BD0C       call ?SetInterpolationMode@Graphics@Gdiplus@@QAE
\_AWStatus@4InterpolationMode@2@@Z
\_Gdiplus::Graphics::SetInterpolationMode(Gdiplus::InterpolationMode)
.text:6E9BD01       mov eax, [esi+70h]
.text:6E9BD04       mov edi, [ebp+arg_8]
.text:6E9BD07       mov [ebp+var_10], eax
.text:6E9BD0A       mov eax, [esi+74h]
.text:6E9BD0D       mov [ebp+var_C], eax
.text:6E9BD10       mov eax, [edi]
.text:6E9BD12       sub eax, [edi+8]
.text:6E9BD15       push 8000  ; nDenominator
.text:6E9BD1A       push eax  ; nNumerator
.text:6E9BD1B       push dword ptr [ebp+4] ; nNumber
.text:6E9BD1E       mov ebx, ds: __imp_MulDiv@12 ; MulDiv(x,x,x)
.text:6E9BD24       call ebx  ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6E9BD26       add eax, [esi+74h]
.text:6E9BD29       push 8000  ; nDenominator
.text:6E9BD2E       mov [ebp+arg_8], eax
.text:6E9BD31       mov eax, [edi]
.text:6E9BD33       sub eax, [edi+8]
.text:6E9BD36       push eax  ; nNumerator
.text:6E9BD37       mov eax, [ebp+var_4]
.text:6E9BD3A       push dword ptr [eax] ; nNumber
.text:6E9BD3C       call ebx  ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6E9BD3E       add eax, [esi+70h]
.text:6E9BD41       mov ecx, [ebp+arg_0]
.text:6E9BD44       mov [ebp+var_8], eax
.text:6E9BD47       mov eax, [ebp+arg_8]
.text:6E9BD4A       mov [ebp+var_4], eax
.text:6E9BD4D       lea eax, [ebp+var_8]
.text:6E9BD50       push eax
.text:6E9BD51       lea eax, [ebp+var_10]
.text:6E9BD54       push eax
.text:6E9BD55       push [ebp+arg_C]
.text:6E9BD58       call ?DrawLine@Graphics@Gdiplus@@QAE
\_AWStatus@2@PBVPen@2@ABVPoint@2@1@Z ; Gdiplus::Graphics::DrawLine(Gdiplus::Pen const *
*,Gdiplus::Point const &,Gdiplus::Point const &)
.text:6E9BD5D       mov ecx, [edi+8]
.text:6E9BD60       test ecx, ecx
.text:6E9BD62       jbe short loc_6E99DCAA
.text:6E9BD64       test eax, eax
.text:6E9BD66       jnz short loc_6E99DCAA
.text:6E9BD68       mov eax, [ebp+arg_4]
.text:6E9BD6B       push 8000  ; nDenominator
.text:6E9BD70       push ecx  ; nNumerator
.text:6E9BD71       push dword ptr [eax+4] ; nNumber
.text:6E9BD74       call ebx  ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6E9BD76       add eax, [esi+74h]
.text:6E9BD79       push 8000  ; nDenominator
.text:6E9BD7E       push dword ptr [edi+8] ; nNumerator
.text:6E9BD81       mov [ebp+arg_8], eax
.text:6E9BD84       mov eax, [ebp+arg_4]
.text:6E9BD87       push dword ptr [eax] ; nNumber
.text:6E9BD89       call ebx  ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6E9BD8B       add eax, [esi+70h]
.text:6E9BD8E       mov ecx, [ebp+arg_0]
We can see that `DrawLine()` arguments are dependent on result of `MulDiv()` function and a `table[]` table (name is mine), which has 8-byte elements (look at `LEA`'s second operand).

What is inside of `table[]`?

It’s referenced only from `DrawHand()` function. It has 120 32-bit words or 60 32-bit pairs... wait, 60? Let’s take a closer look at these values. First of all, I’ll zap 6 pairs or 12 32-bit words with zeros, and then I’ll put patched `TIMEDATE.CPL` into `C:\WINDOWS\SYSTEM32`. (You may need to set owner of the `*TIMEDATE.CPL` file to your primary user account (instead of `TrustedInstaller`), and also, boot in safe mode with command prompt so you can copy the file, which is usually locked.)
Now when any hand is located at 0..5 seconds/minutes, it’s invisible! However, opposite (shorter) part of second hand is visible and moving. When any hand is outside of this area, hand is visible as usual.

Let’s take even closer look at the table in Mathematica. I have copypasted table from the TIMEDATE.CPL to a tbl file (480 bytes). We will take for granted the fact that these are signed values, because half of elements are below zero (0xFFFFE0C1h, etc.). If these values would be unsigned, they would be suspiciously huge.

```plaintext
In[] := tbl = BinaryReadList["~/.../tbl", "Integer32"]
Out[] = {0, -7999, 836, -7956, 1663, -7825, 2472, -7608, 3253, -7308, 3999, \
        -6928, 4702, -6472, 5353, -5945, 5945, -5353, 6472, -4702, 6928, \
        -4000, 7308, -3253, 7608, -2472, 7825, -1663, 7956, -836, 8000, 0, \
        7956, 836, 7825, 1663, 7608, 2472, 7308, 3253, 6928, 4000, 6472, \
        -4702, 5945, 5353, 5353, 5945, 4702, 6472, 3999, 6928, 3253, 7308, \
        -2472, 7608, -3253, 7308, -4000, 6928, -4702, 6472, -5353, 5945, \
        -5945, 5353, -6472, 4702, -6928, 3999, -7308, 3253, -7608, 2472, \
        -7825, 1663, -7956, 836, -7999, 0, -7956, -836, -7825, -1663, -7608, \
        -2472, -7308, -3253, -6928, -4000, -6472, -4702, -5945, -5353, -5353, \
        -5945, -4702, -6472, -3999, -6928, -3253, -7308, -2472, -7608, -1663, \
        -7825, -836, -7956}

In[] := Length[tbl]
Out[] = 120

Let’s treat two consecutive 32-bit values as pair:

```plaintext
In[] := pairs = Partition[tbl, 2]
Out[] = {{0, -7999}, {836, -7956}, {1663, -7825}, {2472, -7608}, \
        {3253, -7308}, {3999, -6928}, {4702, -6472}, {5353, -5945}, {5945, \
        -5353}, {6472, -4702}, {6928, -4000}, {7308, -3253}, {7608, -2472}, \
        {7825, -1663}, {7956, -836}, {8000, 0}, {7956, 836}, {7825, \
        1663}, {7608, 2472}, {7308, 3253}, {6928, 4000}, {6472, \
        4702}, {5945, 5353}, {5353, 5945}, {4702, 6472}, {3999, \
        6928}, {3253, 7308}, {2472, 7608}, {1663, 7825}, {836, 7956}, {0, \
        7999}, {-836, 7956}, {-1663, 7825}, {-2472, 7608}, {-3253, \
        7308}, {-4000, 6928}, {-4702, 6472}, {-5353, 5945}, {-5945, \
        5353}, {-6472, 4702}, {-6928, 3999}, {-7308, 3253}, {-7608, \
        2472}, {-7825, 1663}, {-7956, 836}, {-7999, \
        0}, {-7956, -836}, {-7825, -1663}, {-7608, -2472}, {-7308, -3253}, \ 
        {-6928, -4000}, {-6472, -4702}, {-5945, -5353}, {-5353, -5945}, \ 
        {-4702, -6472}, {-3999, -6928}, {-3253, -7308}, {-2472, -7608}, \ 
        {-1663, -7825}, {-836, -7956}}

In[] := Length[pairs]
Out[] = 804
Let's try to treat each pair as X/Y coordinate and draw all 60 pairs, and also first 15 pairs:

![Figure 8.12: Mathematica](image)

Now this is something! Each pair is just coordinate. First 15 pairs are coordinates for $\frac{1}{4}$ of circle.

Perhaps, Microsoft developers precalculated all coordinates and put them into table. This is widespread, though somewhat old school practice – precalculated table access is faster than calling relatively slow sine/cosine functions. Sine/cosine operations are not that expensive anymore...

Now I can understand why when I zapped first 6 pairs, hands were invisible at that area: in fact, hands were drawn, they just had zero length, because hand started at 0:0 coordinate and ended there.

---

9Today this is known as memoization
The prank (practical joke)

Given all that, how would we force hands to go counterclockwise? In fact, this is simple, we need just to rotate the table, so each hand, instead of drawing at place of zeroth second, would be drawing at place of 59th second.

I made the patcher a long time ago, at the very beginning of 2000s, for Windows 2000. Hard to believe, it still works for Windows 7, perhaps, the table hasn’t been changed since then!

The patcher source code: https://beginners.re/current-tree/examples/timedate/time_pt.c.

Now I can see all hands goes backwards:

![Figure 8.13: Now it works](image)

Well, there is no animation in this book, but if you look closer, you can see, that hands are in fact shows correct time, but the whole clock face is rotated vertically, like we see it from the inside of clock.

Windows 2000 leaked source code

So I did the patcher and then Windows 2000 source code has been leaked (I can’t force you to trust me, though). Let’s take a look on source code if that function and table.

The file is `win2k/private/shell/cpls/utc/clock.c`:

```c
// Array containing the sine and cosine values for hand positions.
//
POINT rCircleTable[] =
{
    { 0, -7999},
    { 836, -7956},
    { 1663, -7825},
    { 2472, -7608},
    { 3253, -7308},
    ...
    { -4702, -6472},
    { -3999, -6928},
    { -3253, -7308},
    { -2472, -7608},
    { -1663, -7825},
    { -836 , -7956},
};

// Draws the hands of the clock.
```
Now it's clear: coordinates has been precalculated as if clock face has height and width of $2 \cdot 8000$, and then it's rescaled to current clock face radius using \texttt{MulDiv()} function.

\textbf{POINT} structure\textsuperscript{10} is a structure of two 32-bit values, first is \texttt{x}, second is \texttt{y}.

\section{8.6 (Windows 7) Solitaire: practical jokes}

\subsection{8.6.1 51 cards}

This is a joke I did once for my coworkers who played a Solitaire game too much. I was wondering, if it's possible to remove some cards, or maybe add (duplicates).

I opened Solitaire.exe in IDA disassembler, which asked if to download PDB file for it from Microsoft servers. This is usually a rule for many Windows executables and DLLs. At least, PDB has all function names.

Then I tried to find a 52 number in all functions (because this card game uses 52 cards). As it turned out, only 2 functions has it.

The first is:

\begin{verbatim}
.text:00000001000393B4 ; __int64 __fastcall SolitaireGame::OnMoveComplete(SolitaireGame *this)
.text:00000001000393B4 ?OnMoveComplete@SolitaireGame@@QEAAHXZ proc near ...
\end{verbatim}

The second is the function with self-describing name (name pulled from PDB by IDA): \texttt{InitialDeal()}: 

\begin{verbatim}
.text:00000001000365F8 ; void __fastcall SolitaireGame::InitialDeal(SolitaireGame *__hidden /uni2926/rcurvearrowse this)
.text:00000001000365F8 ?InitialDeal@SolitaireGame@@QEAAXXZ proc near
.text:00000001000A55C2
\end{verbatim}

mov rax, rsp
push rdi
push r12
push r13
sub rsp, 60h
mov [rsp+78h+var_48], 0FFFFFFFFFFFFFFFEh
mov [rax+8h], rbx
mov [rax+10h], rbp
mov [rax+18h], rsi
movaps xmmword ptr [rax-28h], xmm6
mov rsi, rcx
xor edx, edx ; struct Card *
call ?SetSelectedCard@SolitaireGame@@QEAAXPEAVCard@@@Z ; SolitaireGame::SetSelectedCard(Card *)
and qword ptr [rsi+0F0h], 0
mov rax, cs:?
g_pSolitaireGame@@3PEAVSolitaireGame@@EA ; SolitaireGame * g_pSolitaireGame
mov rdx, [rax+48h]
cmp byte ptr [rdx+51h], 0
jz short loc_10003664E
xor r8d, r8d ; bool
mov dl, 1 ; int
lea ecx, [r8+3] ; this
call ?PlaySoundProto@GameAudio@@YA_NH_NPEAI@Z ; GameAudio::PlaySoundProto(int,bool,uint *)
loc_10003664E: ; CODE XREF: SolitaireGame::InitialDeal(void)+46
mov rbx, [rsi+88h]
mov r8d, 4
lea rdx, aCardstackCreat ; "CardStack::CreateDeck()"
::uiNumSuits == "..."
mov ebp, 10000h
cmp byte ptr [rdx+51h], 0
jz short loc_10003664E
mov ecx, ebp ; unsigned int
call ?Log@@YAXIPEBGZZ ; Log(uint,ushort const *,...)
mov r8d, 52 ; ---
lea rdx, aCardstackCreat_0 ; "CardStack::CreateDeck()"
::uiNumCards == "..."
mov ecx, ebp ; unsigned int
call ?Log@@YAXIPEBGZZ ; Log(uint,ushort const *,...)
xor edi, edi
loc_100036684: ; CODE XREF: SolitaireGame::InitialDeal(void)+C0
mov eax, 4EC4EC4Fh
mul edi
mov r8d, edx
shr r8d, 4 ; unsigned int
mov eax, r8d
imul eax, 52 ; ---
mov edx, edi
sub edx, eax ; unsigned int
mov rcx, [rbx+128h] ; this
call ?CreateCard@CardTable@@IEAAPEAVCard@@II@Z ; CardTable::CreateCard(uint,uint)
mov rdx, rax ; struct Card *
call ?Push@CardStack@@QEAAXPEAVCard@@@Z ; CardStack::Push(Card *)
inc edi
cmp edi, 52 ; ---
jb short loc_100036684
mov r8d, r8d ; bool
xor edx, edx ; bool
mov rcx, rbx ; this
call ?Arrange@CardStack@@QEAAXN0@Z ; CardStack::Arrange(bool,bool)
lea rdx, aCardstackShuff ; "CardStack::Shuffle()"
Anyway, we clearly see a loop of 52 iterations. A loop body has calls to CardTable::CreateCard() and CardStack::Push().

The CardTable::CreateCard() eventually calls Card::Init() with values in 0..51 range, as one of its arguments. This can be easily checked using debugger.

So I tried just to change the 52 (0x34) number to 51 (0x33) in the cmp edi, 52 instruction at 0x1000366B5 and run it. At first glance, nothing happened, but I noticed that now it’s hard to solve the game. I spent almost an hour to reach this position:

Ace of hearts is missing. Perhaps, internally, this card is numbered as 51th (if to number them from zero).

In the other place I found all card names. Maybe names to be used to fetch card graphics from resources?
data:00000001000869C0 dq offset aQueenofclubs ; "QueenOfClubs"
data:00000001000869C8 dq offset aKingofclubs ; "KingOfClubs"
data:00000001000869D0 dq offset aAceofclubs ; "AceOfClubs"
data:00000001000869D8 dq offset aTwoofdiamonds ; "TwoOfDiamonds"
data:00000001000869E0 dq offset aThreeofdiamond ; "ThreeOfDiamonds"
data:00000001000869E8 dq offset aFourofdiamonds ; "FourOfDiamonds"
data:00000001000869F0 dq offset aFiveofdiamonds ; "FiveOfDiamonds"
data:00000001000869F8 dq offset aSixofdiamonds ; "SixOfDiamonds"
data:0000000100086A00 dq offset aSevenofdiamonds ; "SevenOfDiamonds"
data:0000000100086A08 dq offset aEightofdiamonds ; "EightOfDiamonds"
data:0000000100086A10 dq offset aNineofdiamonds ; "NineOfDiamonds"
data:0000000100086A18 dq offset aTenofdiamonds ; "TenOfDiamonds"
data:0000000100086A20 dq offset aJackofdiamonds ; "JackOfDiamonds"
data:0000000100086A28 dq offset aQueenofdiamonds ; "QueenOfDiamonds"
data:0000000100086A30 dq offset aKingofdiamonds ; "KingOfDiamonds"
data:0000000100086A38 dq offset aAceofdiamonds ; "AceOfDiamonds"
data:0000000100086A40 dq offset aTwoofspades ; "TwoOfSpades"
data:0000000100086A48 dq offset aThreeofspades ; "ThreeOfSpades"
data:0000000100086A50 dq offset aFourofspades ; "FourOfSpades"
data:0000000100086A58 dq offset aFivespades ; "FiveOfSpades"
data:0000000100086A60 dq offset aSixofspades ; "SixOfSpades"
data:0000000100086A68 dq offset aSevenspades ; "SevenOfSpades"
data:0000000100086A70 dq offset aEightspades ; "EightOfSpades"
data:0000000100086A78 dq offset aNinеспades ; "NineOfSpades"
data:0000000100086A80 dq offset aTenspades ; "TenOfSpades"
data:0000000100086A88 dq offset aJacks ; "JackOfSpades"
data:0000000100086A90 dq offset aQueens ; "QueenOfSpades"
data:0000000100086A98 dq offset aKings ; "KingOfSpades"
data:0000000100086AA0 dq offset aAces ; "AceOfSpades"
data:0000000100086AA8 dq offset aTwos ; "TwoOfHearts"
data:0000000100086AB0 dq offset aThrees ; "ThreeOfHearts"
data:0000000100086AB8 dq offset aFours ; "FourOfHearts"
data:0000000100086AC0 dq offset aFives ; "FiveOfHearts"
data:0000000100086AC8 dq offset aSixes ; "SixOfHearts"
data:0000000100086AD0 dq offset aSevens ; "SevenOfHearts"
data:0000000100086ADD dq offset aEights ; "EightOfHearts"
data:0000000100086AE0 dq offset aNines ; "NineOfHearts"
data:0000000100086AEE dq offset aTens ; "TenOfHearts"
data:0000000100086AF0 dq offset aJacks ; "JackOfHearts"
data:0000000100086AF8 dq offset aQueens ; "QueenOfHearts"
data:0000000100086B00 dq offset aKings ; "KingOfHearts"
data:0000000100086B08 dq offset aAces ; "AceOfHearts"
Hearts"
.dq offset a47108Cardnames ; "|47108|CardNames|Jack Of
Hearts"
.dq offset a55659Cardnames ; "|55659|CardNames|Queen Of
Hearts"
.dq offset a44572Cardnames ; "|44572|CardNames|King Of
Hearts"
.dq offset a44183Cardnames ; "|44183|CardNames|Ace Of
Hearts"

If you want to do this to someone, be sure his/her mental health is stable.

Aside of function names from PDB file, there are lots of Log() function calls that can help significantly, because the Solitaire game reports about what it’s doing right now.

Homework: try to remove several cards or two of clubs. And what if to swap card names in arrays of strings?

I also tried to pass a numbers like 0, 0..50 to Card:Init() (so to have 2 zeroes in a list of 52 numbers). Then I saw two two of clubs cards at one moment, but Solitaire behaves erratically.

This is patched Windows 7 Solitare: Solitaire51.exe.

8.6.2 53 cards

Now take a look on the first part of the loop:

v5 = 0;
do {
  v6 = CardTable::CreateCard(v4[37], v5 % 0x34, v5 / 0x34);
  CardStack::Push((CardStack *)v4, v6);
  ++v5;
} while (v5 < 0x34);

Somehow, CreateCard() functions takes two arguments: iterator divided by 52 and a remainder of the division operation. Hard to say, why they did so. Solitaire can’t allow more than 52 cards, so the last argument is senseless, it’s always zero.

But when I patch cmp edi, 52 instruction at 0x1000366B5 to be cmp edi, 53, I found that there are now 53 cards. The last one is two of clubs, because it’s numbered as 0th card.

During the last iteration, 0x52 is divided by 0x52, remainder is zero, so 0th card is added twice.

What a frustration, there are two two of clubs:
This is patched Windows 7 Solitaire: Solitaire53.

8.7 FreeCell prank (Windows 7)

This is a prank I once played for my coworkers who played FreeCell solitaire too much. Can we make FreeCell always deal the same game each time? Like, you know, in “Groundhog Day” movie?

(I’m writing this in November 2019. Somehow, IDA can’t get PDBs from Microsoft servers. Maybe Windows 7 is unsupported anymore? Anyway, I can’t get function names...)

8.7.1 Part I

So I loaded FreeCell.exe into IDA and found that both rand(), srand() and time() are imported from msvcrtd.dll. time() is indeed used as a seed for srand():

```assembly
.text:01029612  sub_1029612  proc near ; CODE XREF:
 .text:01029612 8B FF
 .text:01029614 56
 .text:01029615 57
 .text:01029616 6A 00
 .text:01029618 8B F9
 .text:0102961A FF 15 80 16 00+
 .text:01029620 50
 .text:01029621 FF 15 84 16 00+
 .text:01029622 8B 35 AC 16 00+
 .text:0102962D 59
 .text:0102962E 59
 .text:0102962F 8F D6
 .text:01029631 FF D6
 mov edi, edi
 push esi
 push edi
 push 0
 mov edi, ecx
 call ds:time
 push eax
 call ds:srand
 mov esi, ds:srand
 pop ecx
 pop ecx
 call esi ; rand
 call esi ; rand
```
Several (redundant) calls to rand() are funny, that reminds me:

“In the morning you will send for a hansom, desiring your man to take neither the first nor the second which may present itself.” (The Memoirs of Sherlock Holmes, by Arthur Conan Doyle11)

There is another call of time() and srand() pair, but my tracer showed that this is the point of our interest:

```plaintext
tracer.exe -l:FreeCell.exe bpf=msvcrt.dll!time bpf=msvcrt.dll!srand,args:1
```

```plaintext
TID=5340|(0) msvcrt.dll!time() (called from FreeCell.exe!BASE+0x29620 (0x29620))
TID=5340|(0) msvcrt.dll!time() -> 0x5ddf68aa
TID=5340|(1) msvcrt.dll!srand(0x5ddf68aa) (called from FreeCell.exe!BASE+0x29627 (0x29627))
TID=5340|(1) msvcrt.dll!srand() -> 0x5507e0
TID=5340|(1) msvcrt.dll!srand(0x399f) (called from FreeCell.exe!BASE+0x27d3a (0x207d3a))
TID=5340|(1) msvcrt.dll!srand() -> 0x3707e0
```

You see, the time() function returned 0x5ddf68aa and the very same value is used as an argument for srand().

Let's try to force time() to always return 0:

```plaintext
tracer.exe -l:FreeCell.exe bpf=msvcrt.dll!time,rt:0 bpf=msvcrt.dll!srand,args:1
```

```plaintext
TID=2104|(0) msvcrt.dll!time() (called from FreeCell.exe!BASE+0x29620 (0xb19620))
TID=2104|(0) msvcrt.dll!time() -> 0x5ddf68f6
TID=2104|(0) Modifying EAX register to 0x0
TID=2104|(1) msvcrt.dll!srand(0x0) (called from FreeCell.exe!BASE+0x29627 (0xb19627))
TID=2104|(1) msvcrt.dll!srand() -> 0x3707e0
TID=2104|(1) msvcrt.dll!srand(0x52f6) (called from FreeCell.exe!BASE+0x27d3a (0xb17d3a))
TID=2104|(1) msvcrt.dll!srand() -> 0x3707e0
```

Now I'm seeing the same game each time I'm running FreeCell using tracer:

11http://www.gutenberg.org/files/834/834-0.txt
Now how to patch the executable?

We want to pass 0 as an argument to srand() at 0x01029620. But there is a one-byte instruction: PUSH EAX. PUSH 0 is two-byte instruction. How to squeeze it into?

What is in other registers at this moment? Using tracer I’m dumping all them:

```
tracer.exe -l:FreeCell.exe bpx=FreeCell.exe!0x01029620
...
TID=4448|(0) FreeCell.exe!0x01029620
EAX=0x5dd6ac4 EBX=0x00000000 ECX=0x00000000 EDX=0x00000000
ESI=0x054732d0EDI=0x054732d0 EBP=0x0020f2bc ESP=0x0020f298
EIP=0x00899620
FLAGS=PF ZF IF
TID=4448|(0) FreeCell.exe!0x01029620
EAX=0x5dd6ac8 EBX=0x00000002 ECX=0x00000000 EDX=0x00000000
ESI=0xffffffff1EDI=0x054732d0 EBP=0x0020da78 ESP=0x0020d9d4
EIP=0x00899620
FLAGS=PF ZF IF
TID=4448|(0) FreeCell.exe!0x01029620
EAX=0x5dd6aca EBX=0x00000002 ECX=0x00000000 EDX=0x00000000
ESI=0x7740c460 EDI=0x054732d0 EBP=0x0020da78 ESP=0x0020d9d4
EIP=0x00899620
FLAGS=PF ZF IF
...
```

No matter how often I restart the game, ECX and EDX are seems to be always 0. So I patching PUSH EAX at 0x01029620 to PUSH EDX (also one-byte instruction), and now FreeCell always shows the same game to the player.

However, other options could exist. As a matter of fact, we don’t need to pass 0 to srand(). Rather, we
want to pass a constant to srand() to make game the same each time. As we can see, EDI's value hasn't been changing. Maybe we could try it as well.

Now a bit harder patching. Let's open FreeCell.exe in Hiew:

We have no space to replace one-byte PUSH EAX with two-byte PUSH 0. And we can't simply fill CALL ds:time with NOPs, because there is a FIXUP (address of time() function in msvcrtd.dll). (Hiew marked these 4 bytes are gray bytes.) So what I'm doing: patching first 2 bytes to EB 04. This is a JMP to bypass 4 FIXUP-ed bytes:

Then I replace PUSH EAX with NOP. So that srand() would have its zeroes arguments from PUSH 0 above. Also, I patch one of POP ECX to NOP, because I removed one PUSH.
Now Windows loader will write 4-byte FIXUP at 0x0102961C, but we don’t care: time()’s address will not be used anymore.

### 8.7.2 Part II: breaking the Select Game submenu

The user can still choose different game in the menu. Let’s see if srand() is still called. I’m trying to enter 1/2/3 in “Select Game” dialog box:

```plaintext
tracer.exe -l:FreeCell.exe bpf=msvcrtd.dll!srand,args:1
```

...  
TID=4936|(0) msvcrtd.dll!srand(0x5dab6df9) (called from FreeCell.exe!BASE+0x29627 (0xb49627))  
TID=4936|(0) msvcrtd.dll!srand() -> 0x5907e0  
TID=4936|(0) msvcrtd.dll!srand(0x2b40) (called from FreeCell.exe!BASE+0x27d3a (0xb47d3a))  
TID=4936|(0) msvcrtd.dll!srand() -> 0x5907e0  
TID=4936|(0) msvcrtd.dll!srand(0x1) (called from FreeCell.exe!BASE+0x27d3a (0xb47d3a))  
TID=4936|(0) msvcrtd.dll!srand() -> 0x5907e0  
TID=4936|(0) msvcrtd.dll!srand(0x2) (called from FreeCell.exe!BASE+0x27d3a (0xb47d3a))  
TID=4936|(0) msvcrtd.dll!srand() -> 0x5907e0  
TID=4936|(0) msvcrtd.dll!srand(0x3) (called from FreeCell.exe!BASE+0x27d3a (0xb47d3a))  
TID=4936|(0) msvcrtd.dll!srand() -> 0x5907e0  

Yes, the number user enters is just an argument for srand(). Where it is called?

```
.text:01027CBA loc_1027CBA: ; CODE XREF: sub_1027AC6  
\ \ +179  
.text:01027CBA 83 FF FC cmp edi, 0FFFFFFFC  
.text:01027CBD 75 74 jnz short loc_1027D33  
```

```
.text:01027D33 loc_1027D33: ; CODE XREF: sub_1027AC6  
\ \ +1F7  
.text:01027D33 57 push edi ; Seed  
.text:01027D34 FF 15 84 16 00+ call ds:srand  
.text:01027D3A 59 pop ecx  
.text:01027D3B 6A 34 push 34h  
.text:01027D3D 5B pop ebx  
.text:01027D3E 33 C0 xor eax, eax  
```

I couldn’t patch one-byte PUSH EDI to two-byte PUSH 0. But I see that there is only one single jump to loc_1027D33 from the above.

I’m patching CMP EDI, ... to XOR EDI, EDI, padding the 3rd byte to NOP. I’m patching also JNZ to JMP, so that jump will always occur.

Now FreeCell ignores the number user enters, but suddenly, there is also the same game at start:
It seems that the code we patched in part I is somehow connected to a code after 0x01027CBD, that executes if EDI==0xFFFFFFFF. Anyway, our goal is accomplished — the game is always the same at the start and the user can’t choose another using the menu.

8.8 Dongles

The author of these lines, occasionally did software copy-protection dongle replacements, or “dongle emulators” and here are couple examples of how it’s happening.

About one of the cases about Rocket and Z3 that is not present here, you can read here: [http://yurichev.com/tmp/SAT_SMT_DRAFT.pdf](http://yurichev.com/tmp/SAT_SMT_DRAFT.pdf).

8.8.1 Example #1: MacOS Classic and PowerPC

Here is an example of a program for MacOS Classic, for PowerPC. The company who developed the software product has disappeared a long time ago, so the (legal) customer was afraid of physical dongle damage.

While running without a dongle connected, a message box with the text “Invalid Security Device” appeared.

Luckily, this text string could easily be found in the executable binary file.

Let's pretend we are not very familiar both with Mac OS Classic and PowerPC, but will try anyway.

IDA opened the executable file smoothly, reported its type as “PEF (Mac OS or Be OS executable)” (indeed, it is a standard Mac OS Classic file format).

By searching for the text string with the error message, we've got into this code fragment:

\(^{12}\)pre-UNIX MacOS
Yes, this is PowerPC code.

The CPU is a very typical 32-bit RISC of 1990s era.

Each instruction occupies 4 bytes (just as in MIPS and ARM) and the names somewhat resemble MIPS instruction names.

check1() is a function name we'll give to it later. BL is Branch Link instruction, e.g., intended for calling subroutines.

The crucial point is the BNE instruction which jumps if the dongle protection check passes or not if an error occurs: then the address of the text string gets loaded into the r3 register for the subsequent passing into a message box routine.

From the [Steve Zucker, SunSoft and Kari Karhi, IBM, SYSTEM V APPLICATION BINARY INTERFACE: PowerPC Processor Supplement, (1995)] we will found out that the r3 register is used for return values (and r4, in case of 64-bit values).

Another yet unknown instruction is CLRLWI. From [PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, (2000)] we'll learn that this instruction does both clearing and loading. In our case, it clears the 24 high bits from the value in r3 and puts them in r0, so it is analogical to MOVZX in x86 ([1.23.1 on page 201]), but it also sets the flags, so BNE can check them afterwards.

Let's take a look into the check1() function:

As you can see in IDA, that function is called from many places in the program, but only the r3 register’s value is checked after each call.

All this function does is to call the other function, so it is a thunk function: there are function prologue and epilogue, but the r3 register is not touched, so check1() returns what check2() returns.

BLR looks like the return from the function, but since IDA does the function layout, we probably do not need to care about this.

Since it is a typical RISC, it seems that subroutines are called using a link register, just like in ARM.

The check2() function is more complex:

---

13Also available as http://yurichev.com/mirrors/PowerPC/elfspec_ppc.pdf
14Also available as http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf
15(PowerPC) Branch to Link Register
We are lucky again: some function names are left in the executable (debug symbols section?)
We know that a “good” r3 value has to be non-zero, zero r3 leads the execution flow to the message box.

What we see next is, however, easier to understand: with an error message.

There are two li %r3, 1 instructions present in the function and one li %r3, 0 (Load Immediate, i.e., loading a value into a register). The first instruction is at 0x001186B0—and frankly speaking, it’s hard to say what it means.

What we see next is, however, easier to understand: .RBEFINDFIRST() is called: if it fails, 0 is written into r3 and we jump to exit, otherwise another function is called (check3())—if it fails too, .RBEFINDNEXT() is called, probably in order to look for another USB device.

N.B.: clrlwi. %r0, %r3, 16 is analogical to what we already saw, but it clears 16 bits, i.e., .RBEFINDFIRST() probably returns a 16-bit value.

B (stands for branch) unconditional jump.

BEQ is the inverse instruction of BNE.

Let’s see check3():

```
seg000:0011873C  check3:   # CODE XREF: check2+88p
seg000:0011873C    set var_18, -0x18
seg000:0011873C    set var_C, -0xC
seg000:0011873C    set var_8, -8
seg000:0011873C    set var_4, -4
seg000:0011873C    set arg_8, 8
seg000:0011873C    li %r3, 1
seg000:0011873C 93 E1 FF FC    stw  %r31, var_4(%sp)
seg000:00118740 7C 08 02 A6    mtlr  %r0
seg000:00118744 38 A0 00 00    li  %r5, 0
seg000:00118748 93 C1 FF F8    stw  %r30, var_8(%sp)
seg000:0011874C 83 C2 95 A8    lwz  %r30, off_1485E8 # dword_24B704
seg000:00118750    using dword_24B704, %r30
seg000:00118759 93 A1 FF F4    stw  %r29, var_C(%sp)
seg000:0011875D 38 A3 00 00    addi  %r29, %r3, 0
seg000:0011875F 38 60 00 00    li  %r3, 0
seg000:0011877C 90 01 00 00    stw  %r0, arg_8(%sp)
seg000:0011877E 94 21 FF B0    stwu  %sp, -0x50(%sp)
seg000:00118780 80 DE 00 00    lwz  %r6, dword_24B704
seg000:00118784 38 81 00 38    addi  %r4, %sp, 0x50+var_18
seg000:0011878C 48 00 C0 5D    bl  .RBEREAD
seg000:0011877D 60 00 00 00    nop
seg000:0011877E 54 60 04 3F    clrlwi. %r0, %r3, 16
seg000:0011877F 41 82 00 0C    beq  loc_118784
seg000:0011877F 38 60 00 00    li  %r3, 0
seg000:00118780 48 00 02 F0    b  exit
seg000:00118784   loc_118784:   # CODE XREF: check3+3Cj
seg000:00118784  A0 01 00 38    lhz  %r0, 0x50+var_18(%sp)
seg000:00118788 28 00 04 B2    cmplwi  %r0, 0x1100
seg000:0011878C 41 82 00 0C    beq  loc_118798
seg000:00118790 38 60 00 00    li  %r3, 0
seg000:00118794 48 00 02 DC    b  exit
seg000:00118798   loc_118798:   # CODE XREF: check3+50j
seg000:00118798  80 DE 00 00    lwz  %r6, dword_24B704
seg000:0011879C 38 81 00 38    addi  %r4, %sp, 0x50+var_18
seg000:001187A0 38 60 00 01    li  %r3, 1
seg000:001187A4 38 A0 00 00    li  %r5, 0
seg000:001187AB 48 00 C0 21    bl  .RBEREAD
seg000:001187AC 60 00 00 00    nop
```

(6.5.2), like .RBEFINDNEXT() and .RBEFINDFIRST().

Eventually these functions call other functions with names like .GetNextDeviceViaUSB(), .USBSendPKT(), so these are clearly dealing with an USB device.

There is even a function named .GetNextEve3Device()—sounds familiar, there was a Sentinel Eve3 dongle for ADB port (present on Macs) in 1990s.

Let’s first take a look on how the r3 register is set before return, while ignoring everything else.

We know that a “good” r3 value has to be non-zero, zero r3 leads the execution flow to the message box with an error message.

Let’s see on how the r3 register is set before return, while ignoring everything else.
seg000:001187B0 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:001187B4 41 82 00 0C  beq loc_1187C0
seg000:001187B8 38 60 00 00  li %r3, 0
seg000:001187BC 48 00 02 B4  b exit
seg000:001187C0
seg000:001187C0  loc_1187C0:  # CODE XREF: check3+78j
seg000:001187C0 A0 01 00 38  lhz %r0, 0x50+var_18(%sp)
seg000:001187C4 28 00 06 4B  cmplwi %r0, 0x09AB
seg000:001187C8 41 82 00 0C  beq loc_1187D4
seg000:001187CC 38 60 00 00  li %r3, 0
seg000:001187D0 48 00 02 A0  b exit
seg000:001187D4
seg000:001187D4  loc_1187D4:  # CODE XREF: check3+8Cj
seg000:001187D4 4B F9 F3 D9  bl sub_B78AC
seg000:001187D8 60 00 00 00  nop
seg000:001187DC 54 60 06 3E  clrlwi %r0, %r3, 24
seg000:001187E0 2C 00 00 05  cmpwi %r0, 5
seg000:001187E4 41 82 01 00  beq loc_1188E4
seg000:001187E8 40 80 00 10  bge loc_1187F8
seg000:001187EC 2C 00 00 04  cmpwi %r0, 4
seg000:001187F0 40 80 00 58  bge loc_118848
seg000:001187F4 48 00 01 8C  b loc_118980
seg000:001187F8
seg000:001187F8  loc_1187F8:  # CODE XREF: check3+ACj
seg000:001187F8 2C 00 00 08  cmpwi %r0, 0xB
seg000:001187FC 41 82 00 08  beq loc_118804
seg000:00118800 48 00 01 80  b loc_118980
seg000:00118804
seg000:00118804  loc_118804:  # CODE XREF: check3+C0j
seg000:00118804 80 DE 00 00  lwz %r6, dword_248704
seg000:00118808 38 81 00 38  addi %r4, %sp, 0x50+var_18
seg000:0011880C 38 60 00 00  li %r3, 8
seg000:00118810 38 A0 00 00  li %r5, 0
seg000:00118814 48 00 BF B5  bl .RBREAD
seg000:00118818 60 00 00 00  nop
seg000:0011881C 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:00118820 41 82 00 0C  beq loc_11882C
seg000:00118824 38 60 00 00  li %r3, 0
seg000:00118828 48 00 02 48  b exit
seg000:0011882C
seg000:0011882C  loc_11882C:  # CODE XREF: check3+E4j
seg000:0011882C A0 01 00 38  lhz %r0, 0x50+var_18(%sp)
seg000:00118830 28 00 11 30  cmplwi %r0, 0xFEA0
seg000:00118834 41 82 00 0C  beq loc_118840
seg000:00118838 38 60 00 00  li %r3, 0
seg000:0011883C 48 00 02 34  b exit
seg000:00118840
seg000:00118840  loc_118840:  # CODE XREF: check3+F8j
seg000:00118840 38 60 00 01  li %r3, 1
seg000:00118844 48 00 02 2C  b exit
seg000:00118848
seg000:00118848  loc_118848:  # CODE XREF: check3+B4j
seg000:00118848 80 DE 00 00  lwz %r6, dword_248704
seg000:0011884C 38 81 00 38  addi %r4, %sp, 0x50+var_18
seg000:00118850 38 60 00 00  li %r3, 0xA
seg000:00118854 38 A0 00 00  li %r5, 0
seg000:00118858 48 00 BF 71  bl .RBREAD
seg000:0011885C 60 00 00 00  nop
seg000:00118860 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000:00118864 41 82 00 0C  beq loc_118870
seg000:00118868 38 60 00 00  li %r3, 0
seg000:0011886C 48 00 02 04  b exit
seg000:00118870
seg000:00118870  loc_118870:  # CODE XREF: check3+128j
seg000:00118870 A0 01 00 38  lhz %r0, 0x50+var_18(%sp)
seg000:00118874 28 00 03 F3  cmplwi %r0, 0xA6E1
seg000:00118878 41 82 00 0C  beq loc_118884
seg000:0011887C 38 60 00 00  li %r3, 0
seg000:00118880 48 00 01 F0  b exit
seg000:00118884
seg000: 0011895C  loc_11895C:  # CODE XREF: check3+214j
seg000: 0011895C A0 01 00 38  lhz  %r0, 0x50+var_18(%sp)
seg000: 00118960 28 00 07 CF  cmplwi %r0, 0xFC7
seg000: 00118964 41 82 00 0C  beq  loc_118970
seg000: 00118968 38 60 00 00  li  %r3, 0
seg000: 0011896C 48 00 01 04  b  exit
seg000: 00118970
seg000: 00118970  loc_118970:  # CODE XREF: check3+228j
seg000: 00118970 28 1F 00 03  cmplwi %r31, 3
seg000: 00118974 40 82 00 F8  bne  error
seg000: 00118978 38 60 00 01  li  %r3, 1
seg000: 0011897C 48 00 00 F4  b  exit
seg000: 00118980
seg000: 00118980  loc_118980:  # CODE XREF: check3+B8j
seg000: 00118980  # check3+C4j
seg000: 00118980 80 DE 00 00  lwz  %r6, dword_24B704
seg000: 00118984 38 81 00 38  addi  %r4, %sp, 0x50+var_18
seg000: 00118988 38 E0 00 00  li  %r31, 0
seg000: 0011898C 38 60 00 04  li  %r3, 4
seg000: 00118990 38 A0 00 00  li  %r5, 0
seg000: 00118994 40 00 BE 35  bl  .RBEREAD
seg000: 00118998 60 00 00 00  nop
seg000: 0011899C 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000: 001189A0 41 82 00 0C  beq  loc_1189AC
seg000: 001189A4 38 60 00 00  li  %r3, 0
seg000: 001189AA 48 00 00 C8  b  exit
seg000: 001189AC
seg000: 001189AC  loc_1189AC:  # CODE XREF: check3+264j
seg000: 001189AC A0 01 00 38  lhz  %r0, 0x50+var_18(%sp)
seg000: 001189B0 28 00 1D 6A  cmplwi %r0, 0xAED0
seg000: 001189B4 40 82 00 0C  bne  loc_1189C0
seg000: 001189B8 38 E0 00 01  li  %r31, 1
seg000: 001189BC 48 00 00 14  b  loc_1189D0
seg000: 001189C0
seg000: 001189C0  loc_1189C0:  # CODE XREF: check3+278j
seg000: 001189C0 28 00 18 28  cmplwi %r0, 0x2818
seg000: 001189C4 41 82 00 0C  beq  loc_1189D0
seg000: 001189C8 38 60 00 00  li  %r3, 0
seg000: 001189CC 48 00 00 A4  b  exit
seg000: 001189D0
seg000: 001189D0  loc_1189D0:  # CODE XREF: check3+280j
seg000: 001189D0  # check3+288j
seg000: 001189D0 57 A0 06 3E  clrlwi  %r0, %r29, 24
seg000: 001189D4 28 00 00 02  cmplwi %r0, 2
seg000: 001189D8 40 82 00 20  bne  loc_1189F8
seg000: 001189DC 57 E0 06 3F  clrlwi. %r0, %r31, 24
seg000: 001189EO 41 82 00 10  beq  good2
seg000: 001189E4 48 00 4C 69  bl  sub_11D64C
seg000: 001189E8 60 00 00 00  nop
seg000: 001189EC 48 00 00 84  b  exit
seg000: 001189F0
seg000: 001189F0  good2:  # CODE XREF: check3+2A4j
seg000: 001189F0 38 60 00 01  li  %r3, 1
seg000: 001189F4 48 00 00 7C  b  exit
seg000: 001189F8
seg000: 001189F8  loc_1189F8:  # CODE XREF: check3+29Cj
seg000: 001189F8 80 DE 00 00  lwz  %r6, dword_24B704
seg000: 001189FC 38 81 00 38  addi  %r4, %sp, 0x50+var_18
seg000: 00118A00 38 60 00 05  li  %r3, 5
seg000: 00118A04 38 A0 00 00  li  %r5, 0
seg000: 00118A08 48 00 BD C1  bl  .RBEREAD
seg000: 00118A0C 60 00 00 00  nop
seg000: 00118A10 54 60 04 3F  clrlwi. %r0, %r3, 16
seg000: 00118A14 41 82 00 0C  beq  loc_118A20
seg000: 00118A18 38 60 00 00  li  %r3, 0
seg000: 00118A1C 48 00 00 54  b  exit
seg000: 00118A20
seg000: 00118A20  loc_118A20:  # CODE XREF: check3+2D8j
seg000: 00118A20 A0 01 00 38  lhz  %r0, 0x50+var_18(%sp)
seg000: 00118A24 28 00 11 03  cmplwi %r0, 0x300
We also see that the r3 register is also filled before each call to .RBEREAD().

And now it all works without a dongle connected.

The function returns some values from the dongle, so they are compared here with some hard-coded variables using CMPLWI.

We also see that the r3 register is also filled before each call to .RBEREAD() with one of these values: 0, 1, 8, 0xA, 0xB, 0xC, 0xD, 4, 5. Probably a memory address or something like that?

Perhaps we don’t even have to learn any other PowerPC instructions: all this function does is just call .RBEREAD(), compare its results with the constants and returns 1 if the comparisons are fine or 0 otherwise.

OK, all we’ve got is that check1() has always to return 1 or any other non-zero value.

But since we are not very confident in our knowledge of PowerPC instructions, we are going to be careful: we will patch the jumps in check2() at 0x001186FC and 0x00118718.

At 0x001186FC we’ll write bytes 0x48 and 0 thus converting the BEQ instruction in an B (unconditional jump): we can spot its opcode in the code without even referring to [PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, (2000)]16.

At 0x00118718 we’ll write 0x60 and 3 zero bytes, thus converting it to a NOP instruction: Its opcode we could spot in the code too.

And now it all works without a dongle connected.

In summary, such small modifications can be done with IDA and minimal assembly language knowledge.

---

16Also available as http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf
8.8.2 Example #2: SCO OpenServer

An ancient software for SCO OpenServer from 1997 developed by a company that disappeared a long time ago.

There is a special dongle driver to be installed in the system, that contains the following text strings: “Copyright 1989, Rainbow Technologies, Inc., Irvine, CA” and “Sentinel Integrated Driver Ver. 3.0”.

After the installation of the driver in SCO OpenServer, these device files appear in the /dev filesystem:

/dev/rbsl8
/dev/rbsl9
/dev/rbsl10

The program reports an error without dongle connected, but the error string cannot be found in the executables.

Thanks to IDA, it is easy to load the COFF executable used in SCO OpenServer.

Let’s also try to find “rbsl” string and indeed, found it in this code fragment:

```
.text:0002AB8     public SSQC
.text:0002AB8     SSQC proc near ; CODE XREF: SSQ+7p
.text:0002AB8     var_44 = byte ptr -44h
.text:0002AB8     var_29 = byte ptr -29h
.text:0002AB8     arg_0 = dword ptr 8
.text:0002AB8     push ebp
.text:0002AB8     mov ebp, esp
.text:0002AB8     sub esp, 44h
.text:0002ABE     push edi
.text:0002AC4     push esi
.text:0002AC5     mov esi, [ebp+arg_0]
.text:0002AC8     push ebx
.text:0002AC9     push esi
.text:0002ACA     call strlen
.text:0002ACF     add esp, 4
.text:0002ADD     jnz loc_22B6B
.text:0002ADE     mov al, [esi-1]
.text:0002AA1     movsx eax, al
.text:0002AA4     cmp eax, '3'
.text:0002A99     jz loc_22B84
.text:0002A9F     cmp eax, '4'
.text:0002A94     jz loc_22B94
.text:0002FAA     cmp eax, '5'
.text:0002AFF     jnz short loc_22B68
.text:0002B01     movsx ebx, byte ptr [esi]
.text:0002B04     sub ebx, '0'
.text:0002B07     mov eax, 7
.text:0002B0C     add eax, ebx
.text:0002B0E     push eax
.text:0002B0F     lea eax, [ebp+var_44]
.text:0002B12     push offset aDevSlD ; "/dev/sl%d"
.text:0002B17     push eax
.text:0002B18     call nl_printf
.text:0002B1D     push 0
.text:0002B1F     push offset aDevRbsl8 ; char *
.text:0002B24     call _access
.text:0002B29     add esp, 14h
.text:0002B2C     cmp eax, 0FFFFFFFFh
.text:0002B31     jz short loc_22B48
.text:0002B33     lea eax, [ebx+7]
.text:0002B36     push eax
.text:0002B37     lea eax, [ebp+var_44]
.text:0002B3A     push offset aDevRbslD ; "/dev/rbsl%d"
.text:0002B3F     push eax
.text:0002B40     call nl_printf
.text:0002B45     add esp, 0Ch
```
.text:00022B48  loc_22B48:  ; CODE XREF: SSQC+79j
.text:00022B48  mov  edx, [edi]
.text:00022B4A  test  edx, edx
.text:00022B4C  jle  short loc_22B57
.text:00022B4E  push  edx  ; int
.text:00022B4F  call  _close
.text:00022B54  add  esp, 4
.text:00022B57  loc_22B57:  ; CODE XREF: SSQC+94j
.text:00022B57  push  2  ; int
.text:00022B59  lea  eax, [ebp+var_44]
.text:00022B5C  push  eax  ; char *
.text:00022B5D  call  _open
.text:00022B62  add  esp, 8
.text:00022B65  test  eax, eax
.text:00022B67  mov  [edi], eax
.text:00022B69  jge  short loc_22B78
.text:00022B6B  loc_22B6B:  ; CODE XREF: SSQC+47j
.text:00022B6B  mov  eax, 0FFFFFFFFh
.text:00022B70  pop  ebx
.text:00022B71  pop  esi
.text:00022B72  pop  edi
.text:00022B73  mov  esp, ebp
.text:00022B75  pop  ebp
.text:00022B76  retn
.text:00022B78  loc_22B8:  ; CODE XREF: SSQC+81j
.text:00022B78  pop  ebx
.text:00022B79  pop  esi
.text:00022B7A  pop  edi
.text:00022B7B  xor  eax, eax
.text:00022B7D  mov  esp, ebp
.text:00022B7F  pop  ebp
.text:00022B80  retn
.text:00022B84  loc_22B84:  ; CODE XREF: SSQC+31j
.text:00022B84  mov  al, [esi]
.text:00022B86  pop  ebx
.text:00022B87  pop  esi
.text:00022B88  pop  edi
.text:00022B89  mov  ds:byte_407224, al
.text:00022B8E  mov  esp, ebp
.text:00022B90  xor  eax, eax
.text:00022B92  pop  ebp
.text:00022B93  retn
.text:00022B94  loc_22B94:  ; CODE XREF: SSQC+3Cj
.text:00022B94  mov  al, [esi]
.text:00022B96  pop  ebx
.text:00022B97  pop  esi
.text:00022B98  pop  edi
.text:00022B99  mov  ds:byte_407225, al
.text:00022B9E  mov  esp, ebp
.text:00022B9F  xor  eax, eax
.text:00022BA0  pop  ebp
.text:00022BA2  retn
.text:00022BA3  loc_22BA4:  ; CODE XREF: SSQC+1Fj
.text:00022BA4  movsx  eax, ds:byte_407225
.text:00022B4A  push  esi
.text:00022B4C  push  eax
.text:00022B4D  movsx  eax, ds:byte_407224
.text:00022B4E  push  eax
.text:00022B50  lea  eax, [ebp+var_44]
.text:00022B55  push  offset a46CCS  ; "46%c%c%s"
.text:00022B5B  push  eax
.text:00022B66  call  nl_sprintf
.text:00022B6C  lea  eax, [ebp+var_44]
Yes, indeed, the program needs to communicate with the driver somehow.

The only place where the SSQC() function is called is the thunk function:

SSQC() can be called from at least 2 functions.

One of these is:

; these names we gave to the labels:
.data:004016B8 _3C_or_3E dd offset a3c ; DATA XREF: sys_info:loc_D67Br
.data:004016B8 _3C_and_B db 0Ch ; DATA XREF: sys_info+BAr

xor  eax, eax
mov   al, ds:ctl_port
mov   ecx, _51_52_53[eax*4]
push  ecx
call  SSQ
add   esp, 4
cmp   eax, 0FFFFFFFFh
ejz   loc_D6D1
xor   ebx, ebx
mov   al, _C_and_B
test  al, al
jz    short loc_D6C0

.loc_D6B: ; CODE XREF: sys_info+106j
mov   eax, _3C_or_3E[ebx*4]
push  eax
call  SSQ
push  offset a4g ; "4G"
call  SSQ
push  offset a0123456789 ; "0123456789"
push  esp, 0Ch
mov   edx, answers1[ebx*4]
cmp   eax, edx
jz    short OK
mov   ecx, answers2[ebx*4]
cmp   eax, ecx
jz    short OK
mov   al, byte_4016D1[ebx]
inc   ebx
test  al, al
jnz   short loc_D6B

.loc_D6C0: ; CODE XREF: sys_info+C1j
inc   ds:ctl_port
xor   eax, eax
mov   al, ds:ctl_port
cmp   eax, edi
jle   short loc_D652

.loc_D6D1: ; CODE XREF: sys_info+98j
mov   edx, [ebp+var_8]
inc   edx
mov   [ebp+var_8], edx
cmp   edx, 3
jle   loc_D641

.loc_D6E1: ; CODE XREF: sys_info+16j
pop   ebx
pop   edi
mov   esp, ebp
pop   ebp
ret

OK:    ; CODE XREF: sys_info+F0j
mov   al, _C_and_B[ebx]
pop   ebx
pop   edi
mov   ds:ctl_model, al
mov   esp, ebp
pop   ebp
ret
“3C” and “3E” sound familiar: there was a Sentinel Pro dongle by Rainbow with no memory, providing only one crypto-hashing secret function.

You can read a short description of what hash function is here: 2.11 on page 462.

But let’s get back to the program.

So the program can only check the presence or absence of a connected dongle.

No other information can be written to such dongle, as it has no memory. The two-character codes are commands (we can see how the commands are handled in the SSQC() function) and all other strings are hashed inside the dongle, being transformed into a 16-bit number. The algorithm was secret, so it was not possible to write a driver replacement or to remake the dongle hardware that would emulate it perfectly.

However, it is always possible to intercept all accesses to it and to find what constants the hash function results are compared to.

But we need to say that it is possible to build a robust software copy protection scheme based on secret cryptographic hash-function: let it encrypt/decrypt the data files your software uses.

But let’s get back to the code.

Codes 51/52/53 are used for LPT printer port selection. 3x/4x are used for “family” selection (that’s how Sentinel Pro dongles are differentiated from each other: more than one dongle can be connected to a LPT port).

The only non-2-character string passed to the hashing function is “0123456789”.

Then, the result is compared against the set of valid results.

If it is correct, 0xC or 0xB is to be written into the global variable ctl_model.

Another text string that gets passed is “PRESS ANY KEY TO CONTINUE: ”, but the result is not checked.

Hard to say why, probably by mistake.

Let’s see where the value from the global variable ctl_model is used.

One such place is:

```
.text:0000D708 prep_sys  proc near ; CODE XREF: init_sys+46Ap
.text:0000D708 .text:0000D708 var_14  = dword ptr -14h
.text:0000D708 .text:0000D708 var_10  = byte ptr -10h
.text:0000D708 .text:0000D708 var_8   = dword ptr -8
.text:0000D708 .text:0000D708 var_2   = word ptr -2
.text:0000D708 push   ebp
.text:0000D708 mov    eax, ds:net_env
.text:0000D709 mov    ebp, esp
.text:0000D710 sub    esp, 1Ch
.text:0000D711 test   eax, eax
.text:0000D715 jnz   short loc_D734
.text:0000D717 mov    al, ds:ctl_model
.text:0000D71C test   al, al
.text:0000D71E jnz   short loc_D77E
.text:0000D720 mov    [ebp+var_8], offset aIeCvulnvV0KgT_ ; "Ie-cvulnvV\bOKG\T_"
.text:0000D727 mov    edx, 7
.text:0000D72C jmp   loc_D7E7

... 
```

```
.text:0000D7E7 loc_D7E7: ; CODE XREF: prep_sys+24j
.text:0000D7E7 ; prep_sys+33j
.text:0000D7E7 push   edx
.text:0000D7E8 mov    edx, [ebp+var_8]
.text:0000D7EB push   20h
.text:0000D7ED push   edx
.text:0000D7EE push   16h
.text:0000D7F0 call   err_warn
.text:0000D7F5 push   offset station_sem

17What a strange feeling: to find bugs in such ancient software.
```
If it is 0, an encrypted error message is passed to a decryption routine and printed.

The error string decryption routine seems a simple xoring:

```
.text:0000A43C proc near ; CODE XREF: prep_sys+E8p
.text:0000A43C proc near ; prep_sys2+2Fp ...
.text:0000A43C var_55 = byte ptr -55h
.text:0000A43C var_54 = byte ptr -54h
.text:0000A43C arg_0 = dword ptr 8
.text:0000A43C arg_4 = dword ptr 0Ch
.text:0000A43C arg_8 = dword ptr 10h
.text:0000A43C arg_C = dword ptr 14h
.text:0000A43C push ebp
.text:0000A43D mov ebp, esp
.text:0000A43F sub esp, 54h
.text:0000A442 push edi
.text:0000A443 mov ecx, [ebp+arg_8]
.text:0000A446 xor edi, edi
.text:0000A448 test ecx, ecx
.text:0000A44A push esi
.text:0000A44B jle short loc_A466
.text:0000A44D mov esi, [ebp+arg_C] ; key
.text:0000A450 mov edx, [ebp+arg_4] ; string
.text:0000A453 loc_A453:
.text:0000A453 xor eax, eax
.text:0000A455 mov al, [edx+edi]
.text:0000A458 xor eax, esi
.text:0000A45A add esi, 3
.text:0000A45D inc edi
.text:0000A45E cmp edi, ecx
.text:0000A460 mov [ebp+edi+var_55], al
.text:0000A464 jl short loc_A453
.text:0000A466 loc_A466:
.text:0000A466 mov [ebp+edi+var_54], 0
.text:0000A46B mov eax, [ebp+arg_0]
.text:0000A46E cmp eax, 18h
.text:0000A473 jnz short loc_A49C
.text:0000A475 lea eax, [ebp+var_54]
.text:0000A478 push eax
.text:0000A479 call status_line
.text:0000A47E add esp, 4
.text:0000A481 loc_A481:
.text:0000A481 push 50h
.text:0000A483 push 0
.text:0000A485 lea eax, [ebp+var_54]
.text:0000A488 push eax
.text:0000A489 call memset
.text:0000A48E call pcv_refresh
.text:0000A493 add esp, 0Ch
.text:0000A496 pop esi
.text:0000A497 pop edi
.text:0000A498 mov esp, ebp
.text:0000A499 pop ebp
.text:0000A49B retn
.text:0000A49C loc_A49C:
.text:0000A49C push 0 ; CODE XREF: err_warn+3Ef
.text:0000A49E lea eax, [ebp+var_54]
.text:0000A4A1 mov edx, [ebp+arg_0]
.text:0000A4A4 push edx
.text:0000A4A5 push eax
.text:0000A4A6 call pcv_lputs
```
That's why we were unable to find the error messages in the executable files, because they are encrypted (which is in popular practice).

Another call to the SSQ() hashing function passes the "offln" string to it and compares the result with 0xFE81 and 0x12A9.

If they don't match, it works with some timer() function (maybe waiting for a poorly connected dongle to be reconnected and check again?) and then decrypts another error message to dump.

...
As we can see, not just string is supplied to the decryption function, but also the key:

Another option is to write our own SCO OpenServer driver, containing a table of questions and answers, all of those which present in the program.

Decrypting error messages

By the way, we can also try to decrypt all error messages. The algorithm that is located in the `err_warn()` function is very simple, indeed:

Listing 8.5: Decryption function

```
.text:0000A44D  mov   esi, [ebp+arg_C]; key
.text:0000A450  mov   edx, [ebp+arg_4]; string
.text:0000A453  loc_A453:
.text:0000A453  xor   eax, eax
.text:0000A455  mov   al, [edx+edi]; load encrypted byte
.text:0000A458  xor   eax, esi; decrypt it
.text:0000A45A  add   esi, 3; change key for the next byte
.text:0000A45D  inc   edi
.text:0000A45E  cmp   edi, ecx
.text:0000A460  mov   [ebp+edi+var_55], al
.text:0000A464  jl    short loc_A453
```

As we can see, not just string is supplied to the decryption function, but also the key:

```
.text:0000DAF7  error:                   ; CODE XREF: sync_sys+255j
.text:0000DAF7  ; sync_sys+274j ...
.text:0000DAF7  mov   [ebp+var_8], offset encrypted_error_message2
.text:0000DAFE  mov   [ebp+var_C], 17h; decrypting key
.text:0000DB05  jmp   decrypt_end_print_message

; this name we gave to label manually:
.text:0000D9B6  decrypt_end_print_message:               ; CODE XREF: sync_sys+29Dj
.text:0000D9B6  ; sync_sys+2ABj
.text:0000D9B6  mov   eax, [ebp+var_18]
.text:0000D9B9  test   eax, eax
.text:0000D9BB  jnz   short loc_D9FB
.text:0000D9BD  mov   edx, [ebp+var_C]; key
.text:0000D9C0  mov   ecx, [ebp+var_8]; string
.text:0000D9C3  push  edx
.text:0000D9C4  push  20h
.text:0000D9C6  push  ecx
.text:0000D9C7  push  18h
.text:0000D9C9  call  err_warn
```
The algorithm is a simple xor: each byte is xored with a key, but the key is increased by 3 after the processing of each byte.

We can write a simple Python script to check our hypothesis:

```
Listing 8.6: Python 3.x
#!/usr/bin/python
import sys
msg=[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x00, 0x0B, 0x1F, 0x7, 0x1E, 0x1A]
key=0x17
tmp=key
for i in msg:
    sys.stdout.write ('%c' % (i^tmp))
tmp=tmp+3
sys.stdout.flush()
```

And it prints: “check security device connection”. So yes, this is the decrypted message.

There are also other encrypted messages with their corresponding keys. But needless to say, it is possible to decrypt them without their keys. First, we can see that the key is in fact a byte. It is because the core decryption instruction (XOR) works on byte level. The key is located in the ESI register, but only one byte part of ESI is used. Hence, a key may be greater than 255, but its value is always to be rounded.

As a consequence, we can just try brute-force, trying all possible keys in the 0..255 range. We are also going to skip the messages that has unprintable characters.

```
Listing 8.7: Python 3.x
#!/usr/bin/python
import sys, curses.ascii
msgs=
    [0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76,
0x3A, 0x33, 0x31, 0x0C, 0x00, 0x0B, 0x1F, 0x7, 0x1E, 0x1A],
[0x49, 0x65, 0x2D, 0x63, 0x75, 0x6C, 0x6E, 0x76, 0x56, 0x5C,
8, 0x4F, 0x4B, 0x47, 0x5D, 0x54, 0x5F, 0x1D, 0x26, 0x2C, 0x33,
0x27, 0x28, 0x66, 0x72, 0x75, 0x78, 0x77, 0x7E, 0x41, 0x44],
[0x45, 0x61, 0x31, 0x67, 0x72, 0x79, 0x68, 0x52, 0x4A, 0x52, 0x50,
0x0C, 0x4B, 0x57, 0x43, 0x51, 0x58, 0x5B, 0x61, 0x37, 0x33, 0x2B,
0x39, 0x39, 0x33, 0x38, 0x79, 0x3A, 0x30, 0x17, 0x0B, 0x0C],
[0x40, 0x64, 0x79, 0x75, 0x7F, 0x6F, 0x8, 0x4C, 0x40, 0x9, 0x4D, 0x5A,
0x46, 0x5D, 0x57, 0x49, 0x57, 0x3B, 0x21, 0x23, 0x6A, 0x38, 0x23,
0x36, 0x24, 0x2A, 0x7C, 0x3A, 0x1A, 0x6, 0x0D, 0x0E, 0x0A, 0x14,
0x10],
[0x72, 0x7C, 0x72, 0x79, 0x76, 0x0,
0x50, 0x43, 0x4A, 0x59, 0x5D, 0x5B, 0x41, 0x41, 0x1B, 0x5A,
0x24, 0x32, 0x2E, 0x29, 0x28, 0x70, 0x20, 0x22, 0x38, 0x28, 0x36,
0x0D, 0x0B, 0x48, 0x4B, 0x4E]]
def is_string_printable(s):
    return all(list(map(lambda x: curses.ascii.isprint(x), s)))

cnt=1
for msg in msgs:
    print ("message #%d" % cnt)
    for key in range(0,256):
        result=[]
        tmp=key
        for i in msg:
            result.append (i^tmp)
            tmp=tmp+3
        if is_string_printable(result):
            print ("".join(map(lambda x: chr(x), result)))
            break
    cnt=cnt+1
```
if is_string_printable (result):
    print ("key=", key, "value=", ".join(list(map(chr, result))))

cnt=cnt+1

And we get:

Listing 8.8: Results

message #1
key= 20 value= `eb^h%|`hudw| af{n-f%ljmsbnwlpk
key= 21 value= ajc)i``cauwgv{bgto}g"millcmvkh
key= 22 value= bkd\j#rbbvsfuiz|cduh|d#bhomdlujni
key= 23 value= check security device connection
key= 24 value= lifbl!pd|tqhx#ejwjjb`nQofbshlo

message #2
key= 7 value= No security device found
key= 8 value= An#rbbvsVuz|cduhld#ghtme?!#!'!#

message #3
key= 7 value= Bk<waoqNUpu$`yreoa\wpmpusj,bkJh
key= 8 value= Mj?vfnrOjqv%gxqd`_ywlstlk/cIHi
key= 9 value= Lm>ugasLkvw&fgpgag^uvcrwml.'mwhj
key= 10 value= Ol!td`Mhwx`efwfb!tubuvnmlanvok
key= 11 value= No security device station found
key= 12 value= In#rjbsnuz!{dudhd#r{`whho#gPtme

message #4
key= 14 value= Number of authorized users exceeded
key= 15 value= Ovlmdq!hg#`juknuhydk!vrbsp!Zy`dbefe

message #5
key= 17 value= check security device station
key= 18 value= `ijbh!td`tmhwx`efwfb!tubuVnm!'

There is some garbage, but we can quickly find the English-language messages!

By the way, since the algorithm is a simple xoring encryption, the very same function can be used to encrypt messages. If needed, we can encrypt our own messages, and patch the program by inserting them.

8.8.3 Example #3: MS-DOS

Another very old software for MS-DOS from 1995 also developed by a company that disappeared a long time ago.

In the pre-DOS extenders era, all the software for MS-DOS mostly relied on 16-bit 8086 or 80286 CPUs, so the code was 16-bit en masse.

The 16-bit code is mostly same as you already saw in this book, but all registers are 16-bit and there are less instructions available.

The MS-DOS environment has no system drivers, and any program can deal with the bare hardware via ports, so here you can see the OUT/IN instructions, which are present in mostly in drivers in our times (it is impossible to access ports directly in user mode on all modern OSes).

Given that, the MS-DOS program which works with a dongle has to access the LPT printer port directly.

So we can just search for such instructions. And yes, here they are:

seg030:0034 out_port proc far ; CODE XREF: sent_pro+22p
seg030:0034 ; sent_pro+2Ap ...
seg030:0034 arg_0 = byte ptr 6
seg030:0034 seg030:0034 55
seg030:0034 push bp
seg030:0034 seg030:0035 8B EC
seg030:0035 mov bp, sp
seg030:0037 8B 16 7E E7
seg030:0038 mov dx, _out_port ; 0x378
seg030:0039 8A 46 06
seg030:003A mov al, [bp+arg_0]
seg030:003B seg030:003E EE
seg030:003C out dx, al
seg030:003D seg030:003F 5D
seg030:003E pop bp
seg030:0040 CB seg030:0040 out_port endp
out_port() is referenced only in one function:

```asm
seg030:0041  sent_pro proc far ; CODE XREF: check_dongle+34p
seg030:0041  var_3 = byte ptr -3
seg030:0041  var_2 = word ptr -2
seg030:0041  arg_0 = dword ptr 6
seg030:0041  C8 04 00 00 enter 4, 0
seg030:0045  56 push si
seg030:0046  57 push di
seg030:0047  8B 16 82 E7 mov dx, _in_port_1 ; 0x37A
seg030:0048  EC in al, dx
seg030:004C  8A D8 mov bl, al
seg030:004E  80 E3 FE and bl, 0FEh
seg030:0051  80 CB 04 or bl, 4
seg030:0054  8A C3 mov al, bl
seg030:0056  8B 46 FD mov [bp+var_3], al
seg030:0059  80 E3 1F and bl, 1Fh
seg030:005C  8A C3 mov al, bl
seg030:005E  EE out dx, al
seg030:005F  68 FF 00 push 0FFh
seg030:0062  0E push cs
seg030:0063  E8 CE FF call near ptr out_port
seg030:0066  59 pop cx
seg030:0067  68 D3 00 push 003h
seg030:006A  0E push cs
seg030:006B  E8 C6 FF call near ptr out_port
seg030:006E  59 pop cx
seg030:006F  33 F6 xor si, si
seg030:0071  EB 01 jmp short loc_359D4
seg030:0073  loc_359D3: ; CODE XREF: sent_pro+37j
seg030:0073  46 inc si
seg030:0074  loc_359D4: ; CODE XREF: sent_pro+30j
seg030:0074  81 FE 96 00 cmp si, 96h
seg030:0078  7C F9 jl short loc_359D3
seg030:007A  68 C3 00 push 0C3h
seg030:007D  0E push cs
seg030:007E  E8 B3 FF call near ptr out_port
seg030:0081  59 pop cx
seg030:0082  68 C7 00 push 0C7h
seg030:0085  0E push cs
seg030:0086  E8 AB FF call near ptr out_port
seg030:0089  59 pop cx
seg030:008A  68 D3 00 push 003h
seg030:008D  0E push cs
seg030:008E  E8 A3 FF call near ptr out_port
seg030:0091  59 pop cx
seg030:0092  68 C3 00 push 0C3h
seg030:0095  0E push cs
seg030:0096  E8 9B FF call near ptr out_port
seg030:0099  59 pop cx
seg030:009A  68 C7 00 push 0C7h
seg030:009D  0E push cs
seg030:009E  E8 93 FF call near ptr out_port
seg030:00A1  59 pop cx
seg030:00A2  68 D3 00 push 003h
seg030:00A5  0E push cs
seg030:00A6  E8 BB FF call near ptr out_port
seg030:00A9  59 pop cx
seg030:00AA  BF FF FF mov di, 0FFFFh
seg030:00AD  EB 40 jmp short loc_35A4F
seg030:00AF  loc_35A0F: ; CODE XREF: sent_pro+BDj
seg030:00AF BE 04 00 mov si, 4
seg030:00B2  loc_35A12: ; CODE XREF: sent_pro+ACj
```

836
This is again a Sentinel Pro “hashing” dongle as in the previous example. It is noticeably because text strings are passed here, too, and 16 bit values are returned and compared with others.

So that is how Sentinel Pro is accessed via ports.

The output port address is usually 0x378, i.e., the printer port, where the data to the old printers in pre-USB era was passed to.

The port is uni-directional, because when it was developed, no one imagined that someone will need to transfer information from the printer. The only way to get information from the printer is a status register on port 0x379, which contains such bits as “paper out”, “ack”, “busy”—thus the printer may signal to the host computer if it is ready or not and if paper is present in it.

So the dongle returns information from one of these bits, one bit at each iteration. _in_port_2 contains the address of the status word (0x379) and _in_port_1 contains the control register address (0x37A).

It seems that the dongle returns information via the “busy” flag at seg030:00B9: each bit is stored in the DI register, which is returned at the end of the function.

What do all these bytes sent to output port mean? Hard to say. Perhaps, commands to the dongle. But generally speaking, it is not necessary to know: it is easy to solve our task without that knowledge.

Here is the dongle checking routine:

```
seg030:0134 8A C3    mov    al, bl
seg030:0136 EE       out    dx, al
seg030:0137 8B C7    mov    ax, di
seg030:0139 5F       pop     di
seg030:013A 5E       pop     si
seg030:013B C9       leave
seg030:013C CB       retf
seg030:013C         sent_pro endp
```

```
This is again a Sentinel Pro “hashing” dongle as in the previous example. It is noticeably because text strings are passed here, too, and 16 bit values are returned and compared with others.

So that is how Sentinel Pro is accessed via ports.

The output port address is usually 0x378, i.e., the printer port, where the data to the old printers in pre-USB era was passed to.

The port is uni-directional, because when it was developed, no one imagined that someone will need to transfer information from the printer.

The only way to get information from the printer is a status register on port 0x379, which contains such bits as “paper out”, “ack”, “busy”—thus the printer may signal to the host computer if it is ready or not and if paper is present in it.

So the dongle returns information from one of these bits, one bit at each iteration. _in_port_2 contains the address of the status word (0x379) and _in_port_1 contains the control register address (0x37A).

It seems that the dongle returns information via the “busy” flag at seg030:00B9: each bit is stored in the DI register, which is returned at the end of the function.

What do all these bytes sent to output port mean? Hard to say. Perhaps, commands to the dongle. But generally speaking, it is not necessary to know: it is easy to solve our task without that knowledge.

Here is the dongle checking routine:

```
00000000 struct_0 struct ; (sizeof=0x1B)
00000000 field_0   db 25 dup(?)   ; string(C)
00000019  _A   dw   ?
0000001B struct_0 ends

dseg:3C6 61 63 72 75+ _Q struct_0 '<hello', 01122h>
dseg:3C6 60 00 00+ ; DATA XREF: check_dongle+2Eo

... skipped ...

dseg:3E00 63 6F 66 66+ struct_0 '<coffee', 7EB7h>
dseg:3E1B 64 6F 67 00+ struct_0 '<dog', 0FFADh>
dseg:3E36 63 61 74 00+ struct_0 '<cat', 0FF5Fh>
```

18If we consider Centronics only. The following IEEE 1284 standard allows the transfer of information from the printer.
The text strings seem to be constructed randomly as well, during software development.

Since the routine can be called very frequently, e.g., before the execution of each important software feature, and accessing the dongle is generally slow (because of the slow printer port and also slow MCU in the dongle), they probably added a way to skip some dongle checks, by checking the current time in the biostime() function.

The get_rand() function uses the standard C function:

So the text string is selected randomly, passed into the dongle, and then the result of the hashing is compared with the correct value.

The text strings seem to be constructed randomly as well, during software development.
And this is how the main dongle checking function is called:

```
seg033:087B 9A 45 01 96+ call check_dongle
seg033:0880 0B C0 or ax, ax
seg033:0882 74 62 jz short OK
seg033:0884 83 3E 60 42+ cmp word_620E0, 0
seg033:0889 75 5B jnz short OK
seg033:088B FF 06 60 42 inc word_620E0
seg033:088F 1E push ds
seg033:0890 68 22 44 push offset aTrupcRequiresA
seg033:0893 1E push ds
seg033:0894 68 60 E9 push offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+ call _strcpy
seg033:089C 83 C4 08 add sp, 8
seg033:089F 1E push ds
seg033:08A0 68 42 44 push offset aPleaseContactA ; "Please Contact ..."
seg033:08A3 1E push ds
seg033:08A4 68 60 E9 push offset byte_6C7E0 ; dest
seg033:08A7 9A CD 64 00+ call _strcat
```

Bypassing the dongle is easy, just force the check_dongle() function to always return 0.

For example, by inserting this code at its beginning:

```
mov ax,0
retd
```

The observant reader might recall that the strcpy() C function usually requires two pointers in its arguments, but we see that 4 values are passed:

```
seg033:088F 1E push ds
seg033:0890 68 22 44 push offset aTrupcRequiresA
seg033:0893 1E push ds
seg033:0894 68 60 E9 push offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+ call _strcpy
seg033:089C 83 C4 08 add sp, 8
```

This is related to MS-DOS’ memory model. You can read more about it here: [11.6 on page 986](#).

So as you may see, strcpy() and any other function that take pointer(s) in its arguments work with 16-bit pairs.

Let’s get back to our example. DS is currently set to the data segment located in the executable, that is where the text string is stored.

In the sent_pro() function, each byte of the string is loaded at seg030:00EF: the LES instruction loads the ES:BX pair simultaneously from the passed argument.

The MOV at seg030:00F5 loads the byte from the memory at which the ES:BX pair points.

### 8.9 Encrypted database case #1

(This part has been first appeared in my blog at 26-Aug-2015. Some discussion: [https://news.ycombinator.com/item?id=10128684](#).)

#### 8.9.1 Base64 and entropy

I’ve got the XML file containing some encrypted data. Perhaps, it’s related to some orders and/or customers information.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<Orders>
  <Order>
    <OrderID>1</OrderID>
    <Data>yjmxhXUbhB/5MV45chPsXZWAJwIh1S0aD9lFn3XuJMSxJ3/E+UE3hsnH</Data>
  </Order>
</Orders>
```
The file is available here.

This is clearly base64-encoded data, because all strings consisting of Latin characters, digits, plus (+) and slash (/) symbols. There can be 1 or 2 padding symbols (=), but they are never occurred in the middle of string. Keeping in mind these base64 properties, it's very easy to recognize them.

Let's decode them and calculate entropies (9.2 on page 929) of these blocks in Wolfram Mathematica:

```mathematica
In[1]:= ListOfBase64Strings = Map[First[#[[3]]] &, Cases[Import["encrypted.xml"], XMLElement["Data", _, _], Infinity]];  
In[2]:= BinaryStrings = Map[ImportString[#, {"Base64", "String"}] &, ListOfBase64Strings];  
In[3]:= Entropies = Map[N[Entropy[2, #]] &, BinaryStrings];  
In[4]:= Variance[Entropies]  
Out[4]= 0.0238614
```

Variance is low. This means the entropy values are not very different from each other. This is visible on graph:

```mathematica
In[5]:= ListPlot[Entropies]
```

Most values are between 5.0 and 5.4. This is a sign that the data is compressed and/or encrypted.
To understand variance, let’s calculate entropies of all lines in Conan Doyle’s *The Hound of the Baskervilles* book:

```plaintext
In[1]:= BaskervillesLines = Import["http://www.gutenberg.org/cache/epub/2852/pg2852.txt", "List/\n\r\n"];

In[2]:= EntropiesT = Map[N[Entropy[2, #]] & , BaskervillesLines];

In[3]:= Variance[EntropiesT]
Out[3]= 2.73883

In[4]:= ListPlot[EntropiesT]
```

Most values are gathered around value of 4, but there are also values which are smaller, and they are influenced final variance value.

Perhaps, shortest strings has smaller entropy, let’s take short string from the Conan Doyle’s book:

```plaintext
In[5]:= Entropy[2, "Yes, sir."] // N
Out[5]= 2.9477

In[6]:= Entropy[2, "Yes"] // N
Out[6]= 1.58496

In[7]:= Entropy[2, "No"] // N
```

### 8.9.2 Is data compressed?

OK, so our data is compressed and/or encrypted. Is it compressed? Almost all data compressors put some header at the start, signature, or something like that. As we can see, there are no consistent data at the start of each block. It’s still possible that this is a handmade data compressor, but they are very rare. On the other hand, handmade cryptoalgorithms are much more popular, because it’s very easy to make it
work. Even primitive keyless cryptosystems like \texttt{memfrob}\textsuperscript{19} and ROT13 works fine without errors. It’s a serious challenge to write data compressor from scratch using only fantasy and imagination in a way so it will have no evident bugs. Some programmers implements data compression functions by reading textbooks, but this is also rare. The most popular two ways are: 1) just take open-source library like zlib; 2) copy&paste something from somewhere. Open-source data compressions algorithms usually puts some kind of header, and so do algorithms from sites like http://www.codeproject.com/.

8.9.3 Is data encrypted?

Major data encryption algorithms process data in blocks. DES—8 bytes, AES—16 bytes. If the input buffer is not divided evenly by block size, it’s padded by zeroes (or something else), so encrypted data will be aligned by cryptoalgorithm’s block size. This is not our case.

Using Wolfram Mathematica, I analyzed block’s lengths:

\begin{verbatim}
In[] := Counts[Map[StringLength[#] &, BinaryStrings]]
\end{verbatim}

1858 blocks has size of 42 bytes, 1235 blocks has size of 38 bytes, etc.

I made a graph:

\begin{verbatim}
ListPlot[Counts[Map[StringLength[#] &, BinaryStrings]]]
\end{verbatim}

So, most blocks has size between ~36 and ~48. There is also another thing to notice: all block sizes are even. No single block with odd size.

There are, however, stream ciphers which can operate on byte level or even on bit level.

8.9.4 CryptoPP

The program which can browse this encrypted database is written C# and the .NET code is heavily obfuscated. Nevertheless, there is DLL with x86 code, which, after brief examination, has parts of the CryptoPP

\textsuperscript{19}http://linux.die.net/man/3/memfrob
popular open-source library! (I just spotted “CryptoPP” strings inside.) Now it’s very easy to find all functions inside of DLL because CryptoPP library is open-source.

CryptoPP library has a lot of crypto-functions, including AES (AKA Rijndael). Newer x86 CPUs has AES helper instructions like AESENC, AESDEC and AESKEYGENASSIST. They are not performing encryption/decryption completely, but they do significant amount of job. And newer CryptoPP versions use them. For example, here: \(^1\), \(^2\). To my surprise, during decryption, AESENC gets executed, while AESDEC is not (I just checked with my tracer utility, but any debugger can be used). I checked, if my CPU really supports AES instructions. Some Intel i3 CPUs are not. And if not, CryptoPP library falling back to AES functions implemented in old way. But my CPU supports them. Why AESDEC is still not executed? Why the program use AES encryption in order to decrypt database?

OK, it’s not a problem to find a function which encrypts block. It is called CryptoPP::Rijndael::Enc::ProcessAndXorBlock: src, and it can call another function: Rijndael::Enc::AdvancedProcessBlocks() src, which, in turn, can call two other functions (AESNI_Enc_Block and AESNI_Enc_4_Blocks) which has AESENC instructions.

So, judging by CryptoPP internals, CryptoPP::Rijndael::Enc::ProcessAndXorBlock() encrypts one 16-byte block. Let’s set breakpoint on it and see, what happens during decryption. I use my simple tracer tool again. The software must decrypt first data block now. Oh, by the way, here is the first data block converted from base64 encoding to hexadecimal data, let’s have it at hand:

```
00000000: CA 39 B1 85 75 1B 84 4F F9 31 5E 72 13 EC 5D .9..u....1^r..]
00000010: 95 80 27 02 21 D5 2D 1A 0F D9 45 9F 75 EE 24 C4 '..!.....E.u.$.
00000020: B1 27 7F 84 FE 41 37 86 C9 C0 .'...A7...
```

These are also arguments of the function from CryptoPP source files:

```cpp
size_t Rijndael::Enc::AdvancedProcessBlocks(const byte *inBlocks, const byte *xorBlocks, byte * outBlocks, size_t length, word32 flags);
```

So it has 5 arguments. Possible flags are:

```cpp
eenum {BT_InBlockIsCounter=1, BT_DontIncrementInOutPointers=2, BT_XorInput=4, BT_ReverseDirection=8, BT_AllowParallel=16} FlagsForAdvancedProcessBlocks;
```

OK, run tracer on ProcessAndXorBlock() function:

```
... tracer.exe -l:filename.exe bpf=filename.exe!0x4339a0,args:5,dump_args:0x10
```

Warning: no tracer.cfg file.

```
PID=1984\New process software.exe
no module registered with image base 0x77320000
no module registered with image base 0x76e20000
no module registered with image base 0x77320000
no module registered with image base 0x77220000
Warning: unknown (to us) INT3 breakpoint at ntdll.dll!LdrVerifyImageMatchesChecksum+0x96c (0\x776c103b)
(0) software.exe!0x4339a0() -> 0x0
```

```
Argument 1/5
0038B920: 01 00 00 00 FF FF FF FF-79 C1 69 0B 67 C1 04 7D "........y.i.g..}"
Argument 3/5
0038B978: CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD "............."
(0) software.exe!0x4339a0() -> 0x0
```

```
Argument 3/5 difference:
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N{3....1.99..}
(0) software.exe!0x4339a0(0x38a828, 0x38a838, 0x38bb40, 0x0, 0x8) (called from software.exe!.text+0\x33c0d (0x13e4c0d))
Argument 1/5
0038A828: 95 80 27 02 21 D5 2D 1A 0F D9 45 9F 75 EE 24 C4 '..!.....E.u.$.
Argument 2/5
0038A838: B1 27 7F 84 FE 41 37 86-C9 C0 00 CD CD CD CD CD "..A7........"
Argument 3/5
0038BB40: CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD "............."
(0) software.exe!0x4339a0() -> 0x0
```

\(^20\)https://en.wikipedia.org/wiki/AES_instruction_set
\(^21\)https://github.com/mmoss/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L355
Here we can see inputs to the `ProcessAndXorBlock()` function, and outputs from it.

This is output from the function during first call:

```
C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N{3....1.99..}"
```

Then the `ProcessAndXorBlock()` is called with zero-length block, but with 8 flag (`BT_ReverseDirection`). Second call:

```
C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N{3....1.99..}"
```

Wow, there is some string familiar to us!

Third call:

```
B1 27 7F E4 9F 01 E3 81-CF C6 12 FB B9 7C F1 BC ".'............|.."
```

The first output is very similar to the first 16 bytes of the encrypted buffer.

Output of the first call of `ProcessAndXorBlock()`:

```
C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N{3....1.99..}"
```

First 16 bytes of encrypted buffer:

```
CA 39 B1 85 75 1B 84 1F F9 31 5E 39 72 13 EC 5D ".9..u....1^9r..]"
```

There are too much equal bytes! How come AES encryption result can be very similar to the encrypted buffer while this is not encryption but rather decryption?!

### 8.9.5 Cipher Feedback mode

The answer is **CFB**\(^\text{22}\): in this mode, AES algorithm used not as encryption algorithm, but as a device which generates cryptographically secure random data. The actual encryption is happening using simple XOR operation.

Here is encryption algorithm (images are taken from Wikipedia):

\(^{22}\text{Cipher Feedback}\)
And decryption:

Now let’s see: AES encryption operation generates 16 bytes (or 128 bits) of random data to be used while XOR-ing, who forces us to use all 16 bytes? If at the last iteration we’ve got 1 byte of data, let’s xor 1 byte of data with 1 byte of generated random data? This leads to important property of CFB mode: data can be not padded, data of arbitrary size can be encrypted and decrypted.

Oh, that’s why all encrypted blocks are not padded. And that’s why AESDEC instruction is never called.

Let’s try to decrypt first block manually, using Python. CFB mode also use IV, as a seed for CSPRNG\(^{23}\). In our case, IV is the block which is encrypted at first iteration:

\[
0038B920: 01 00 00 00 FF FF FF FF-79 C1 69 0B 67 C1 04 7D "........y.i.g..}"
\]

Oh, and we also have to recover encryption key. There is AESKEYGENASSIST is DLL, and it is called, and it is used in the `Rijndael::Base::UncheckedSetKey()` function: `src`. It’s easy to find it in IDA and set breakpoint. Let’s see:

```
... tracer.exe -l:filename.exe bpf=filename.exe!0x435c30,args:3,dump_args:0x10
Warning: no tracer.cfg file.
PID=2068|New process software.exe
no module registered with image base 0x77320000
no module registered with image base 0x76e20000
no module registered with image base 0x77320000
23Cryptographically Secure Pseudorandom Number Generator
```

846
So this is the key: CD C5 7E AD 28 5F 6D E1-CE 8F CC 29 B1 21 88 8E.

During manual decryption we've got this:

```
00000000: 00 00 FF FE 46 00 52 00 41 00 4E 00 4B 00 49 00 ....F.R.A.N.K.I.
00000010: 45 00 20 00 4A 00 4F 00 48 00 4E 00 53 00 66 66 E. .J.O.H.N.S.ff
00000020: 66 66 66 9E 61 40 D4 07 06 01 fff.a@....
```

Now this is something readable! And now we can see why there were so many equal bytes at the first decryption iteration: because plaintext has so many zero bytes!

Let's decrypt the second block:

```
00000000: 17 98 D0 84 3A E9 72 4F DB 82 3F AD E9 3E 2A A8 ....:.rO..?..>*.
00000010: 41 00 52 00 52 00 4F 00 4N 00 CD CC CC CC CC CC A.R.R.O.N.......
00000020: 1B 40 D4 07 06 01 .@....
```

Third, fourth and fifth:

```
00000000: 5D 90 59 06 EF F4 96 B4 7C 33 A7 4A BE FF 6B AB ]...Y....|3.J..f.
00000010: 49 00 47 00 47 00 53 00 00 00 00 00 00 C0 65 40 I.G.G.S.......e@
00000020: D4 07 06 03 ....
```

```
00000000: D3 15 34 5D 21 18 7C 6A AA F8 2D FE 38 F9 D7 4E ..4]!.|n...8..N
00000010: 41 00 20 00 44 00 4F 00 48 00 45 00 52 00 54 00 A. .D.O.H.E.R.T.
00000020: 59 00 48 E1 7A 14 AE FF 68 40 D4 07 06 02 Y.H.z...h....
```

```
00000000: 1E 8B 90 0A 17 7B C5 52 31 6C 4E 2F DE 1B 27 19 .....{.R1N...'.
00000010: 41 00 52 00 43 00 55 00 53 00 00 00 00 00 00 00 A.R.C.U.S.......
00000020: 66 40 D4 07 06 03 f@....
```

All blocks decrypted seems correct except of first 16 bytes part.

### 8.9.6 Initializing Vector

What can affect first 16 bytes?

Let's back to CFB decryption algorithm again: 8.9.5 on the previous page.

We can see that IV can affect to first block decryption operation, but not the second, because during the second iteration, ciphertext from the first iteration is used, and in case of decryption, it's the same, no matter what IV has!

So probably, IV is different each time. Using my tracer, I'll take a look at the first input during decryption of the second block of XML file:

```
0038B920: 02 00 00 00 FE FF FF FF-79 C1 69 0B 67 C1 04 7D ".........y.i.g..}"
```

...third:

```
0038B920: 03 00 00 00 FD FF FF FF-79 C1 69 0B 67 C1 04 7D ".........y.i.g..}"
```

It seems, first and fifth byte are changed each time. I finally concluded that the first 32-bit integer is just OrderID from the XML file, and the second 32-bit integer is also OrderID, but negated. All other 8
bytes are same for each operation. Now I have decrypted the whole database: https://beginners.re/current-tree/examples/encrypted_DB1/decrypted.full.txt.

The Python script used for this is: https://beginners.re/current-tree/examples/encrypted_DB1/decrypt_blocks.py.

Perhaps, the author wanted each block encrypted differently, so he/she used OrderID as part of key. It would be also possible to make different AES key instead of IV.

So now we know that IV only affects first block during decryption in CFB mode, this is feature of it. All other blocks can be decrypted without knowledge IV, but using the key.

OK, so why CFB mode? Apparently, because the very first AES example on CryptoPP wiki uses CFB mode: http://www.cryptopp.com/wiki/Advanced_Encryption_Standard#Encrypting_and_Decrypting_Using_AES. Supposedly, developer choose it for simplicity: the example can encrypt/decrypt text strings with arbitrary lengths, without padding.

It is very likely, program’s author(s) just copypasted the example from CryptoPP wiki page. Many programmers do so.

The only difference that IV is chosen randomly in CryptoPP wiki example, while this indeterminism wasn’t allowable to programmers of the software we are dissecting now, so they choose to initialize IV using Order ID.

Now we can proceed to analyzing matter of each byte in the decrypted block.

### 8.9.7 Structure of the buffer

Let’s take first four decrypted blocks:

```
00000000: 0D 00 FF FE 46 00 52 00 41 00 4E 00 4B 00 49 00 ....F.R.A.N.K.I.
00000010: 45 00 20 00 4A 00 4F 00 48 00 53 00 53 00 ff..J.O.H.N.S.
00000020: 66 66 66 6E 61 63 69 67 68 00 52 00 41 00 4E 00 ....f.f.f.N.A.N.
00000030: 53 00 49 00 20 00 42 00 fff.a@....
```

UTF-16 encoded text strings are clearly visible, these are names and surnames. The first byte (or 16-bit word) is seems string length, we can visually check it. FF FE is seems Unicode BOM.

There are 12 more bytes after each string.

Using this script (https://beginners.re/current-tree/examples/encrypted_DB1/dump_buffer_rest.py) I’ve got random selection of the tails:

```
dennis@...:$ python decrypt.py encrypted.xml | shuf | head -20
00000000: 48 E1 7A 14 AE FF 62 40 DD 07 05 08 H..b@....
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....@Z@....
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....V@....
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....a@....
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....c@....
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....@@....
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....@@....
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....@@....
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....@@....
00000000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....@@....
```

848
We first see the 0x40 and 0x07 bytes present in each tail. The very last byte s always in 1..0x1F (1..31) range, I’ve checked. The penultimate byte is always in 1..0xC (1..12) range. Wow, that looks like a date! Year can be represented as 16-bit value, and maybe last 4 bytes is date (16 bits for year, 8 bits for month and 8 more for day)? 0x7DD is 2013, 0x7D5 is 2005, etc. Seems fine. This is a date. There are 8 more bytes. Judging by the fact this is database named orders, maybe some kind of sum is present here? I made attempt to interpret it as double-precision IEEE 754 floating point and dump all values!

Some are:

71.0
134.0
51.95
53.0
121.99
96.95
98.95
15.95
85.95
184.99
94.95
29.95
85.0
36.0
130.99
115.95
87.99
127.95
114.0
150.95

Looks like real!

Now we can dump names, sums and dates.

plain:
00000000: 0D 00 FF FE 46 00 52 00 41 00 4E 00 4B 00 49 00 ....F.R.A.N.K.I.
00000010: 45 00 20 00 4A 00 4F 00 48 00 4N 00 53 00 66 66 E. .J.O.H.N.S.ff
00000020: 66 66 9E 61 40 D4 07 06 01 fff.a@....
OrderID= 1 name= FRANKIE JOHNS sum= 140.95 date= 2004 / 6 / 1

plain:
00000000: 0B 00 FF FE 4C 00 4F 00 52 00 49 00 20 00 42 00 ....L.O.R.I. .B.
00000010: 41 00 52 00 52 00 4F 00 4E 00 CD CC CC CC CC CC A.R.R.O.N.......
00000020: 1B 40 D4 07 06 01 .@....
OrderID= 2 name= LORI BARRON sum= 6.95 date= 2004 / 6 / 1

plain:
00000000: 0A 00 FF FE 47 00 41 00 52 00 59 00 20 00 42 00 ....G.A.R.Y. .B.
00000010: 49 00 47 00 47 00 53 00 00 00 00 00 00 00 65 64 I.G.G.S.......e@
00000020: 0D 07 06 01 ....
OrderID= 3 name= GARY BIGGS sum= 174.0 date= 2004 / 6 / 1

plain:
00000000: 0F 00 FF FE 4D 00 45 00 4C 00 49 00 4E 00 44 00 ....M.E.L.I.N.D.
00000010: 41 00 20 00 44 00 4F 00 48 00 45 00 52 00 54 00 A. .D.O.H.E.R.T.
00000020: 59 00 48 E1 7A 14 AE FF 68 40 D4 07 06 02 Y.H.z...h@....
OrderID= 4 name= MELINDA DOHERTY sum= 199.99 date= 2004 / 6 / 2

plain:
00000000: 08 00 FF FE 4C 00 45 00 4E 00 41 00 20 00 40 00 ....L.E.N.A. .M.
00000010: 41 00 52 00 43 00 55 00 53 00 00 00 00 00 00 60 A.R.C.U.S........
00000020: 66 40 D4 07 06 03 f@....
OrderID= 5 name= LENA MARCUS sum= 179.0 date= 2004 / 6 / 3

This is some kind of OOP serialization, i.e., packing differently typed values into binary buffer for storing and/or transmitting.

**8.9.8 Noise at the end**

The only question remaining is that sometimes, tail is bigger:

```
00000000: 0E 00 FF FF 00 45 00 4E 00 49 00 45 00 ....T.H.E.R.E.S.
00000010: 45 00 20 00 54 00 55 00 54 00 54 00 4C 00 45 00 E. .T.U.T.T.L.E.
00000020: 66 66 66 66 66 66 66 66 66 63 40 D4 07 07 1A 00 07 07 19 ffff.c@
```

OrderID= 172 name= THERESE TUTTLE sum= 152.95 date= 2004 / 7 / 26

(00 07 07 19 bytes are not used and is ballast.)

```
00000000: 0C 00 FF FE 4D 00 45 00 4C 00 49 00 49 00 ....M.E.L.A.N.I.
00000010: 45 00 20 00 4B 00 49 00 52 00 48 00 00 00 00 00 E. .K.I.R.K.....
00000020: 00 20 64 64 64 0D 07 09 02 00 02 .... . d@......
```

OrderID= 286 name= MELANIE KIRK sum= 161.0 date= 2004 / 9 / 2

(00 02 are not used.)

After close examination, we can see, that the noise at the end of tail is just left from previous encryption!

Here are two subsequent buffers:

```
00000000: 10 00 FF FE 42 00 4F 00 4E 00 4E 00 49 00 45 00 ....B.O.N.N.I.E.
00000010: 20 00 47 00 4F 00 4C 00 45 00 CD CC 00 20 64 40 D4 07 09 02 ....
```

OrderID= 171 name= BONNIE GOLDSTEIN sum= 44.95 date= 2004 / 7 / 25

Another two subsequent buffers:

```
00000000: 0E 00 FF FE 54 00 48 00 45 00 52 00 45 00 53 00 ....T.H.E.R.E.S.
00000010: 45 00 20 00 54 00 55 00 54 00 54 00 4C 00 45 00 E. .T.U.T.T.L.E.
00000020: 66 66 66 66 66 66 66 66 66 63 40 D4 07 07 1A 00 07 07 19 ffff.c@
```

OrderID= 172 name= THERESE TUTTLE sum= 152.95 date= 2004 / 7 / 26

(The last 07 07 19 bytes are copied from the previous plaintext buffer.)

Here are two subsequent buffers:

```
00000000: 0D 00 FF FE 4L 00 4F 00 4E 00 4E 00 45 00 ....L.O.R.E.N.E.
00000010: 20 00 4F 00 54 00 4F 00 4F 00 4L 00 CD CC 00 20 64 40 D4 07 09 02 ....
```

OrderID= 285 name= LORENE OTOOLE sum= 120.95 date= 2004 / 9 / 2

The last 02 byte has been copied from the previous plaintext buffer.

It's possible if the buffer used while encrypting is global and/or isn't clearing before each encryption. The final buffer size is also chaotic, nevertheless, the bug left uncaught because it doesn't affect decrypting process, which just ignores noise at the end. This is common mistake. It's been present in OpenSSL (Heartbleed bug).

**8.9.9 Conclusion**

Summary: every practicing reverse engineer should be familiar with major crypto algorithms and also major cryptographical modes. Some books about it: 12.1.10 on page 1000.

Encrypted database contents has been artificially constructed by me for the sake of demonstration. I've got most popular USA names and surnames from there: [http://stackoverflow.com/questions/1803628/raw-list-of-person-names](http://stackoverflow.com/questions/1803628/raw-list-of-person-names), and combined them randomly. Dates and sums were also generated randomly.

All files used in this part are here: [https://beginners.re/current-tree/examples/encrypted_DB1](https://beginners.re/current-tree/examples/encrypted_DB1).

Nevertheless, many features like these I've observed in real-world software applications. This example is based on them.
8.9.10 Post Scriptum: brute-forcing IV

The case you have just seen has been artificially constructed, but is based on a real application I’ve reverse engineered. When I’ve been working on it, I first noticed that IV has been generating using some 32-bit number, and I wasn’t able to find a link between this value and OrderID. So I prepared to use brute-force, which is indeed possible here.

It’s not a problem to enumerate all 32-bit values and try each as a base for IV. Then you decrypt the first 16-byte block and check for zero bytes, which are always at fixed places.

8.10 Overclocking Cointerra Bitcoin miner

There was Cointerra Bitcoin miner, looking like that:

![Image](image_url)

Figure 8.14: Board

And there was also (possibly leaked) utility\(^\text{24}\) which can set clock rate for the board. It runs on additional BeagleBone Linux ARM board (small board at bottom of the picture).

And the author was once asked, is it possible to hack this utility to see, which frequency can be set and which are not. And it is possible to tweak it?

The utility must be executed like that: \(./\text{cointool-overclock 0 0 900}\), where 900 is frequency in MHz. If the frequency is too high, utility will print “Error with arguments” and exit.

This is a fragment of code around reference to “Error with arguments” text string:

```plaintext
...
```

\(^{24}\)Can be downloaded here: [https://beginners.re/current-tree/examples/bitcoin_miner/files/cointool-overclock](https://beginners.re/current-tree/examples/bitcoin_miner/files/cointool-overclock)
.text:0000ABC4  STR  R3, [R11,#var_2C]
.text:0000ABC8  MOV  R3, #optind
.text:0000ABD0  LDR  R3, [R3]
.text:0000ABD4  ADD  R3, R3, #1
.text:0000ABD8  MOV  R3, R3,LSL#2
.text:0000ABDC  LDR  R2, [R11,#argv]
.text:0000ABE0  ADD  R3, R2, R3
.text:0000ABE4  LDR  R3, [R3]
.text:0000ABE8  MOV  R0, R3 ; nptr
.text:0000ABEC  MOV  R1, #0 ; endptr
.text:0000ABF0  MOV  R2, #0 ; base
.text:0000ABF4  BL  strtoll
.text:0000ABF8  MOV  R2, R0
.text:0000ABFC  MOV  R3, R1
.text:0000AC00  MOV  R3, R2
.text:0000AC04  STR  R3, [R11,#var_2C]
.text:0000AC08  MOV  R3, #optind
.text:0000AC10  LDR  R3, [R3]
.text:0000AC14  ADD  R3, R3, #2
.text:0000AC18  MOV  R3, R3,LSL#2
.text:0000AC1C  LDR  R2, [R11,#argv]
.text:0000AC20  ADD  R3, R2, R3
.text:0000AC24  LDR  R3, [R3]
.text:0000AC28  MOV  R0, R3 ; nptr
.text:0000AC2C  MOV  R1, #0 ; endptr
.text:0000AC30  MOV  R2, #0 ; base
.text:0000AC34  BL  strtoll
.text:0000AC38  MOV  R2, R0
.text:0000AC3C  MOV  R3, R1
.text:0000AC40  MOV  R3, R2
.text:0000AC44  STR  R3, [R11,#third_argument]
.text:0000AC48  LDR  R3, [R11,#var_28]
.text:0000AC4C  CMP  R3, #0
.text:0000AC50  BLT  errors_with_arguments
.text:0000AC54  LDR  R3, [R11,#var_28]
.text:0000AC58  CMP  R3, #1
.text:0000AC5C  BGT  errors_with_arguments
.text:0000AC60  LDR  R3, [R11,#var_2C]
.text:0000AC64  CMP  R3, #0
.text:0000AC68  BLT  errors_with_arguments
.text:0000AC6C  LDR  R3, [R11,#var_2C]
.text:0000AC70  CMP  R3, #3
.text:0000AC74  BGT  errors_with_arguments
.text:0000AC78  LDR  R3, [R11,#third_argument]
.text:0000AC7C  CMP  R3, #0x31
.text:0000AC80  BLE  errors_with_arguments
.text:0000AC84  LDR  R2, [R11,#third_argument]
.text:0000AC88  MOV  R3, #950
.text:0000AC8C  CMP  R2, R3
.text:0000AC90  BGT  errors_with_arguments
.text:0000AC94  LDR  R2, [R11,#third_argument]
.text:0000AC98  MOV  R3, #0x51EB851F
.text:0000ACA0  SMULL  R1, R3, R2
.text:0000ACA4  MOV  R1, R3,ASR#4
.text:0000ACA8  MOV  R3, R2,ASR#31
.text:0000ACAC  RSB  R3, R3, R1
.text:0000ACB0  MOV  R1, #50
.text:0000ACB4  MUL  R3, R1, R3
.text:0000ACBB  RSB  R3, R3, R2
.text:0000ACBC  CMP  R3, #0
.text:0000ACC0  BEQ  loc_ACEC
.text:0000ACC4
.text:0000ACC4  errors_with_arguments
.text:0000ACC4
.text:0000ACC4  LDR  R3, [R11,#argv]
.text:0000ACC8  LDR  R3, [R3]
.text:0000ACCC  MOV  R0, R3 ; path
.text:0000ACD0  BL  __xpg_basename
.text:0000ACD4  MOV  R3, R0
CODE XREF: main+6C
LDR R2, [R11,#third_argument]
MOV R3, #499
CMP R2, R3
BGT loc_AD08
MOV R3, #0x64
STR R3, [R11,#unk_constant]
B jump_to_write_power

CODE XREF: main+6A4
LDR R2, [R11,#third_argument]
MOV R3, #799
CMP R2, R3
BGT loc_AD24
MOV R3, #0xF5
STR R3, [R11,#unk_constant]
B jump_to_write_power

CODE XREF: main+6C0
LDR R2, [R11,#third_argument]
MOV R3, #999
CMP R2, R3
BGT loc_AD40
MOV R3, #0x5A
STR R3, [R11,#unk_constant]
B jump_to_write_power

CODE XREF: main+6F8
LDR R2, [R11,#third_argument]
MOV R3, #1099
CMP R2, R3
BGT jump_to_write_power
MOV R3, #0x50
STR R3, [R11,#unk_constant]

jump_to_write_power

LDR R3, [R11,#var_28]
UXTB R1, R3
LDR R3, [R11,#var_2C]
UXTB R2, R3
LDR R3, [R11,#unk_constant]
UXTB R3, R3
LDR R0, [R11,#third_argument]
UXTB R0, R0
STR R0, [SP,#0x44+var_44]
LDR R0, [R11,#var_24]
BL write_power
LDR R0, [R11,#var_24]
MOV R1, #0x5A
BL read_loop

main+6CC ...

main+6CC
Function names were present in debugging information of the original binary, like write_power, read_loop. But labels inside functions were named by me.

optind name looks familiar. It is from getopt *NIX library intended for command-line parsing—well, this is exactly what happens inside. Then, the 3rd argument (where frequency value is to be passed) is converted from a string to a number using a call to strtoll() function.

The value is then checked against various constants. At 0xACEC, it’s checked, if it is lesser or equal to 499, and if it is so, 0x64 is to be passed to write_power() function (which sends a command through USB using send_msg()). If it is greater than 499, jump to 0xAD08 is occurred.

At 0xAD08 it’s checked, if it’s lesser or equal to 799. 0x5F is then passed to write_power() function in case of success.

There are more checks: for 899 at 0xAD24, for 0x999 at 0xAD40 and finally, for 1099 at 0xAC88, and if it is greater, error message will be displayed and the utility will finish.

Now the table between frequency in MHz and value passed to write_power() function:

<table>
<thead>
<tr>
<th>MHz</th>
<th>hexadecimal</th>
<th>decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>499MHz</td>
<td>0x64</td>
<td>100</td>
</tr>
<tr>
<td>799MHz</td>
<td>0x5f</td>
<td>95</td>
</tr>
<tr>
<td>899MHz</td>
<td>0x5a</td>
<td>90</td>
</tr>
<tr>
<td>999MHz</td>
<td>0x55</td>
<td>85</td>
</tr>
<tr>
<td>1099MHz</td>
<td>0x50</td>
<td>80</td>
</tr>
</tbody>
</table>

As it seems, a value passed to the board is gradually decreasing during frequency increasing.

Now we see that value of 950MHz is a hardcoded limit, at least in this utility. Can we trick it?

Let’s back to this piece of code:

```
.text:0000AC84  LDR  R2, [R11,#third_argument]
.text:0000AC88  MOV  R3, #950
.text:0000AC8C  CMP  R2, R3
.text:0000AC90  BGT  errors_with_arguments ; I've patched here to 00 00 00 00
```

We must disable BGT branch instruction at 0xAC90 somehow. And this is ARM in ARM mode, because, as we see, all addresses are increasing by 4, i.e., each instruction has size of 4 bytes. NOP (no operation) instruction in ARM mode is just four zero bytes: 00 00 00 00. So by writing four zeros at 0xAC90 address (or physical offset in file 0x2C90) we can disable the check.

Now it’s possible to set frequencies up to 1050MHz. Even more is possible, but due to the bug, if input value is greater than 1099, a value as is in MHz will be passed to the board, which is incorrect.

I didn’t go further, but if I had to, I would try to decrease a value which is passed to write_power() function.

Now the scary piece of code which I skipped at first:

```
.text:0000AC94  LDR  R2, [R11,#third_argument]
.text:0000AC98  MOV  R3, #50
.text:0000AC9F  MOV  R3, #0x51E8851F
.text:0000ACAB  SMULL R1, R3, R3, R2 ; R3=3rg_arg/3.125
.text:0000ACAD  MOV  R1, R3, ASR#4 ; R1=R3/16=3rg_arg/50
.text:0000ACAE  MOV  R3, R2, ASR#31 ; R3=MSB(3rg_arg)
.text:0000ACAF  RSB R3, R3, R1 ; R3=3rd_arg/50
.text:0000ACB0  MOV  R1, #50
.text:0000ACB4  MUL  R3, R1, R3 ; R3=50*(3rd_arg/50)
```
Division via multiplication is used here, and constant is 0x51EB851F. I wrote a simple programmer’s calculator\textsuperscript{25} for myself. And I have there a feature to calculate modulo inverse.

\begin{verbatim}
modinv32(0x51EB851F)
Warning, result is not integer: 3.125000
(unsigned) dec: 3 hex: 0x3 bin: 11
\end{verbatim}

That means that SMULL instruction at 0xACA0 is basically divides 3rd argument by 3.125. In fact, all modinv32() function in my calculator does, is this:

\[
\frac{1}{\text{input}} = \frac{2^{32}}{\text{input}}
\]

Then there are additional shifts and now we see than 3rd argument is just divided by 50. And then it’s multiplied by 50 again. Why? This is simplest check, if the input value is can be divided by 50 evenly. If the value of this expression is non-zero, \(x\) can’t be divided by 50 evenly:

\[
x - ((x/50) \cdot 50)
\]

This is in fact simple way to calculate remainder of division.
And then, if the remainder is non-zero, error message is displayed. So this utility takes frequency values in form like 850, 900, 950, 1000, etc., but not 855 or 911.
That’s it! If you do something like that, please be warned that you may damage your board, just as in case of overclocking other devices like CPUs, GPU\textsuperscript{26}, etc. If you have a Cointerra board, do this on your own risk!

### 8.11 Breaking simple executable cryptor

I’ve got an executable file which is encrypted by relatively simple encryption. Here is it (only executable section is left here).

First, all encryption function does is just adds number of position in buffer to the byte. Here is how this can be encoded in Python:

\begin{verbatim}
Listing 8.9: Python script
#!/usr/bin/env python
def e(i, k):
    return chr(ord(i)+k) % 256
def encrypt(buf):
    return e(buf[0], 0)+ e(buf[1], 1)+ e(buf[2], 2) + e(buf[3], 3)+ e(buf[4], 4)+ e(buf[5], 5)+
    e(buf[6], 6)+ e(buf[7], 7)+
    e(buf[8], 8)+ e(buf[9], 9)+ e(buf[10], 10)+ e(buf[11], 11)+ e(buf[12], 12)+ e(buf[13], 13)+
    e(buf[14], 14)+ e(buf[15], 15)
\end{verbatim}

Hence, if you encrypt buffer with 16 zeros, you’ll get 0, 1, 2, 3 ... 12, 13, 14, 15.
Propagating Cipher Block Chaining (PCBC) is also used, here is how it works:

\textsuperscript{25}https://yurichev.com/progcalc/
\textsuperscript{26}Graphics Processing Unit
The problem is that it's too boring to recover IV (Initialization Vector) each time. Brute-force is also not an option, because IV is too long (16 bytes). Let's see, if it's possible to recover IV for arbitrary encrypted executable file?

Let's try simple frequency analysis. This is 32-bit x86 executable code, so let's gather statistics about most frequent bytes and opcodes. I tried huge oracle.exe file from Oracle RDBMS version 11.2 for windows x86 and I've found that the most frequent byte (no surprise) is zero (10%). The next most frequent byte is (again, no surprise) 0xFF (5%). The next is 0x8B (5%).

0x8B is opcode for MOV, this is indeed one of the most busy x86 instructions. Now what about popularity of zero byte? If compiler needs to encode value bigger than 127, it has to use 32-bit displacement instead of 8-bit one, but large values are very rare, so it is padded by zeros. This is at least in LEA, MOV, PUSH, CALL.

For example:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>8D B0 28 01 00 00</td>
<td>lea esi, [eax+128h]</td>
</tr>
<tr>
<td>8D BF 40 38 00 00</td>
<td>lea edi, [edi+3840h]</td>
</tr>
</tbody>
</table>

Displacements bigger than 127 are very popular, but they are rarely exceeds 0x10000 (indeed, such large memory buffers/structures are also rare).

Same story with MOV, large constants are rare, the most heavily used are 0, 1, 10, 100, 2^n, and so on. Compiler has to pad small constants by zeros to represent them as 32-bit values:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF 02 00 00 00 00</td>
<td>mov edi, 2</td>
</tr>
<tr>
<td>BF 01 00 00 00 00</td>
<td>mov edi, 1</td>
</tr>
</tbody>
</table>

Now about 00 and FF bytes combined: jumps (including conditional) and calls can pass execution flow forward or backwards, but very often, within the limits of the current executable module. If forward, displacement is not very big and also padded with zeros. If backwards, displacement is represented as negative value, so padded with FF bytes. For example, transfer execution flow forward:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E8 43 0C 00 00 00</td>
<td>call _function1</td>
</tr>
<tr>
<td>E8 5C 00 00 00 00</td>
<td>call _function2</td>
</tr>
<tr>
<td>0F 84 F0 0A 00 00</td>
<td>jz loc_4F09A0</td>
</tr>
<tr>
<td>0F 84 EB 00 00 00</td>
<td>jz loc_4EFBB8</td>
</tr>
</tbody>
</table>

Backwards:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E8 79 0C FE FF</td>
<td>call _function1</td>
</tr>
<tr>
<td>E8 F4 16 FF FF</td>
<td>call _function2</td>
</tr>
<tr>
<td>0F 84 F8 FB FF FF</td>
<td>jz loc_8212BC</td>
</tr>
<tr>
<td>0F 84 06 FD FF FF</td>
<td>jz loc_FF1E7D</td>
</tr>
</tbody>
</table>

FF byte is also very often occurred in negative displacements like these:
So far so good. Now we have to try various 16-byte keys, decrypt executable section and measure how often 00, FF and 8B bytes are occurred. Let’s also keep in sight how PCBC decryption works:

Figure 8.16: Propagating Cipher Block Chaining decryption (image is taken from Wikipedia article)

The good news is that we don’t really have to decrypt whole piece of data, but only slice by slice, this is exactly how I did in my previous example: 9.1.5 on page 924.

Now I’m trying all possible bytes (0..255) for each byte in key and just pick the byte producing maximal amount of 00/FF/8B bytes in a decrypted slice:

```python
#!/usr/bin/env python
import sys, hexdump, array, string, operator

KEY_LEN=16

def chunks(l, n):
    # split n by l-byte chunks
    # https://stackoverflow.com/q/312443
    n = max(1, n)
    return [l[i:i + n] for i in range(0, len(l), n)]

def read_file(fname):
    file=open(fname, mode='rb')
    content=file.read()
    file.close()
    return content

def decrypt_byte (c, key):
    return chr((ord(c)-key) % 256)

def XOR_PCBC_step (IV, buf, k):
    prev=IV
    rt=""
    for c in buf:
        new_c=decrypt_byte(c, k)
        plain=chr(ord(new_c)^ord(prev))
        prev=chr(ord(c)^ord(plain))
        rt=rt+plain
    return rt

each_Nth_byte=['' for i in range(KEY_LEN)]

content=read_file(sys.argv[1])
```
# split input by 16-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    stat={}
    for i in range(256):
        tmp_key=chr(i)
        tmp=XOR_PCBC_step(tmp_key,each_Nth_byte[N], N)
        # count 0, FFs and 8Bs in decrypted buffer:
        important_bytes=tmp.count('\x00')+tmp.count('\xFF')+tmp.count('\x8B')
        stat[i]=important_bytes
    sorted_stat = sorted(stat.iteritems(), key=operator.itemgetter(1), reverse=True)
    print sorted_stat[0]

(Source code can be downloaded here.)

I run it and here is a key for which 00/FF/8B bytes presence in decrypted buffer is maximal:

<table>
<thead>
<tr>
<th>N</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(147, 1224)</td>
</tr>
<tr>
<td>1</td>
<td>(94, 1327)</td>
</tr>
<tr>
<td>2</td>
<td>(252, 1223)</td>
</tr>
<tr>
<td>3</td>
<td>(218, 1266)</td>
</tr>
<tr>
<td>4</td>
<td>(38, 1209)</td>
</tr>
<tr>
<td>5</td>
<td>(192, 1378)</td>
</tr>
<tr>
<td>6</td>
<td>(199, 1204)</td>
</tr>
<tr>
<td>7</td>
<td>(213, 1332)</td>
</tr>
<tr>
<td>8</td>
<td>(225, 1251)</td>
</tr>
<tr>
<td>9</td>
<td>(112, 1223)</td>
</tr>
<tr>
<td>10</td>
<td>(143, 1177)</td>
</tr>
<tr>
<td>11</td>
<td>(108, 1286)</td>
</tr>
<tr>
<td>12</td>
<td>(10, 1164)</td>
</tr>
<tr>
<td>13</td>
<td>(3, 1271)</td>
</tr>
<tr>
<td>14</td>
<td>(128, 1253)</td>
</tr>
<tr>
<td>15</td>
<td>(232, 1330)</td>
</tr>
</tbody>
</table>

Let’s write decryption utility with the key we got:

```
#!/usr/bin/env python
import sys, hexdump, array

def xor_strings(s,t):
    # https://en.wikipedia.org/wiki/XOR_cipher#Example_implementation
    """xor two strings together""
    return ''.join(chr(ord(a)^ord(b)) for a,b in zip(s,t))

IV=array.array('B', [147, 94, 252, 218, 38, 192, 199, 213, 225, 112, 143, 108, 10, 3, 128, \n 232]).tostring()

def chunks(l, n):
```
n = max(1, n)
return [l[i:i + n] for i in range(0, len(l), n)]

def read_file(fname):
  file=open(fname, mode='rb')
  content=file.read()
  file.close()
  return content

def decrypt_byte(i, k):
  return chr((ord(i)-k) % 256)

def decrypt(buf):
  return ''.join(decrypt_byte(buf[i], i) for i in range(16))

fout=open(sys.argv[2], mode='wb')
prev=IV
content=read_file(sys.argv[1])
tmp=chunks(content, 16)
for c in tmp:
  new_c=decrypt(c)
  p=xor_strings(new_c, prev)
  prev=xor_strings(c, p)
  fout.write(p)
fout.close()

(Source code can be downloaded here.)

Let's check resulting file:

$ objdump -b binary -m i386 -D decrypted.bin
...
...
here.

In fact, this is text section from regedit.exe from Windows 7. But this example is based on a real case I encountered, so just executable is different (and key), algorithm is the same.

8.11.1 Other ideas to consider

What if I would fail with such simple frequency analysis? There are other ideas on how to measure correctness of decrypted/decompressed x86 code:

- Many modern compilers aligns functions on 0x10 border. So the space left before is filled with NOPs (0x90) or other NOP instructions with known opcodes: 1.7 on page 1024.
- Perhaps, the most frequent pattern in any assembly language is function call: PUSH chain / CALL / ADD ESP, X. This sequence can easily detected and found. I've even gathered statistics about average number of function arguments: 11.2 on page 980. (Hence, this is average length of PUSH chain.)

Read more about incorrectly/correctly disassembled code: 5.11 on page 719.

8.12 SAP

8.12.1 About SAP client network traffic compression

(This article first appeared in my blog, in 13-Jul-2010.)

(Tracing the connection between the TDW_NOCOMPRESS SAPGUI environment variable and the pesky annoying pop-up window and the actual data compression routine.)

It is known that the network traffic between SAPGUI and SAP is not encrypted by default, but compressed (see here and here).

It is also known that by setting the environment variable TDW_NOCOMPRESS to 1, it is possible to turn the network packet compression off.

But you will see an annoying pop-up window that cannot be closed:

27SAP GUI client
28http://go.yurichev.com/17221
29blog.yurichev.com
Let’s see if we can remove the window somehow.

But before this, let’s see what we already know.

First: we know that the environment variable `TDW_NOCOMPRESS` is checked somewhere inside the SAPGUI client.

Second: a string like “data compression switched off” must be present somewhere in it.

With the help of the FAR file manager\(^3\), we can find that both of these strings are stored in the `SAPguilib.dll` file.

So let’s open `SAPguilib.dll` in IDA and search for the `TDW_NOCOMPRESS` string. Yes, it is present and there is only one reference to it.

We see the following fragment of code (all file offsets are valid for SAPGUI 720 win32, `SAPguilib.dll` file version 7200,1,0,9009):

```plaintext
.text:6440D51B lea   eax, [ebp+2108h+var_211C]
.text:6440D51E push  eax ; int
.text:6440D51F push  offset aTdw_nocompress ; "TDW_NOCOMPRESS"
.text:6440D524 mov    byte ptr [edi+15h], 0
.text:6440D528 call   chk_env
.text:6440D52D pop    ecx
.text:6440D52E pop    ecx
.text:6440D52F push   offset byte 64443AF8
.text:6440D534 lea    ecx, [ebp+2108h+var_211C]

; demangled name: int ATL::CStringT::Compare(char const *)const
.text:6440D537 call   ds:mfc90_1603
.text:6440D53D test   eax, eax
.text:6440D53F jz    short loc_6440D55A
.text:6440D541 lea    ecx, [ebp+2108h+var_211C]

; demangled name: const char* ATL::CSimpleStringT::operator PCXSTR
.text:6440D544 call   ds:mfc90_910
```

\(^3\)http://go.yurichev.com/17347
The string returned by `chk_env()` via its second argument is then handled by the MFC string functions and then `atoi()` is called. After that, the numerical value is stored in edi+15h.

Also take a look at the `chk_env()` function (we gave this name to it manually):

```assembly
; (demangled name) ATL::CStringT::operator=(char const *)
.text:64413F3C proc near
.text:64413F3C call ds:mfc90_820
.text:64413F42 mov eax, [ebp+VarName]
.text:64413F45 push eax
.text:64413F46 mov ecx, [ebp+DstSize]
.text:64413F49 push ecx
.text:64413F4A mov edx, [ebp+DstBuf]
.text:64413F4D push edx
.text:64413F4E lea eax, [ebp+ReturnSize]
.text:64413F51 push eax
.text:64413F52 call ds:getenv_s
.text:64413F55 add esp, 10h
.text:64413F5E cmp [ebp+var_8], 0
.text:64413F62 jz short loc_64413F68
.text:64413F64 xor eax, eax
.text:64413F66 jmp short loc_64413F68:
.text:64413F68 cmp [ebp+DstSize], 0
.text:64413F6C jnz short loc_64413F72
.text:64413F6E xor eax, eax
.text:64413F70 jmp short loc_64413F68
.text:64413F72 loctext
.text:64413F72 loc_64413F72:
.text:64413F72 mov ecx, [ebp+DstSize]
.text:64413F75 push ecx
.text:64413F76 mov ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT<char, 1>::Preallocate(int)
.text:64413F79 call ds:mfc90_2691
.text:64413F7F mov [ebp+DstBuf], eax
.text:64413F82 mov edx, [ebp+VarName]
.text:64413F85 push edx
.text:64413F86 mov eax, [ebp+DstSize]
.text:64413F89 push eax
.text:64413F8A mov ecx, [ebp+DstBuf]
.text:64413F8D push ecx
.text:64413F8E lea edx, [ebp+DstSize]
.text:64413F91 push edx
```

31Standard C library function that converts the digits in a string to a number
Yes. The `getenv_s()` function is a Microsoft security-enhanced version of `getenv()`.

There are also some MFC string manipulations.

Lots of other environment variables are checked as well. Here is a list of all variables that are being checked and what SAPGUI would write to its trace log when logging is turned on:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Trace Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPTRACE</td>
<td>&quot;GUI-OPTION: Trace set to %d&quot;</td>
</tr>
<tr>
<td>TDW_HEXDUMP</td>
<td>&quot;GUI-OPTION: Hexdump enabled&quot;</td>
</tr>
<tr>
<td>TDW_WORKDIR</td>
<td>&quot;GUI-OPTION: working directory %s&quot;</td>
</tr>
<tr>
<td>TDW_SPLASHSCREENOFF</td>
<td>&quot;GUI-OPTION: Splash Screen Off&quot;</td>
</tr>
<tr>
<td>TDW_REPLYTIMEOUT</td>
<td>&quot;GUI-OPTION: reply timeout %d milliseconds&quot;</td>
</tr>
<tr>
<td>TDW_PLAYBACKTIMEOUT</td>
<td>&quot;GUI-OPTION: PlaybackTimeout set to %d milliseconds&quot;</td>
</tr>
<tr>
<td>TDW_NOCOMPRESS</td>
<td>&quot;GUI-OPTION: no compression read&quot;</td>
</tr>
<tr>
<td>TDW_EXPERT</td>
<td>&quot;GUI-OPTION: expert mode&quot;</td>
</tr>
<tr>
<td>TDW_PLAYBACKPROGRESS</td>
<td>&quot;GUI-OPTION: PlaybackProgress&quot;</td>
</tr>
<tr>
<td>TDW_PLAYBACKNETTRAFFIC</td>
<td>&quot;GUI-OPTION: PlaybackNetTraffic&quot;</td>
</tr>
<tr>
<td>TDW_PLAYLOG</td>
<td>&quot;GUI-OPTION: /PlayLog is YES, file %s&quot;</td>
</tr>
<tr>
<td>TDW_PLAYTIME</td>
<td>&quot;GUI-OPTION: /PlayTime set to %d milliseconds&quot;</td>
</tr>
<tr>
<td>TDW_LOGFILE</td>
<td>&quot;GUI-OPTION: TDW_LOGFILE %s&quot;</td>
</tr>
<tr>
<td>TDW_WAN</td>
<td>&quot;GUI-OPTION: WAN - low speed connection enabled&quot;</td>
</tr>
<tr>
<td>TDW_FULLMENU</td>
<td>&quot;GUI-OPTION: FullMenu enabled&quot;</td>
</tr>
<tr>
<td>SAP_CP / SAP_CODEPAGE</td>
<td>&quot;GUI-OPTION: SAP_CODEPAGE %d&quot;</td>
</tr>
<tr>
<td>UPDOWNLOAD_CP</td>
<td>&quot;GUI-OPTION: UPDOWNLOAD_CP %d&quot;</td>
</tr>
<tr>
<td>SNC_PARTNERNAME</td>
<td>&quot;GUI-OPTION: SNC name %s&quot;</td>
</tr>
<tr>
<td>SNC_QOP</td>
<td>&quot;GUI-OPTION: SNC_QOP %s&quot;</td>
</tr>
<tr>
<td>SNC_LIB</td>
<td>&quot;GUI-OPTION: SNC is set to: %s&quot;</td>
</tr>
<tr>
<td>SAPGUI_INPLACE</td>
<td>&quot;GUI-OPTION: environment variable SAPGUI_INPLACE is on&quot;</td>
</tr>
</tbody>
</table>

The settings for each variable are written in the array via a pointer in the EDI register. EDI is set before the function call:
Now, can we find the data record mode switched on string?

Yes, and the only reference is in CDwsGui::PrepareInfoWindow().

How do we get know the class/method names? There are a lot of special debugging calls that write to the log files, like:

...or:

It is very useful.

So let’s see the contents of this pesky annoying pop-up window’s function:

| .text:6440F4F | push offset aEnvironmentInf; "Environment information:
| .text:6440F4F | push eax
| .text:6440F4F | push offset aWorkingDirecto; "working directory: '%s'
| .text:6440F4F | push offset aMfc90_316
| .text:6440F4F | push [ebp+var_4], ebx
| .text:6440F4F | lea edi, [esi+2854h]
| .text:6440F4F | push offset aEnvironmentInf
| .text:6440F4F | mov ecx, edi
| ; demangled name: ATL::CStringT::operator=(char const *)
| .text:6440F4F | push offset aMfc90_2539
| .text:6440F4F | mov ebx, ds:mfc90_2539
| .text:6440F4F | lea ecx, [ebp+var_10]
| .text:6440F4F | push offset aWorkingDirecto
| ; demangled name: ATL::CStringT::Format(char const *,...) |
text

| .text:64404F98   | call   ebx ; mfc90_2539 |
| .text:64404F9A   | add esp, 0Ch          |
| .text:64404FD0   | lea eax, [ebp+var_14] |
| .text:64404FA0   | push eax              |
| .text:64404FA1   | mov ecx, edi          |

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)

| .text:64404FA3   | call ds:mfc90_941     |
| .text:64404FA9   | loc_64404FA9:        |
| .text:64404FA9   | mov eax, [esi+38h]   |
| .text:64404FAE   | test eax, eax        |
| .text:64404F80   | push eax             |
| .text:64404F81   | lea eax, [ebp+var_14]|
| .text:64404F84   | push offset aTraceLevelDAct ; "trace level %d activated\n"
| .text:64404F89   | push eax             |

; demangled name: ATL::CStringT::Format(char const *,...)

| .text:64404FBA   | call ebx ; mfc90_2539  |
| .text:64404FBC   | add esp, 0Ch           |
| .text:64404FBE   | lea eax, [ebp+var_14]  |
| .text:64404FC2   | push eax              |
| .text:64404FC3   | mov ecx, edi          |

; demangled name: ATL::CStringT::operator+=(class ATL::CSimpleStringT<char, 1> const &)

| .text:64404FC5   | call ds:mfc90_941     |
| .text:64404FCD   | xor ebx, ebx          |
| .text:64404FCE   | inc ebx              |
| .text:64404FD1   | jmp short loc_64404FD6|
| .text:64404FD3   | loc_64404FD3:        |
| .text:64404FD3   | xor ebx, ebx          |
| .text:64404FD5   | inc ebx              |
| .text:64404FD6   | loc_64404FD6:        |
| .text:64404FD6   | cmp [esi+38h], ebx   |
| .text:64404FD9   | jbe short loc_64404FF1|
| .text:64404FD8   | cmp dword ptr [esi+2978h], 0 |
| .text:64404DE2   | jz short loc_64404FF1 |
| .text:64404DE4   | push offset aHexdumpInTrace ; "hexdump in trace activated\n"
| .text:64404DE9   | mov ecx, edi         |

; demangled name: ATL::CStringT::operator+=(char const *)

| .text:64404FEB   | call ds:mfc90_945     |
| .text:64404FF1   | loc_64404FF1:        |
| .text:64404FF1   | cmp byte ptr [esi+78h], 0 |
| .text:64404FF5   | jz short loc 64405007 |
| .text:64404FF7   | push offset aLoggingActivat ; "logging activated\n"
| .text:64404FFC   | mov ecx, edi         |

; demangled name: ATL::CStringT::operator+=(char const *)

| .text:64404FFE   | call ds:mfc90_945     |
| .text:64405004   | mov [ebp+var_10], ebx|
| .text:64405007   | loc_64405007:        |
| .text:64405007   | cmp byte ptr [esi+30h], 0 |
| .text:6440500B   | jz short bypass      |
| .text:6440500D   | push offset aDataCompressio ; 
| .text:64405012   | mov ecx, edi         |

; demangled name: ATL::CStringT::operator+=(char const *)

| .text:64405014   | call ds:mfc90_945     |
| .text:6440501A   | mov [ebp+var_10], ebx|
| .text:6440501D   | bypass:              |
For maximum data security delete the s...
At the start of the function ECX has a pointer to the object (since it is a thiscall (3.21.1 on page 542)-type of function). In our case, the object obviously has class type of CDwsGui. Depending on the option turned
on in the object, a specific message part is to be concatenated with the resulting message. If the value at address this+0x3D is not zero, the compression is off:

let's check our theory on practice.

JNZ at this line …

...replace it with just JMP, and we get SAPGUI working without the pesky annoying pop-up window appearing!

Now let's dig deeper and find a connection between the 0x15 offset in the load_command_line() (we gave it this name) function and the this+0x3D variable in CDwsGui::PrepareInfoWindow. Are we sure the value is the same?

We are starting to search for all occurrences of the 0x15 value in code. For a small programs like SAPGUI, it sometimes works. Here is the first occurrence we've got:
The function has been called from the function named CDwsGui::CopyOptions! And thanks again for debugging information.

But the real answer is in CDwsGui::Init():

Finally, we understand: the array filled in the load_command_line() function is actually placed in the CDwsGui class, but at address this +0x28. 0x15 + 0x28 is exactly 0x3D. OK, we found the point where the value is copied to.

Let’s also find the rest of the places where the 0x3D offset is used. Here is one of them in the CDwsGui::SapguiRun function (again, thanks to the debugging calls):

Let’s check our findings.
Replace the setz al here with the xor eax, eax / nop instructions, clear the TDW_NOCOMPRESS environment variable and run SAPGUI. Wow! There pesky annoying window is no more (just as expected, because the variable is not set) but in Wireshark we can see that the network packets are not compressed.
any more! Obviously, this is the point where the compression flag is to be set in the \texttt{CConnectionContext} object.

So, the compression flag is passed in the 5th argument of \texttt{CConnectionContext::CreateNetwork}. Inside the function, another one is called:

```
... .text:64403476   push [ebp+compression]
.text:64403479   push [ebp+arg_0]
.text:6440347C   push [ebp+arg_8]
.text:6440347F   push [ebp+arg_4]
.text:64403482   push [ebp+arg_0]
.text:64403485   call CNetwork\_CNetwork
```

The compression flag is passed here in the 5th argument to the \texttt{CNetwork::CNetwork} constructor.

And here is how the \texttt{CNetwork} constructor sets the flag in the \texttt{CNetwork} object according to its 5th argument and another variable which probably could also affect network packets compression:

```
.text:64411DF1   cmp [ebp+compression], esi
.text:64411DF7   jz short set_EAX_to_0
.text:64411DF9   mov al, [ebx+78h] ; another value may affect compression?
.text:64411DFC   cmp al, '3'
.text:64411DFE   jz short set_EAX_to_1
.text:64411E00   cmp al, '4'
.text:64411E02   jnz short set_EAX_to_0
.text:64411E04   .text:64411E08 set_EAX_to_1:
.text:64411E04   xor eax, eax
.text:64411E06   inc eax ; EAX -> 1
.text:64411E07   jmp short loc_64411E0B
.text:64411E09 set_EAX_to_0:
.text:64411E09   xor eax, eax ; EAX -> 0
.text:64411E0B   .text:64411E0B loc_64411E0B:
.text:64411E0B   mov [ebx+3A4h], eax ; EBX is pointer to CNetwork object
```

At this point we know the compression flag is stored in the \texttt{CNetwork} class at address \texttt{this+0x3A4}.

Now let’s dig through SAPgilib.dll for the 0x3A4 value. And here is the second occurrence in \texttt{CDws\_Gui::OnClientMessageWrite} (endless thanks for the debugging information):

```
.text:64406F76 loc_64406F76:
.text:64406F76   loc_64406F99:
.text:64406F99 push dword ptr [edi+2868h] ; int
.text:64406F99   lea edx, [ebp+7728h+var_77A4]
.text:64406F9F   push edx
.text:64406FA2 push 30000  ; int
.text:64406FA3 push 30000  ; int
.text:64406FA8 lea edx, [ebp+7728h+Dst]
.text:64406FAB push edx ; Dst
.text:64406FAC push ecx ; int
.text:64406FAD push eax ; Src
.text:64406FAE push dword ptr [edi+28C0h] ; int
.text:64406F84 call sub 644055C5 ; actual compression routine
.text:64406F89 add esp, 1Ch
.text:64406FBC cmp eax, 0FFFFFFF6h
.text:64406F8F jz short loc_64407004
.text:64406FC1 cmp eax, 1
```

870
Let's take a look in sub_644055C5. In it we can only see the call to memcpy() and another function named (by IDA) sub_64417440.

And, let's take a look inside sub_64417440. What we see is:

Voilà! We’ve found the function that actually compresses the data. As it was shown in past 34, this function is used in SAP and also the open-source MaxDB project. So it is available in source form.

Doing the last check here:

Replace JNZ here for an unconditional JMP. Remove the environment variable TDW_NOCOMPRESS. Voilà!

In Wireshark we see that the client messages are not compressed. The server responses, however, are compressed.

So we found exact connection between the environment variable and the point where data compression routine can be called or bypassed.

### 8.12.2 SAP 6.0 password checking functions

One time when the author of this book have returned again to his SAP 6.0 IDES installed in a VMware box, he figured out that he forgot the password for the SAP* account, then he have recalled it, but then he got this error message «Password logon no longer possible - too many failed attempts», since he’ve made all these attempts in attempt to recall it.

The first extremely good news was that the full disp+work.pdb PDB file is supplied with SAP, and it contain almost everything: function names, structures, types, local variable and argument names, etc. What a lavish gift!

There is TYPEINFODUMP 35 utility for converting PDB files into something readable and grepable.

Here is an example of a function information + its arguments + its local variables:

```plaintext
FUNCTION ThVmcSysEvent
  Address: 10143190 Size: 675 bytes Index: 60483 TypeIndex: 60484
  Type: int NEAR_C ThVmcSysEvent (unsigned int, unsigned char, unsigned short*)
  Flags: 0
PARAMETER events
  Address: Reg335+288 Size: 4 bytes Index: 60488 TypeIndex: 60489
  Type: unsigned int
  Flags: d0
PARAMETER opcode
  Address: Reg335+296 Size: 1 bytes Index: 60490 TypeIndex: 60491
  Type: unsigned char
  Flags: d0
PARAMETER serverName
  Address: Reg335+304 Size: 8 bytes Index: 60492 TypeIndex: 60493
  Type: unsigned short*
  Flags: d0
STATIC_LOCAL_VAR func
```

34 [http://go.yurichev.com/17312](http://go.yurichev.com/17312)
35 [http://go.yurichev.com/17038](http://go.yurichev.com/17038)
And here is an example of some structure:

```plaintext
STRUCT DBSL_STMTID
Size: 120 Variables: 4 Functions: 0 Base classes: 0
MEMBER moduletype
    Type: DBSL_MODULETYPE
    Offset: 0 Index: 3 TypeIndex: 38653
MEMBER module
    Type: wchar_t module[40]
    Offset: 4 Index: 3 TypeIndex: 831
MEMBER stmtnum
    Type: long
    Offset: 84 Index: 3 TypeIndex: 440
MEMBER timestamp
    Type: wchar_t timestamp[15]
    Offset: 88 Index: 3 TypeIndex: 6612
```

Wow!

Another good news: debugging calls (there are plenty of them) are very useful.

Here you can also notice the ct_level global variable, that reflects the current trace level.

There are a lot of debugging inserts in the disp+work.exe file:

```plaintext
cmp     cs:ct_level, 1
jl      short loc_1400375DA
call    DpLock
lea     rcx, aDpxxtool4_c ; "dpxxtool4.c"
mov     edx, 4Eh        ; line
call    CTrcSaveLocation
mov     r8, cs:func_48
mov     rcx, cs:hdl     ; hdl
lea     rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov     r9d, ebx
call    DpTrcErr
call    DpUnlock
```

If the current trace level is bigger or equal to threshold defined in the code here, a debugging message is to be written to the log files like dev_w0, dev_disp, and other dev* files.

Let’s try grepping in the file that we have got with the help of the TYPEINFODUMP utility:

```plaintext
cat "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

We have got:

```plaintext
FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt
FUNCTION ssf_password_decrypt
FUNCTION password_logon_disabled
FUNCTION dySignSkipUserPassword
FUNCTION migrate_password_history
```

---

36 More about trace level: [http://go.yurichev.com/17039](http://go.yurichev.com/17039)
Let's also try to search for debug messages which contain the words «password» and «locked». One of them is the string «user was locked by subsequently failed password logon attempts», referenced in function `password_attempt_limit_exceeded()`.

Other strings that this function can write to a log file are: «password logon attempt will be rejected immediately (preventing dictionary attacks)», «failed-logon lock: expired (but not removed due to ‘read-only’ operation)», «failed-logon lock: expired => removed».

After playing for a little with this function, we noticed that the problem is exactly in it. It is called from the `chckpass()` function —one of the password checking functions.

First, we would like to make sure that we are at the correct point:

Run tracer:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,unicode
```

```
PID=2236|TID=2248|(0) disp+work.exe!chckpass (0x202c770, L"Brewered1", 0x41) (called from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
```

The call path is: `syssigni()` -> `DyISigni()` -> `dychkusr()` -> `usrexist()` -> `chckpass()`.

The number 0x35 is an error returned in `chckpass()` at that point:

```
.text:00000001402ED567 loc_1402ED567: ; CODE XREF: chckpass+B4
.text:00000001402ED567 mov rcx, rbx ; usr02
.text:00000001402ED56A call password_idle_check
.text:00000001402ED56F cmp eax, 33h
.text:00000001402ED572 jz loc_1402EDB4E
.text:00000001402ED578 cmp eax, 36h
.loc:00000001402ED57B jz loc_1402EDB3D
.text:00000001402ED581 xor edx, edx ; usr02_readonly
.text:00000001402ED583 mov rcx, rbx ; usr02
.text:00000001402ED586 call password_attempt_limit_exceeded
.text:00000001402ED58B test al, al
.text:00000001402ED58D jz short loc_1402ED5A0
```
Fine, let's check:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!password_attempt_limit_exceeded,args:4,unicode,rt:0
```

PID=2744|TID=360|disp+work.exe!password_attempt_limit_exceeded (0x202c770, 0, 0x257758, 0) (called from 0x1402e9794 (disp+work.exe!chngpass+0xe4))

PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0

PID=2744|TID=360|disp+work.exe!password_attempt_limit_exceeded -> 1

PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0

```
Excellent! We can successfully login now.
```

By the way, we can pretend we forgot the password, fixing the `chkpass()` function to return a value of 0 is enough to bypass the check:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chkpass,args:3,unicode,rt:0
```

PID=2744|TID=360|disp+work.exe!chkpass (0x202c770, L"bogus", 0x41) (called from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))

PID=2744|TID=360|We modify return value (EAX/RAX) of this function to 0

What also can be said while analyzing the `password_attempt_limit_exceeded()` function is that at the very beginning of it, this call can be seen:

```
lea     rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call    sapgparam
test    rax, rax
jz      short loc_1402E19DE
movzx   eax, word ptr [rax]
cmp     ax, 'N'
jz      short loc_1402E1904
cmp     ax, 'n'
jz      short loc_1402E1904
cmp     ax, '0'
jnz     short loc_1402E190DE
```

Obviously, function `sapgparam()` is used to query the value of some configuration parameter. This function can be called from 1768 different places. It seems that with the help of this information, we can easily find the places in code, the control flow of which can be affected by specific configuration parameters.

It is really sweet. The function names are very clear, much clearer than in the Oracle RDBMS.

It seems that the `disp+work` process is written in C++. Has it been rewritten some time ago?

## 8.13 Oracle RDBMS

### 8.13.1 V$VERSION table in the Oracle RDBMS

Oracle RDBMS 11.2 is a huge program, its main module `oracle.exe` contain approx. 124,000 functions. For comparison, the Windows 7 x86 kernel (`ntoskrnl.exe`) contains approx. 11,000 functions and the Linux 3.9.8 kernel (with default drivers compiled)—31,000 functions.

Let's start with an easy question. Where does Oracle RDBMS get all this information, when we execute this simple statement in SQL*Plus:
SQL> select * from V$VERSION;

And we get:

BANNER
--------------------------------------------------------
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
PL/SQL Release 11.2.0.1.0 - Production
CORE 11.2.0.1.0 Production
TNS for 32-bit Windows: Version 11.2.0.1.0 - Production
NLSRTL Version 11.2.0.1.0 - Production

Let's start. Where in the Oracle RDBMS can we find the string V$VERSION?

In the win32-version, oracle.exe file contains the string, it's easy to see. But we can also use the object (.o) files from the Linux version of Oracle RDBMS since, unlike the win32 version oracle.exe, the function names (and global variables as well) are preserved there.

So, the kqf.o file contains the V$VERSION string. The object file is in the main Oracle-library libserver11.a.

A reference to this text string can find in the kqfviw table stored in the same file, kqf.o:

Listing 8.10: kqf.o

```plaintext
.rodata:0000C4A0 kqfviw dd 0Bh ; DATA XREF: kqfchk:loc_8003A6D
.rodata:0000C4A0 ; kqfgbn+34
.rodata:0000C4A4 dd offset _2_STRING_10102_0 ; "GV$WAITSTAT"
.rodata:0000C4A8 dd 4
.rodata:0000C4AC dd offset _2_STRING_10103_0 ; "NULL"
.rodata:0000C4B0 dd 3
.rodata:0000C4B4 dd 0
.rodata:0000C4B8 dd 195h
.rodata:0000C4BC dd 4
.rodata:0000C4C0 dd 0
.rodata:0000C4C4 dd 0FFFFC1CBh
.rodata:0000C4C8 dd 3
.rodata:0000C4CC dd 0
.rodata:0000C4D0 dd 0Ah
.rodata:0000C4D4 dd offset _2_STRING_10104_0 ; "V$WAITSTAT"
.rodata:0000C4D8 dd 4
.rodata:0000C4DC dd offset _2_STRING_10103_0 ; "NULL"
.rodata:0000C4E0 dd 3
.rodata:0000C4E4 dd 0
.rodata:0000C4E8 dd 4Eh
.rodata:0000C4EC dd 3
.rodata:0000C4F0 dd 0
.rodata:0000C4F4 dd 0FFFFC003h
.rodata:0000C4F8 dd 4
.rodata:0000C4FC dd 0
.rodata:0000C500 dd 5
.rodata:0000C504 dd offset _2_STRING_10105_0 ; "GV$BH"
.rodata:0000C508 dd 4
.rodata:0000C50C dd offset _2_STRING_10103_0 ; "NULL"
.rodata:0000C510 dd 3
.rodata:0000C514 dd 0
.rodata:0000C518 dd 269h
.rodata:0000C51C dd 15h
.rodata:0000C520 dd 0
.rodata:0000C524 dd 0FFFFC1EDh
.rodata:0000C528 dd 8
.rodata:0000C52C dd 0
.rodata:0000C530 dd 4
.rodata:0000C534 dd offset _2_STRING_10106_0 ; "V$BH"
.rodata:0000C538 dd 4
.rodata:0000C53C dd offset _2_STRING_10103_0 ; "NULL"
.rodata:0000C540 dd 3
.rodata:0000C544 dd 0
.rodata:0000C548 dd 0F5h
.rodata:0000C54C dd 14h
.rodata:0000C550 dd 0
```
By the way, often, while analyzing Oracle RDBMS’s internals, you may ask yourself, why are the names of the functions and global variable so weird.

Probably, because Oracle RDBMS is a very old product and was developed in C in the 1980s.

And that was a time when the C standard guaranteed that the function names/variables can support only up to 6 characters inclusive: «6 significant initial characters in an external identifier».

Probably, the table kqfviw contains most (maybe even all) views prefixed with V$, these are fixed views, present all the time. Superficially, by noticing the cyclic recurrence of data, we can easily see that each kqfviw table element has 12 32-bit fields. It is very simple to create a 12-elements structure in IDA and apply it to all table elements. As of Oracle RDBMS version 11.2, there are 1023 table elements, i.e., in it are described 1023 of all possible fixed views.

We are going to return to this number later.

As we can see, there is not much information in these numbers in the fields. The first field is always equals to the name of the view (without the terminating zero). This is correct for each element. But this information is not very useful.

We also know that the information about all fixed views can be retrieved from a fixed view named V$FIXED_VIEW_DEFINITION (by the way, the information for this view is also taken from the kqfviw and kqfvip tables.) Incidentally, there are 1023 elements in those too. Coincidence? No.

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$VERSION';

VIEW_NAME
-------------------------------
VIEW_DEFINITION
-------------------------------
V$VERSION
select BANNER from GV$VERSION where inst_id = USERENV('Instance')

So, V$VERSION is some kind of a thunk view for another view, named GV$VERSION, which is, in turn:

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$VERSION';

VIEW_NAME
-------------------------------
VIEW_DEFINITION
-------------------------------
GV$VERSION
select inst_id, banner from x$version

The tables prefixed with X$ in the Oracle RDBMS are service tables too, undocumented, cannot be changed by the user and are refreshed dynamically.

If we search for the text

```
select BANNER from GV$VERSION where inst_id = USERENV('Instance')
```

... in the kqf.o file, we find it in the kqfvip table:

```
Listing 8.11: kqf.o
.rodata:0800C554  dd 0FFFC1EEh  .rodata:0800C558  dd 5  .rodata:0800C55C  dd 0
.rodata:080185A0  kqfvip dd offset _2_STRING_11126_0 ; DATA XREF: kqfgvcn+18
.rodata:080185A0  ; kqfgvt+F
.rodata:080185A4  ; "select inst_id,decode(indx,1,'data bloc"
.rodata:080185A8  dd 0
.rodata:080185AC  dd 0
```

The table appear to have 4 fields in each element. By the way, there are 1023 elements in it, again, the number we already know.

The second field points to another table that contains the table fields for this fixed view. As for V$VERSION, this table has only two elements, the first is 6 and the second is the BANNER string (the number 6 is this string’s length) and after, a terminating element that contains 0 and a null C string:

```
Listing 8.12: kqf.o
.rodata:08019570 dd offset _2__STRING_11378_0;
"select BANNER from GV$VERSION where in"
.rodata:08019574 dd offset kqfv133_c_0
.rodata:08019578 dd 0
.rodata:0801957C dd 0
.rodata:08019580 dd offset _2__STRING_11379_0;
"select inst_id,decode(bitand(cfflg,1),0"
.rodata:08019584 dd offset kqfv403_c_0
.rodata:08019588 dd 0
.rodata:0801958C dd 0
.rodata:08019590 dd offset _2__STRING_11380_0;
"select STATUS , NAME, IS_RECOVERY DEST"
.rodata:08019594 dd offset kqfv199_c_0
```

By joining data from both kqfviw and kqfvip tables, we can get the SQL statements which are executed when the user wants to query information from a specific fixed view.

So we can write an oracle tables program, to gather all this information from Oracle RDBMS for Linux’s object files. For V$VERSION, we find this:

```
Listing 8.13: Result of oracle tables
kqfviw_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xffffc085 0x4
kqfvip_element.statement: [select BANNER from GV$VERSION where inst_id = USERENV('Instance')]
kqfvip_element.params: [BANNER]
```

And:

```
Listing 8.14: Result of oracle tables
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0xffffc192 0x1
kqfvip_element.statement: [select inst_id, banner from x$version]
kqfvip_element.params: [INST_ID] [BANNER]
```

The GV$VERSION fixed view is different from V$VERSION only in that it has one more field with the identifier instance.

Anyway, we are going to stick with the X$VERSION table. Just like any other X$-table, it is undocumented, however, we can query it:

```
SQL> select * from x$version;
ADDR  INDX  INST_ID
------- -------- ----------
BANNER
---------------------------
0DBAF574 0 1
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
```
This table has some additional fields, like ADDR and INDX.

While scrolling kqf.o in IDA we can spot another table that contains a pointer to the X$VERSION string, this is kqftab:

Listing 8.15: kqf.o
.
.rodata:0003CAC0 dd 9 ; element number 0x1f6
.rodata:0003C4 dd offset _STRING_13113_0 ; "X$VERSION"
.rodata:0003C8 dd 4
.rodata:0003CA dd offset _STRING_13114_0 ; "kqvt"
.rodata:0003CAD0 dd 4
.rodata:0003CAD4 dd 4
.rodata:0003CAD8 dd 0
.rodata:0003CADC dd 4
.rodata:0003CAE0 dd 0Ch
.rodata:0003CAE4 dd 0FFFC075h
.rodata:0003CAE8 dd 3
.rodata:0003CAEC dd 0
.rodata:0003CAF0 dd 0
.rodata:0003CAF4 dd offset _STRING_13115_0 ; "X$KQFSZ"
.rodata:0003CAF8 dd 5
.rodata:0003C0 dd offset _STRING_13116_0 ; "kqfsz"
.rodata:0003C100 dd 1
.rodata:0003C0B0 dd 38h
.rodata:0003C0B8 dd 0
.rodata:0003C0B0 dd 7
.rodata:0003C0B10 dd 0
.rodata:0003C0B14 dd 0FFFC09Dh
.rodata:0003C0B18 dd 2
.rodata:0003C0B1C dd 0

There are a lot of references to the X$-table names, apparently, to all Oracle RDBMS 11.2 X$-tables. But again, we don’t have enough information.

It’s not clear what does the kqvt string stands for.

The kq prefix may mean kernel or query.

v apparently stands for version and t—type? Hard to say.

A table with a similar name can be found in kqf.o:

Listing 8.16: kqf.o
.
.rodata:0008C360 kqvt_c_0 kqftap_param <4, offset _STRING_19_0, 917h, 0, 0, 0, 4, 0, 0>
.rodata:0008C360 ; DATA XREF: .rodata:00042680
.rodata:0008C360 ; "ADDR"
.rodata:0008C384 kqftap_param <4, offset _STRING_20_0, 0B02h, 0, 0, 4, 0, 0>
.rodata:0008C3A8 kqftap_param <7, offset _STRING_21_0, 0B02h, 0, 0, 4, 0, 0>
.rodata:0008C3CC kqftap_param <6, offset _STRING_5017_0, 601h, 0, 0, 50h, 0, 0>
.rodata:0008C3F0 kqftap_param <0, offset _STRING_0_0, 0, 0, 0, 0, 0, 0>

It contains information about all fields in the X$VERSION table. The only reference to this table is in the kqftab table:

Listing 8.17: kqf.o
.
.rodata:00042680 kqftap_element <0, offset kqvt_c_0, offset kqvrow, 0>
.
\ element 0x1f6

It is interesting that this element here is 0x1f6th (502nd), just like the pointer to the X$VERSION string in the kqftab table.

Probably, the kqftap and kqftab tables complement each other, just like kqfvip and kqfviiw.

We also see a pointer to the kqvrow() function. Finally, we got something useful!

So we will add these tables to our oracle tables utility too. For X$VERSION we get:

39yurichev.com
Listing 8.18: Result of oracle tables

```
<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>kqftab_element.name</td>
<td>[X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0xffffc075 0x3</td>
</tr>
<tr>
<td>kqftap_param.name=ADDR</td>
<td>0x917 0x0 0x4 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0</td>
</tr>
<tr>
<td>kqftap_param.name=INDX</td>
<td>0xb02 0x0 0x0 0x4 0x0 0x4 0x0 0x0 0x0 0x0</td>
</tr>
<tr>
<td>kqftap_param.name=BANNER</td>
<td>0x601 0x0 0x0 0x0 0x0 0x50 0x0 0x0</td>
</tr>
<tr>
<td>kqftap_element.fn1</td>
<td>kqvrow</td>
</tr>
<tr>
<td>kqftap_element.fn2</td>
<td>NULL</td>
</tr>
</tbody>
</table>
```

With the help of tracer, it is easy to check that this function is called 6 times in row (from the qerfxFetch() function) while querying the X$VERSION table.

Let's run tracer in cc mode (it comments each executed instruction):

```
tracer -a:oracle.exe bpf=oracle.exe!_kqvrow,trace:cc
```

```
.kqvrow_  proc near

var_7C   = byte ptr -7Ch
var_18   = dword ptr -18h
var_14   = dword ptr -14h
Dest     = dword ptr -10h
var_C    = dword ptr -0Ch
var_8    = dword ptr -8h
var_4    = dword ptr -4h
arg_8    = dword ptr 10h
arg_C    = dword ptr 14h
arg_14   = dword ptr 1Ch
arg_18   = dword ptr 20h

; FUNCTION CHUNK AT .text1:056C11A0 SIZE 00000049 BYTES

    push ebp
    mov ebp, esp
    sub esp, 7Ch
    mov eax, [ebp+arg_14 ]; [EBP+1Ch]=1
    mov ecx, TlsIndex ; [69AE0B08h]=0
    mov edx, large fs:2Ch
    mov edx, [edx+ecx*4]; [EDX+ECX*4]=0xc98c938
    cmp eax, 2; EAX=1
    mov eax, [ebp+arg_8 ]; [EBP+10h]=0xcdfe554
    jz loc_2CE1288
    mov ecx, [eax]; [EAX]=0..5
    mov [ebp+var_4], edi; EDI=0xc98c938

loc_2CE10F6: ; CODE XREF: _kqvrow_+10A
    _kqvrow_+1A9
    cmp ecx, 5 ; ECX=0..5
    ja loc_56C11C7
    mov edi, [ebp+arg_18 ]; [EBP+20h]=0
    mov [ebp+var_14], edx; EDX=0xc98c938
    mov [ebp+var_B], ebx; EBX=0
    mov ebx, eax; EAX=0xcdfe554
    mov [ebp+var_C], esi; ESI=0xcdfe248

loc_2CE110D: ; CODE XREF: _kqvrow_+29E00E6
    jmp loc_2CE1116
    mov edx, ds:off_628B09C[ecx*4]; [ECX*4+628B09Ch]=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a
    push eax; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a

loc_2CE1116: ; DATA XREF: .rdata:off_628B09C
    push offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
    mov ecx, [ebp+arg_C]; [EBP+14h]=0x8a172b4
    xor edx, edx
    mov esi, [ebp+var_14]; [EBP+14h]=0xc98c938
    push edx; EDX=0
    push edx; EDX=0
    push 50h
    push ecx; ECX=0x8a172b4
```

879
push dword ptr [esi+10494h] ; [ESI+10494h]=0xc98cd58

call _kghalf ; tracing nested maximum level (1) reached, skipping this CALL
mov esi, ds: __imp_vsnnnum ; [59771A8h]=0x61bc49e0
mov [ebp+Dest], eax ; EAX=0xce2ffb0
mov [ebx+8], eax ; EAX=0xce2ffb0
mov [ebx+4], eax ; EAX=0xce2ffb0
mov edi, [esi] ; [ESI]=0xb200100
mov esi, ds: __imp_vsnnstr ; [597D6D4h]=0x65852148, "- Production"
push esi ; ESI=0x65852148, "- Production"
mov esi, [ebp+10h] ; [EBP+10h]=0
push eax ; EAX=1
mov eax, ds: __imp_vsnnban ;

[597D6D8h]=0x65852100, "Oracle Database 11g Enterprise Edition Release %d.%d.%d.%d.%d %s"
push edx ; EDX=0
push ecx ; ECX=2
push ebx ; EBX=0xb
mov eax, [ebp+var_8] ; [EBP+8]=0
leal eax, [eax+4] ; [EAX+4]=0xce2ffb0, "NLSRTL Version 11.2.0.1.0 - Production",

"Oracle Database 11g Enterprise Edition Release %d.%d.%d.%d.%d %s"
add esp, 38h
mov dword ptr [ebx], 1

loc_2CE1192: ; CODE XREF: _kqvrow_+FB
    _kqvrow_+128 ...
test edi, edi ; EDI=0
jnz _VInfreq__kqvrow
mov esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
mov edi, [ebp+var_4] ; [EBP-4]=0xce2ffb0
mov eax, ebx ; EAX=0xcdfe554
mov ebx, [ebp+var_8] ; [EBP-8]=0
lea eax, [eax+4] ; [EAX+4]=0xce2ffb0, "Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production",

"NLSRTL Version 11.2.0.1.0 - Production", "PL/SQL Release 11.2.0.1.0 - Production",

"Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production",

"PL/SQL Release 11.2.0.1.0 - Production",

"TNS for 32-bit Windows: Version 11.2.0.1.0 - Production"

loc_2CE11A8: ; CODE XREF: _kqvrow_+29E00F6
mov esp, ebp
pop ebp
retn ; EAX=0xcdfe558

loc_2CE11AC: ; DATA XREF: .rdata:0628B0A0
mov edx, [ebx+8] ; [EBX+8]=0xce2ffb0, "Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production"
mov dword ptr [ebx], 2
mov [ebx+4], edx ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production"
push edx ; EDX=0xce2ffb0, "Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production"
call _kkxvsn ; tracing nested maximum level (1) reached, skipping this CALL
pop ecx
mov edx, [ebx+4] ; [EBX+4]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
movzx ecx, byte ptr [edx] ; [EDX]=0x50
test ecx, ecx ; ECX=0x50
jnz short loc_2CE1192
mov edx, [ebp+var_14]
mov esi, [ebp+var_C]
mov    eax, ebx
mov    ebx, [ebp+var_8]
mov    ecx, [eax]
jmp    loc_2CE10F6

loc_2CE11D8: ; DATA XREF: .rdata:0628B0A4
push   0
push   50h
mov    edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
mov    [ebp+var_8], edx  ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
push   edx
            ; EDX=0xce2ffb0, "PL/SQL Release 11.2.0.1.0 - Production"
call    _lmxver
            ; tracing nested maximum level (1) reached, skipping this CALL
add     esp, 0Ch
mov    dword ptr [ebx], 3
jmp    short loc_2CE1192

loc_2CE11F6: ; DATA XREF: .rdata:0628B0A8
mov    edx, [ebx+8]    ; [EBX+8]=0xce2ffb0
mov    [ebp+var_18], 50h
mov    [ebx+4], edx  ; EDX=0xce2ffb0
push   0
            ; tracing nested maximum level (1) reached, skipping this CALL
call    _npinli
            ; tracing nested maximum level (1) reached, skipping this CALL
pop    ecx
test   eax, eax
            ; EAX=0
jnz     loc_56C11DA
lea    edx, [ebp+var_7C]    ; [EBP-7Ch]=1
push   edx
            ; EDX=0xd76c8d8
push   0
mov    esi, [ebp+var_C]    ; [EBP-0Ch]=0xcdfe248
mov    edi, [ebp+var_4]    ; [EBP-4]=0x98c938
mov    eax, ebx
            ; EBX=0xcdfe554
mov    ebx, [ebp+var_8]
            ; [EBP-8]=0
mov    ecx, 5
jmp    loc_2CE110F6

loc_2CE122B: ; CODE XREF: _kqvrow +29E0118
mov    dword ptr [ebx], 4
jmp    loc_2CE1192

loc_2CE1236: ; DATA XREF: .rdata:0628B0AC
lea    edx, [ebp+var_7C]    ; [EBP-7Ch]=1
push   edx
            ; EDX=0xd76c93c
push   0
mov    esi, [ebp+var_8]    ; [EBP+8]=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 - Production"
mov    [ebp+var_14], esi  ; ESI=0xce2ffb0, "TNS for 32-bit Windows: Version 11.2.0.1.0 - Production"
mov    [ebx+var_18], ecx  ; ECX=0x50
            ; "TNS for 32-bit Windows: Version 11.2.0.1.0 - Production"
            ; tracing nested maximum level (1) reached, skipping this CALL
add     esp, 0Ch
mov    edx, [ebp+var_14]  ; [EBP-14h]=0xc98c938
mov    [ebp+var_18], ecx  ; [EBP-18h]=0x50
mov    eax, ebx
            ; EBX=0xcdfe554
mov    ebx, [ebp+var_8]  ; [EBP-8]=0
mov    ecx, 5
jmp    loc_2CE110F6

loc_2CE127A: ; DATA XREF: .rdata:0628B0B0
mov    edx, [ebp+var_14]  ; [EBP-14h]=0xc98c938
mov    esi, [ebp+var_C]  ; [EBP-0Ch]=0xcdfe248
mov    edi, [ebp+var_4]  ; [EBP-4]=0xc98c938
mov    eax, ebx
            ; EBX=0xcdfe554
mov    ebx, [ebp+var_8]  ; [EBP-8]=0

loc_2CE1288: ; CODE XREF: _kqvrow +1F
Now it is easy to see that the row number is passed from outside. The function returns the string, constructing it as follows:

<table>
<thead>
<tr>
<th>String</th>
<th>Using vsnstr, vsnum, vsnban global variables.</th>
<th>Calls sprintf().</th>
</tr>
</thead>
<tbody>
<tr>
<td>String 2</td>
<td>Calls kkxvsn().</td>
<td></td>
</tr>
<tr>
<td>String 3</td>
<td>Calls lmxver().</td>
<td></td>
</tr>
<tr>
<td>String 4</td>
<td>Calls npinli(), nrtnsvrs().</td>
<td></td>
</tr>
<tr>
<td>String 5</td>
<td>Calls lxvers().</td>
<td></td>
</tr>
</tbody>
</table>

That's how the corresponding functions are called for determining each module's version.

### 8.13.2 X$KSMLRU table in Oracle RDBMS

There is a mention of a special table in the *Diagnosing and Resolving Error ORA-04031 on the Shared Pool or Other Memory Pools [Video] [ID 146599.1]* note:

There is a fixed table called X$KSMLRU that tracks allocations in the shared pool that cause other objects in the shared pool to be aged out. This fixed table can be used to identify what is causing the large allocation.

If many objects are being periodically flushed from the shared pool then this will cause response time problems and will likely cause library cache latch contention problems when the objects are reloaded into the shared pool.

One unusual thing about the X$KSMLRU fixed table is that the contents of the fixed table are erased whenever someone selects from the fixed table. This is done since the fixed table stores only the largest allocations that have occurred. The values are reset after being selected so that subsequent large allocations can be noted even if they were not quite as large as others that occurred previously. Because of this resetting, the output of selecting from this table should be carefully kept since it cannot be retrieved back after the query is issued.

However, as it can be easily checked, the contents of this table are cleared each time it's queried. Are we able to find why? Let's get back to tables we already know: kqftab and kqftap which were generated with oracle tables' help, that has all information about the X$-tables. We can see here that the ksmlrs() function is called to prepare this table’s elements:

#### Listing 8.19: Result of oracle tables

```plaintext
kqftab_element.name: [X$KSMLRU] ? [ksmlr] 0x4 0x64 0x11 0xc 0xffffc0bb 0x5
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRIDX] ?: 0xb02 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRSHRPOOL] ?: 0xb02 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRSIZ] ?: 0x2 0x0 0x0 0x0 0x4 0x20 0x0
```

---

882

---

Yurichev.com
Indeed, with tracer's help it is easy to see that this function is called each time we query the X$KSMLRU table.

Here we see a references to the ksmsplu_sp() and ksmsplu_jp() functions, each of them calls the ksmsplu() at the end. At the end of the ksmsplu() function we see a call to memset():

Listing 8.20: ksm.o

...  
.text:00434C50 loc_434C50: ; DATA XREF: .rdata:off_5E50E8A8  
.text:00434C50 mov edx, [ebp-4]  
.text:00434C53 mov [eax], [esi]  
.text:00434C55 mov [edi], [esi]  
.text:00434C57 mov [eax+4], [esi]  
.text:00434C5A mov [edi], eax  
.text:00434C5C add edx, 1  
.text:00434C5F mov [ebp-4], edx  
.text:00434C62 jnz loc_434B7D  
.text:00434C68 mov ecx, [ebp+14h]  
.text:00434C6B mov [ebx], [ebp-10h]  
.text:00434C6E mov [esi], [ebp-0Ch]  
.text:00434C71 mov edi, [ebp-8]  
.text:00434C74 lea eax, [ecx+8Ch]  
.text:00434C79 push 370h ; Size  
.text:00434C7F push 0 ; Val  
.text:00434C8B push eax ; Dst  
.text:00434C82 call __intel_fast_memset  
.text:00434C87 add esp, 0Ch  
.text:00434C8A mov esp, ebp  
.text:00434C8C pop ebp  
.text:00434C8D retn  
.text:00434C8D _ksmsplu endp

Constructions like memset (block, 0, size) are often used just to zero memory block. What if we take a risk, block the memset() call and see what happens?

Let's run tracer with the following options: set breakpoint at 0x434C7A (the point where the arguments to memset() are to be passed), so that tracer will set program counter EIP to the point where the arguments passed to memset() are to be cleared (at 0x434C8A) It can be said that we just simulate an unconditional jump from address 0x434C7A to 0x434C8A.

tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A,set(eip,0x00434C8A)

(Important: all these addresses are valid only for the win32 version of Oracle RDBMS 11.2)

Indeed, now we can query the X$KSMLRU table as many times as we want and it is not being cleared anymore!

Just in case, do not try this on your production servers.

It is probably not a very useful or desired system behavior, but as an experiment for locating a piece of code that we need, it perfectly suits our needs!

8.13.3 V$TIMER table in Oracle RDBMS

V$TIMER is another fixed view that reflects a rapidly changing value:
V$TIMER displays the elapsed time in hundredths of a second. Time is measured since the beginning of the epoch, which is operating system specific, and wraps around to 0 again whenever the value overflows four bytes (roughly 497 days).

(From Oracle RDBMS documentation)

It is interesting that the periods are different for Oracle for win32 and for Linux. Will we be able to find the function that generates this value?

As we can see, this information is finally taken from the X$KSUTM table.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$TIMER';

VIEW_NAME  ------------------------------
VIEW_DEFINITION  ------------------------------
V$TIMER         select HSECS from GV$TIMER where inst_id = USERENV('Instance')

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='GV$TIMER';

VIEW_NAME  ------------------------------
VIEW_DEFINITION  ------------------------------
GV$TIMER      select inst_id,ksutmtim from x$ksutm
```

Now we are stuck in a small problem, there are no references to value generating function(s) in the tables kqftab/kqftap:

```
Listing 8.21: Result of oracle tables

kqftab_element.name: [X$KSUTM] ?: [ksutm] 0x1 0x4 0x4 0x0 0xffffc09b 0x3
kqftap_param.name=[ADDR] ?: 0x10917 0x0 0x0 0x4 0x0 0x0 0x0 0x0
kqftap_param.name=[INDEX] ?: 0x20b02 0x0 0x0 0x4 0x0 0x0 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0 0x0
kqftap_param.name=[KSUTMTIM] ?: 0x1302 0x0 0x0 0x0 0x4 0x0 0x1e
kqftap_element.fn1=NULL
kqftap_element.fn2=NULL
```

When we try to find the string KSUTMTIM, we see it in this function:

```
kqfd_DRN_ksutm_c proc near     ; DATA XREF: .rodata:0805B4E8
arg_0  = dword ptr 8
arg_B  = dword ptr 10h
arg_C  = dword ptr 14h
    push    ebp
    mov     ebp, esp
    push    [ebp+arg_C]
    push    offset ksutm
    push    offset _2_STRING_1263_0 ; "KSUTMTIM"
    push    [ebp+arg_B]
    push    [ebp+arg_0]
    call    kqfd_cfui_drain
    add     esp, 14h
    mov     esp, ebp
    pop     ebp
    retn
kqfd_DRN_ksutm_c endp
```

[^41]: [http://go.yurichev.com/17088](http://go.yurichev.com/17088)
The kqfd_DRN_ksutm_c() function is mentioned in the kqfd_tab_registry_0 table:

```assembly
dd offset _2__STRING_62_0 ; "X$KSUTM"
dd offset kqfd_OPN_ksutm_c
dd offset kqfd_tabl_fetch
dd 0
```

There is a function ksugtm() referenced here. Let's see what's in it (Linux x86):

```
ksugtm proc near
var_1C = byte ptr -1Ch
arg_4 = dword ptr 0Ch

    push ebp
    mov ebp, esp
    sub esp, 1Ch
    lea eax, [ebp+var_1C]
    push eax
    call slgcs
    pop ecx
    mov edx, [ebp+arg_4]
    mov [edx], eax
    mov eax, 4
    mov esp, ebp
    pop ebp
    ret
ksugtm endp
```

The code in the win32 version is almost the same. Is this the function we are looking for? Let's see:

```
tracer -a:oracle.exe bpf=oracle.exe!_ksugtm,args:2,dump_args:0x4
```

Let's try again:

```
SQL> select * from V$TIMER;
HSECS  
--------
 27294929
SQL> select * from V$TIMER;
HSECS  
--------
 27295006
SQL> select * from V$TIMER;
HSECS  
--------
 27295167
```

```
Listing 8.23: tracer output
TID=2428](0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch/\ 
+0xfd (0x5566d5))
Argument 2/2
0D76C5F0: 38 C9 "8. "
TID=2428](0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: D1 7C A0 01 ",. ." 
TID=2428](0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from oracle.exe!__VInfreq__qerfxFetch/\ 
+0xfd (0x5566d5))
```

885
Indeed—the value is the same we see in SQL*Plus and it is returned via the second argument.

Let’s see what is in slgcs() (Linux x86):

```
slgcs proc near

var_4 = dword ptr -4
arg_0 = dword ptr 8

    push    ebp
    mov     ebp, esp
    push    esi
    mov     [ebp+var_4], ebx
    mov     eax, [ebp+arg_0]
    call    $+5
    pop     ebx
    nop      ; PIC mode
    mov     ebx, offset _GLOBAL_OFFSET_TABLE_
    mov     dword ptr [eax], 0
    call    sltrgatime64       ; PIC mode
    push    0
    push    0Ah
    push    edx
    push    eax
    call    __udivdi3          ; PIC mode
    mov     ebx, [ebp+var_4]
    add     esp, 10h
    mov     esp, ebp
    pop     ebp

slgcs endp
```

(it is just a call to sltrgatime64())

and division of its result by 10 (3.12 on page 496))

And win32-version:

```
_slgcs proc near    ; CODE XREF: dbgefgHtElResetCount+15

    db       66h
    nop
    push    ebp
    mov     ebp, esp
    mov     eax, [ebp+8]
    mov     dword ptr [eax], 0
    call    ds:__imp__GetTickCount@0  ; GetTickCount()
    mov     edx, eax
    mov     eax, 0CCCCCCCCDh
    mul     edx
    shr     edx, 3
    mov     eax, edx
    mov     esp, ebp
    pop     ebp

_slgcs endp
```
It is just the result of `GetTickCount()` divided by 10 (3.12 on page 496).

Voilà! That’s why the win32 version and the Linux x86 version show different results, because they are generated by different OS functions.

`Drain` apparently implies connecting a specific table column to a specific function.

We will add support of the table `kqfd_tab_registery_0` to oracle tables, now we can see how the table column’s variables are connected to a specific functions:

```
[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksutm_c]
[X$KSUGIF] [kqfd_OPN_ksusg_c] [kqfd_tabl_fetch] [NULL] [NULL] [kqfd_DRN_ksusg_c]
```

OPN, apparently stands for, open, and DRN, apparently, for drain.

### 8.14 Handwritten assembly code

#### 8.14.1 EICAR test file

This .COM-file is intended for testing antivirus software, it is possible to run in MS-DOS and it prints this string: “EICAR-STANDARD-ANTIVIRUS-TEST-FILE!

Its important property is that it consists entirely of printable ASCII-symbols, which, in turn, makes it possible to create it in any text editor:

```
X5O!P%@AP[4&PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Let’s decompile it:

```assembly
; initial conditions: SP=0FFFEh, SS:[SP]=0
0100 58      pop    ax
0101 35 4F 21 xor     ax, 214Fh
; AX = 214Fh and SP = 0
0104 50      push   ax
0105 25 40 41 and    ax, 4140h
; AX = 140h and SP = FFFEh and SS:[FFE] = 214Fh
0108 50      push   ax
; AX = 140h, SP = FFFCh and SS:[FFFC] = 140h and SS:[FFE] = 214Fh
0109 5B      pop     bx
; AX = 140h, BX = 140h, SP = FFFEh and SS:[FFE] = 214Fh
010A 34 5C xor     al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFEh and SS:[FFE] = 214Fh
010C 50      push    ax
010D 5A      pop      dx
; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFEh and SS:[FFE] = 214Fh
010E 58      push     ax
; AX = 140h, BX = 140h, DX = 11Ch and SP = 0
010F 35 34 2B xor     ax, 2834h
; AX = 97Bh, BX = 140h, DX = 11Ch and SP = 0
0112 50      push     ax
0113 5E      pop      si
; AX = 97Bh, BX = 140h, DX = 11Ch, SI = 97Bh and SP = 0
0114 29 37  sub     [bx], si
0116 43      inc     bx
0117 43      inc     bx
0118 29 37  sub     [bx], si
0119 70 24  jge     short near ptr word_10140
011C 45 49 43 ... db 'EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$
0140 48 2B  word_10140 dw 2B48h ; CD 21 (INT 21) will be here
0142 48 2A  dw 2A48h ; CD 20 (INT 20) will be here
0144 0D      db 0Dh
0145 0A      db 0Ah
```

We will add comments about the registers and stack after each instruction.

Essentially, all these instructions are here only to execute this code:

---

42MSDN
43yurichev.com

887
INT 21h with 9th function (passed in AH) just prints a string, the address of which is passed in DS:DX. By the way, the string has to be terminated with the ‘$’ sign. Apparently, it’s inherited from CP/M and this function was left in DOS for compatibility. INT 20h exits to DOS.

But as we can see, these instruction’s opcodes are not strictly printable. So the main part of EICAR file is:

- preparing the register (AH and DX) values that we need;
- preparing INT 21 and INT 20 opcodes in memory;
- executing INT 21 and INT 20.

By the way, this technique is widely used in shellcode construction, when one have to pass x86 code in string form.

Here is also a list of all x86 instructions which have printable opcodes: .1.6 on page 1023.

### 8.15 Demos

Demos (or demomaking) were an excellent exercise in mathematics, computer graphics programming and very tight x86 hand coding.

#### 8.15.1 10 PRINT CHR$(205.5+RND(1));: GOTO 10

All examples here are MS-DOS .COM files.

In [Nick Montfort et al, 10 PRINT CHR$(205.5+RND(1));: GOTO 10, (The MIT Press:2012)](http://go.yurichev.com/17286) we can read about one of the most simple possible random maze generators.

It just prints a slash or backslash characters randomly and endlessly, resulting in something like this:

There are a few known implementations for 16-bit x86.

**Trixter’s 42 byte version**

The listing was taken from his website[^1], but the comments are mine.

[^1]: Also available as [http://go.yurichev.com/17286](http://go.yurichev.com/17286)
[^2]: [http://go.yurichev.com/17305](http://go.yurichev.com/17305)
The pseudo-random value here is in fact the time that has passed from the system’s boot, taken from the 8253 time chip, the value increases by one 18.2 times per second.

By writing zero to port 43h, we send the command “select counter 0”, “counter latch”, ”binary counter” (not a BCD value).

The interrupts are enabled back with the POPF instruction, which restores the IF flag as well.

It is not possible to use the IN instruction with registers other than AL, hence the shuffling.

My attempt to reduce Trixter’s version: 27 bytes

We can say that since we use the timer not to get a precise time value, but a pseudo-random one, we do not need to spend time (and code) to disable the interrupts.

Another thing we can say is that we need only one bit from the low 8-bit part, so let’s read only it.

We can reduced the code slightly and we’ve got 27 bytes:

```
00000000: B9D007 mov cx,007D0 ; limit output to 2000 characters
00000003: 31C0 xor ax,ax ; command to timer chip
00000005: E643 out 043,al
00000007: E440 in al,040 ; read 8-bit of timer
00000009: D1E8 shr ax,1 ; get second bit to CF flag
0000000B: D1E8 shr ax,1
0000000D: B05C mov al,05C ;'`
00000011: B02F mov al,02F ;'/'
00000013: B40E mov ah,00E
00000015: CD10 int 010
00000017: E2E1 loop 00000009 ; loop 2000 times
00000019: CD20 int 020 ; exit to DOS
```

Taking random memory garbage as a source of randomness

Since it is MS-DOS, there is no memory protection at all, we can read from whatever address we want. Even more than that: a simple LODSB instruction reads a byte from the DS:SI address, but it’s not a
problem if the registers' values are not set up, let it read 1) random bytes; 2) from a random place in memory!

It is suggested in Trixter's webpage\(^{46}\) to use LODSB without any setup.

It is also suggested that the SCASB instruction can be used instead, because it sets a flag according to the byte it reads.

Another idea to minimize the code is to use the INT 29h DOS syscall, which just prints the character stored in the AL register.

That is what Peter Ferrie did \(^{47}\):

<table>
<thead>
<tr>
<th>Listing 8.24: Peter Ferrie: 10 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>; AL is random at this point</td>
</tr>
<tr>
<td>00000000: AE scasb</td>
</tr>
<tr>
<td>; CF is set according subtracting random memory byte from AL.</td>
</tr>
<tr>
<td>; so it is somewhat random at this point</td>
</tr>
<tr>
<td>00000001: D6 setalc</td>
</tr>
<tr>
<td>; AL is set to 0xFF if CF=1 or to 0 if otherwise</td>
</tr>
<tr>
<td>00000002: 242D and al,02D ;'-'</td>
</tr>
<tr>
<td>; AL here is 0x2D or 0</td>
</tr>
<tr>
<td>00000004: 042F add al,02F ;'//'</td>
</tr>
<tr>
<td>; AL here is 0x5C or 0x2F</td>
</tr>
<tr>
<td>00000006: CD29 int 029 ; output AL to screen</td>
</tr>
<tr>
<td>00000008: EBF6 jmps 000000000 ; loop endlessly</td>
</tr>
</tbody>
</table>

So it is possible to get rid of conditional jumps at all. The ASCII code of backslash ("\") is 0x5C and 0x2F for slash ("/"). So we have to convert one (pseudo-random) bit in the CF flag to a value of 0x5C or 0x2F.

This is done easily: by AND-ing all bits in AL (where all 8 bits are set or cleared) with 0x2D we have just 0 or 0x2D.

By adding 0x2F to this value, we get 0x5C or 0x2F.

Then we just output it to the screen.

**Conclusion**

It is also worth mentioning that the result may be different in DOSBox, Windows NT and even MS-DOS, due to different conditions: the timer chip can be emulated differently and the initial register contents may be different as well.

---

\(^{46}\) [http://go.yurichev.com/17305](http://go.yurichev.com/17305)

\(^{47}\) [http://go.yurichev.com/17087](http://go.yurichev.com/17087)
Mandelbrot set is a fractal, which exhibits self-similarity. When you increase scale, you see that this characteristic pattern repeating infinitely.

Here is a demo written by “Sir_Lagsalot” in 2009, that draws the Mandelbrot set, which is just a x86 program with executable file size of only 64 bytes. There are only 30 16-bit x86 instructions.

Here it is what it draws:

Let’s try to understand how it works.

**Theory**

**A word about complex numbers**

A complex number is a number that consists of two parts—real (Re) and imaginary (Im).

The complex plane is a two-dimensional plane where any complex number can be placed: the real part is one coordinate and the imaginary part is the other.

Some basic rules we have to keep in mind:

- **Addition:** \((a + bi) + (c + di) = (a + c) + (b + d)i\)
  
  In other words:
  
  \[
  \text{Re}(\text{sum}) = \text{Re}(a) + \text{Re}(b)
  \]

---

48Download it [here](#).
\[ \text{Im}(\text{sum}) = \text{Im}(a) + \text{Im}(b) \]

- **Multiplication:** \((a + bi)(c + di) = (ac - bd) + (bc + ad)i\)
  
  In other words:
  \[
  \begin{align*}
  \text{Re}(\text{product}) &= \text{Re}(a) \cdot \text{Re}(c) - \text{Re}(b) \cdot \text{Re}(d) \\
  \text{Im}(\text{product}) &= \text{Im}(b) \cdot \text{Im}(c) + \text{Im}(a) \cdot \text{Im}(d)
  \end{align*}
  \]

- **Square:** \((a + bi)^2 = (a + bi)(a + bi) = (a^2 - b^2) + (2ab)i\)
  
  In other words:
  \[
  \begin{align*}
  \text{Re}(\text{square}) &= \text{Re}(a)^2 - \text{Im}(a)^2 \\
  \text{Im}(\text{square}) &= 2 \cdot \text{Re}(a) \cdot \text{Im}(a)
  \end{align*}
  \]

### How to draw the Mandelbrot set

The Mandelbrot set is a set of points for which the \(z_{n+1} = z_n^2 + c\) recursive sequence (where \(z\) and \(c\) are complex numbers and \(c\) is the starting value) does not approach infinity.

In plain English language:

- Enumerate all points on screen.
- Check if the specific point is in the Mandelbrot set.
  
  Here is how to check it:
  
  - Represent the point as a complex number.
  - Calculate the square of it.
  - Add the starting value of the point to it.
  - Does it go off limits? If yes, break.
  - Move the point to the new place at the coordinates we just calculated.
  - Repeat all this for some reasonable number of iterations.
  
  - The point is still in limits? Then draw the point.
  - The point has eventually gone off limits?
    
    - (For a black-white image) do not draw anything.
    - (For a colored image) transform the number of iterations to some color. So the color shows the speed with which point has gone off limits.

Here is Pythonesque algorithm for both complex and integer number representations:

```python
# Listing 8.25: For complex numbers

def check_if_is_in_set(P):
    P_start=P
    iterations=0
    while True:
        if (P>bounds):
            break
        P=P^2+P_start
        if iterations > max_iterations:
            break
        iterations++
    return iterations

# black-white
for each point on screen P:
    if check_if_is_in_set (P) < max_iterations:
        draw point

# colored
for each point on screen P:
```
The integer version is where the operations on complex numbers are replaced with integer operations according to the rules which were explained above.

Listing 8.26: For integer numbers

```python
def check_if_is_in_set(X, Y):
    X_start=X
    Y_start=Y
    iterations=0
    while True:
        if (X^2 + Y^2 > bounds):
            break
        new_X=X^2 - Y^2 + X_start
        new_Y=2*X*Y + Y_start
        if iterations > max_iterations:
            break
        iterations++
    return iterations

# black-white
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        if check_if_is_in_set (X,Y) < max_iterations:
            draw point at X, Y

# colored
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        iterations = if check_if_is_in_set (X,Y)
        map iterations to color
        draw color point at X,Y
```

Here is also a C# source which is present in the Wikipedia article\(^{49}\), but we’ll modify it so it will print the iteration numbers instead of some symbol \(^{50}\):

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Mnoj
{
    class Program
    {
        static void Main(string[] args)
        {
            double realCoord, imagCoord;
            double realTemp, imagTemp, realTemp2, arg;
            int iterations;
            for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
            {
                for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
                {
                    iterations = 0;
                    realTemp = realCoord;
                    imagTemp = imagCoord;
                    arg = (realCoord * realCoord) + (imagCoord * imagCoord);
                    while ((arg < 2*2) && (iterations < 40))
                    {
                        realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp) - realCoord;
                        imagTemp = (2 * realTemp * imagTemp) - imagCoord;
                    }
                }
            }
        }
    }
}
```

\(^{49}\)wikipedia

\(^{50}\)Here is also the executable file: beginners.re
Here is the resulting file, which is too wide to be included here:

`beginners.re`

The maximal number of iterations is 40, so when you see 40 in this dump, it means that this point has been wandering for 40 iterations but never got off limits.

A number \( n \) less than 40 means that point remained inside the bounds only for \( n \) iterations, then it went outside them.
There is a cool demo available at http://go.yurichev.com/17309, which shows visually how the point moves on the plane at each iteration for some specific point. Here are two screenshots.

First, we’ve clicked inside the yellow area and saw that the trajectory (green line) eventually swirls at some point inside:

Figure 8.18: Click inside yellow area

This implies that the point we’ve clicked belongs to the Mandelbrot set.
Then we've clicked outside the yellow area and saw a much more chaotic point movement, which quickly went off bounds:

![Figure 8.19: Click outside yellow area](image)

This means the point doesn't belong to Mandelbrot set.
Another good demo is available here: [http://go.yurichev.com/17310](http://go.yurichev.com/17310).
Let's get back to the demo

The demo, although very tiny (just 64 bytes or 30 instructions), implements the common algorithm described here, but using some coding tricks.

The source code is easily downloadable, so here it is, but let's also add comments:

Listing 8.27: Commented source code

```assembly
1 ; X is column on screen
2 ; Y is row on screen
3
4 ; X=0, Y=0 X=319, Y=0
5 ; +-------------------------->
6 |
7 |
8 |
9 |
10 |
11 |
12 |
13 ; Y
14 ; X=0, Y=199 X=319, Y=199
15
16 ; switch to VGA 320*200*256 graphics mode
17 mov al,13h
18 int 10h
19 ; initial BX is 0
20 ; initial DI is 0xFFFE
21 ; DS:BX (or DS:0) is pointing to Program Segment Prefix at this moment
22 ; ... first 4 bytes of which are CD 20 FF 9F
23 les ax,[bx]
24 ; ES:AX=9FFF:20CD
25
26 FillLoop:
27 ; set DX to 0. CWD works as: DX:AX = sign_extend(AX).
28 ; AX here 0x20CD (at startup) or less then 320 (when getting back after loop),
29 ; so DX will always be 0.
30 cwd
31 mov ax,di
32 ; AX is current pointer within VGA buffer
33 ; divide current pointer by 320
34 mov cx,320
35 div cx
36 ; DX (start_X) - remainder (column: 0..319); AX - result (row: 0..199)
37 sub ax,100
38 ; AX=AX-100, so AX (start_Y) now is in range -100..99
39 ; DX is in range 0..319 or 0x0000..0x013F
40 dec dh
41 ; DX now is in range 0xFF00..0x003F (-256..63)
42 xor bx,bx
43 xor si,si
44 ; BX (temp_X)=0; SI (temp_Y)=0
45
46 ; get maximal number of iterations
47 ; CX is still 320 here, so this is also maximal number of iteration
48 MandelLoop:
49 mov bp,si ; BP = temp_Y
50 imul si,bx ; SI = temp_X*temp_Y
51 add si,si ; SI = SI*2 = (temp_X*temp_Y)*2
52 imul bx,bx ; BX = BX^2 = temp_X^2
53 jo MandelBreak ; overflow?
54 imul bp,bp ; BP = BP^2 = temp_Y^2
55 jo MandelBreak ; overflow?
56 add bx,bp ; BX = BX+BP = temp_X^2 + temp_Y^2
57 jo MandelBreak ; overflow?
58 sub bx,bp ; BX = BX-BP = temp_X^2 + temp_Y^2 - temp_Y^2 = temp_X^2
59 sub bx,bp ; BX = BX-BP = temp_X^2 - temp_Y^2
60
61
62```
; correct scale:
63
sar bx,6 ; BX=BX/64
add bx,dx ; BX=BX+start_X
66
now temp_X = temp_X^2 - temp_Y^2 + start_X
sar si,6 ; SI=SI/64
add si,ax ; SI=SI+start_Y
69
now temp_Y = (temp_X*temp_Y)*2 + start_Y

loop MandelLoop
MandelBreak:
CX=iterations
xchg ax,cx
AX=iterations, store AL to VGA buffer at ES:[DI]
stosb ; stosb also increments DI, so DI now points to the next point in VGA buffer
; jump always, so this is eternal loop here
jmp FillLoop

Algorithm:
- Switch to 320*200 VGA video mode, 256 colors. \(320 \times 200 = 64000\) (0xFA00).
  Each pixel is encoded by one byte, so the buffer size is 0xFA00 bytes. It is addressed using the ES:DI registers pair.
  ES must be 0xA000 here, because this is the segment address of the VGA video buffer, but storing 0xA000 to ES requires at least 4 bytes (PUSH 0A000h / POP ES). You can read more about the 16-bit MS-DOS memory model here: 11.6 on page 986.
  Assuming that BX is zero here, and the Program Segment Prefix is at the zeroth address, the 2-byte LES AX,[BX] instruction stores 0x20CD to AX and 0x9FFF to ES.
  So the program starts to draw 16 pixels (or bytes) before the actual video buffer. But this is MS-DOS, there is no memory protection, so a write happens into the very end of conventional memory, and usually, there is nothing important. That’s why you see a red strip 16 pixels wide at the right side. The whole picture is shifted left by 16 pixels. This is the price of saving 2 bytes.
- An infinite loop processes each pixel.
  Probably, the most common way to enumerate all pixels on the screen is with two loops: one for the X coordinate, another for the Y coordinate. But then you’ll need to multiply the coordinates to address a byte in the VGA video buffer.
  The author of this demo decided to do it otherwise: enumerate all bytes in the video buffer by using one single loop instead of two, and get the coordinates of the current point using division. The resulting coordinates are: X in the range of \(-256..63\) and Y in the range of \(-100..99\). You can see on the screenshot that the picture is somewhat shifted to the right part of screen.
  That’s because the biggest heart-shaped black hole usually appears on coordinates 0,0 and these are shifted here to right. Could the author just subtract 160 from the value to get X in the range of \(-160..159\)? Yes, but the instruction SUB DX, 160 takes 4 bytes, while DEC DH—2 bytes (which subtracts 0x100 (256) from DX). So the whole picture is shifted for the cost of another 2 bytes of saved space.
    - Check, if the current point is inside the Mandelbrot set. The algorithm is the one that has been described here.
    - The loop is organized using the LOOP instruction, which uses the CX register as counter.
      The author could set the number of iterations to some specific number, but he didn’t: 320 is already present in CX (has been set at line 35), and this is good maximal iteration number anyway. We save here some space by not the reloading CX register with another value.
    - IMUL is used here instead of MUL, because we work with signed values: keep in mind that the 0,0 coordinates has to be somewhere near the center of the screen.
      It’s the same with SAR (arithmetic shift for signed values): it’s used instead of SHR.
    - Another idea is to simplify the bounds check. We must check a coordinate pair, i.e., two variables.
      What the author does is to checks thrice for overflow: two squaring operations and one addition.
Indeed, we use 16-bit registers, which hold signed values in the range of -32768..32767, so if any of the coordinates is greater than 32767 during the signed multiplication, this point is definitely out of bounds: we jump to the MandelBreak label.

- There is also a division by 64 (SAR instruction). 64 sets scale.

Try to increase the value and you can get a closer look, or to decrease if for a more distant look.

- We are at the MandelBreak label, there are two ways of getting here: the loop ended with CX=0 (the point is inside the Mandelbrot set); or because an overflow has happened (CX still holds some value). Now we write the low 8-bit part of CX (CL) to the video buffer.

The default palette is rough, nevertheless, 0 is black: hence we see black holes in the places where the points are in the Mandelbrot set. The palette can be initialized at the program’s start, but keep in mind, this is only a 64 bytes program!

- The program runs in an infinite loop, because an additional check where to stop, or any user interface will result in additional instructions.

Some other optimization tricks:

- The 1-byte CWD is used here for clearing DX instead of the 2-byte XOR DX, DX or even the 3-byte MOV DX, 0.

- The 1-byte XCHG AX, CX is used instead of the 2-byte MOV AX, CX. The current value of AX is not needed here anyway.

- DI (position in video buffer) is not initialized, and it is 0xFFFE at the start.

That’s OK, because the program works for all DI in the range of 0..0xFFFF eternally, and the user can’t notice that it is started off the screen (the last pixel of a 320*200 video buffer is at address 0xF9FF). So some work is actually done off the limits of the screen.

Otherwise, you’ll need an additional instructions to set DI to 0 and check for the video buffer’s end.

My “fixed” version

Listing 8.28: My “fixed” version

```assembly
org 100h
mov al,13h
int 10h

; set palette
mov dx, 3c8h
mov al, 0
out dx, al
mov cx, 100h
inc dx
l00:

mov al, cl
shl ax, 2
out dx, al ; red
out dx, al ; green
out dx, al ; blue
loop l00

push 0a000h
pop es
xor di, di

FillLoop:
cwd
mov ax,di
mov cx,320
div cx
sub ax,100
sub dx,160
xor bx,bx
```

51More information about initial register values: [http://go.yurichev.com/17004](http://go.yurichev.com/17004)
xor si,si
MandelLoop:
    mov bp,si
    imul si,bx
    add si,si
    imul bx,bx
    jo MandelBreak
    imul bp,bp
    jo MandelBreak
    add bx,bp
    jo MandelBreak
    sub bx,bp
    sub bx,bp
    sar bx,6
    add bx,dx
    sar si,6
    add si,ax
loop MandelLoop
MandelBreak:
    xchg ax,cx
    stosb
    cmp di, 0FA00h
    jb FillLoop
; wait for keypress
xor ax,ax
; set text video mode
mov ax, 3
; exit
int 16h
int 10h
int 20h

The author of these lines made an attempt to fix all these oddities: now the palette is smooth grayscale, the video buffer is at the correct place (lines 19..20), the picture is drawn on center of the screen (line 30), the program eventually ends and waits for the user’s keypress (lines 58..68).

But now it’s much bigger: 105 bytes (or 54 instructions) \(^{52}\).

\(^{52}\) You can experiment by yourself: get DosBox and NASM and compile it as: nasm file.asm -fbin -o file.com
8.16 A nasty bug in MSVCRT.DLL

This is the bug that costed me several hours of debugging.

In 2013 I was using MinGW, my C project seems to be very unstable and I saw the “Invalid parameter passed to C runtime function.” error message in debugger.

The error message was also visible using Sysinternals DebugView. And my project has no such error messages or strings. So I started to search it in the whole Windows and found in MSVCRT.DLL file. (Needless to say I was using Windows 7.)

So here it is, the error message in MSVCRT.DLL file supplied with Windows 7:

```
.text:6FFB69D0 OutputString   db 'Invalid parameter passed to C runtime function.','.0Ah,0
                         ; DATA XREF: sub_6FFB6930+83
.text:6FFB69D0
.text:6FFB6930
.text:6FFB6930 sub_6FFB6930 proc near   ; CODE XREF: _wfindfirst64+203FC
    ; sub_6FF62563+319AD
.text:6FFB6930 var_200       = dword ptr -200h
.text:6FFB6930 var_244       = word ptr -244h
.text:6FFB6930 var_240       = word ptr -240h
.text:6FFB6930 var_23C       = word ptr -23Ch
.text:6FFB6930 var_238       = word ptr -238h
.text:6FFB6930 var_234       = dword ptr -234h
.text:6FFB6930 var_230       = dword ptr -230h
```

Where it is referenced?
The string it reported into debugger or DebugView utility using the standard return address=0x75b6ef3c (KERNEL32.dll!BaseThreadInitThunk+0x12), arguments in stack: 0x12ff94
return address=0x4010f8 (1.exe!OEP+0xe3), arguments in stack: 0x12ff88, 0x4010f8, 0x1, 0x2e0ea8
return address=0x401010 (1.exe!.text+0x10), arguments in stack: 0x12ff14, 0x401010, 0x403010("
Call stack:
(0) msvcrt.dll!0x6ffb6930() (called from msvcrt.dll!_ftol2_sse_except+0x1b467 (0x759ed222))
PID=3560|New process 1.exe
...
I found that my code was calling `stricmp()` function with NULL as one argument. In fact, I made up this example when writing this:

```
#include <stdio.h>
#include <string.h>

int main()
{
    stricmp("asd", NULL);
}
```

If this piece of code is compiled using old MinGW or old MSVC 6.0, it is linked against MSVCRT.DLL file. Which, as of Windows 7, silently sends the “Invalid parameter passed to C runtime function.” error message to the debugger and then does nothing!

Let's see how `stricmp()` is implemented in MSVCRT.DLL:

```
.text:6FF5DB38 ; Exported entry 855. __strcmpi
.text:6FF5DB38 ; Exported entry 863. __stricmp
.text:6FF5DB38 ; ============= SUBROUTINE ===============
.text:6FF5DB38 ; Attributes: bp-based frame
.text:6FF5DB38 ; int _cdecl strcmpi(const char *, const char *)
.text:6FF5DB38 public _strcmpi
.text:6FF5DB38 _strcmpi proc near
.CODE XREF: LocaleEnumProc-2B
.LocaleEnumProc+5E
.text:6FF5DB38 arg_0 = dword ptr 8
.text:6FF5DB38 arg_4 = dword ptr 0Ch
.text:6FF5DB38 ; FUNCTION CHUNK AT .text:6FF68CFD SIZE 00000012 BYTES
.text:6FF5DB38 ; FUNCTION CHUNK AT .text:6FF9D20D SIZE 00000022 BYTES
.text:6FF5DB38 .text:6FF5DB38 mov edi, edi ; __strcmpi
.text:6FF5DB3A push ebp
.text:6FF5DB3B mov ebp, esp
.text:6FF5DB3D push esi
.text:6FF5DB3E xor esi, esi
.text:6FF5DB40 cmp dword 6FFFO000, esi
.text:6FF5DB46 jnz loc 6FFF8CFD
.text:6FF5DB4C cmp [ebp+arg_0], esi ; is arg_0==NULL?
.text:6FF5DB4F jz loc 6FF9D20D
.text:6FF5DB55 cmp [ebp+arg_4], esi ; is arg_0==NULL?
.text:6FF5DB58 jz loc 6FF9D20D
.text:6FF5DB5E pop esi
.text:6FF5DB5F pop ebp
.text:6FF5DB5F _strcmpi endp ; sp-analysis failed
```

Actual strings comparison here:

```
.text:6FF5DB60 sub_6FF5DB60 proc near ; CODE XREF: _stricmp_l+16C7F
.text:6FF5DB60 ; sub_6FFD19CD+229
.text:6FF5DB60 .text:6FF5DB60 arg_0 = dword ptr 8
.text:6FF5DB60 arg_4 = dword ptr 0Ch
.text:6FF5DB60 .text:6FF5DB60 push ebp
.text:6FF5DB61 mov ebp, esp
.text:6FF5DB63 push edi
.text:6FF5DB64 push esi
.text:6FF5DB65 push ebx
.text:6FF5DB66 mov esi, [ebp+arg_4]
.text:6FF5DB69 mov edi, [ebp+arg_0]
```
Now the invalid_parameter() function:

```
.text:6FF5DB6C  mov   al, 0FFh
.text:6FF5DB6E  mov   edi, edi
.text:6FF5DB70  .text:6FF5DB70 loc_6FF5DB70:                         ; CODE XREF: sub_6FF5DB60+20
.text:6FF5DB70  or    al, al
.text:6FF5DB72  jz    short loc_6FF5DBA6
.text:6FF5DB74  mov   al, [esi]
.text:6FF5DB76  add   esi, 1
.text:6FF5DB79  mov   ah, [edi]
.text:6FF5DB7B  add   edi, 1
.text:6FF5DB7E  cmp   ah, al
.text:6FF5DB80  jz    short loc_6FF5DB70
.text:6FF5DB82  sub   al, 41h
.text:6FF5DB84  cmp   al, 1Ah
.text:6FF5DB86  sbb  cl, cl
.text:6FF5DB88  and   cl, 20h
.text:6FF5DB8B  add   al, cl
.text:6FF5DB8D  add   al, 41h
.text:6FF5DB8F  xchg  ah, al
.text:6FF5DB91  sub   al, 41h
.text:6FF5DB93  cmp   al, 1Ah
.text:6FF5DB95  sbb  cl, cl
.text:6FF5DB97  and   cl, 20h
.text:6FF5DB99  add   al, cl
.text:6FF5DB9C  add   al, 41h
.text:6FF5DB9E  cmp   al, ah
.text:6FF5DBA0  jz    short loc_6FF5DB70
.text:6FF5DBA2  sbb  al, al
.text:6FF5DBA4  sbb  al, 0FFh
.text:6FF5DBA6  .text:6FF5DBA6 loc_6FF5DBA6:                         ; CODE XREF: sub_6FF5DB60+12
.text:6FF5DBA6  movsx  eax, al
.text:6FF5DBA9  pop    ebx
.text:6FF5DBAA  pop    esi
.text:6FF5DBAB  pop    edi
.text:6FF5DBAC  leave
.text:6FF5DBAD  ret
.text:6FF5DBAD  sub_6FF5DB60 endp
.text:6FF68D0C  loc_6FF68D0C:                               ; CODE XREF: _strcmi+3F6F2
.text:6FF68D0C  pop    esi
.text:6FF68D0D  pop    ebp
.text:6FF68D0E  ret
.text:6FF9D20D  loc_6FF9D20D:                             ; CODE XREF: _strcmi+17
.text:6FF9D20D  ; _strcmi+20
.text:6FF9D20D  call   near ptr _errno
.text:6FF9D212  push   esi
.text:6FF9D213  push   esi
.text:6FF9D214  push   esi
.text:6FF9D215  push   esi
.text:6FF9D216  push   esi
.text:6FF9D217  mov    dword ptr [eax], 16h
.text:6FF9D21D  call   _invalid_parameter
.text:6FF9D222  add    esp, 14h
.text:6FF9D225  mov    eax, 7FFFFFFh
.text:6FF9D22A  jmp    loc_6FF68D0C
```

Now the invalid_parameter() function:
The `invalid_parameter()` function eventually calls the function with `OutputDebugStringA()`.

You see, the `stricmp()` code is like:

```c
int stricmp(const char *s1, const char *s2, size_t len)
{
    if (s1==NULL || s2==NULL)
    {
        // print error message AND exit:
        return 0x7FFFFFFFh;
    }
    // do comparison
}
```

How come this error is rare? Because newer MSVC versions links against MSVCR120.DLL file, etc (where 120 is version number).

Let's peek inside the newer MSVCR120.DLL from Windows 7:
But how it was working in the era of Windows XP? It wasn’t: MSVCRT.DLL from Windows XP doesn’t check

Now the invalid parameter() function is rewritten in newer MSVCR*.DLL version, it shows the message box, if you want to kill the process or call debugger. Of course, this is much better than silently return. Perhaps, Microsoft forgot to fix MSVCRT.DLL since then.

But how it was working in the era of Windows XP? It wasn’t: MSVCRT.DLL from Windows XP doesn’t check
arguments against NULL. So under Windows XP my `stricmp ("asd", NULL)` code will crash, and this is good.

My hypothesis: Microsoft upgraded MSVCRT*.DLL files (including MSVCRT.DLL) for Windows 7 by adding sanitizing checks everywhere. However, since MSVCRT.DLL wasn't used much since MSVS .NET (year 2002), it wasn't properly tested and the bug left here. But compilers like MinGW can still use this DLL.

What would I do without my reverse engineering skills?
The MSVCRT.DLL from Windows 8.1 has the same bug.

### 8.17 Other examples

An example about Z3 and manual decompilation was here. It is moved there: [https://yurichev.com/writings/SAT_SMT_by_example.pdf](https://yurichev.com/writings/SAT_SMT_by_example.pdf).
Chapter 9

Examples of reversing proprietary file formats

9.1 Primitive XOR-encryption

9.1.1 Simplest ever XOR encryption

I once saw a software where all debugging messages has been encrypted using XOR by value of 3. In other words, two lowest bits of all characters has been flipped.

“Hello, world” would become “Kfool/#tlqog”:

```
#!/usr/bin/python
msg="Hello, world!"
print ".join(map(lambda x: chr(ord(x)^3), msg))
```

This is quite interesting encryption (or rather obfuscation), because it has two important properties: 1) single function for encryption/decryption, just apply it again; 2) resulting characters are also printable, so the whole string can be used in source code without escaping characters.

The second property exploits the fact that all printable characters organized in rows: 0x2x-0x7x, and when you flip two lowest bits, character moving 1 or 3 characters left or right, but never moved to another (maybe non-printable) row:

![ASCII table](image)

Figure 9.1: 7-bit ASCII table in Emacs

...with a single exception of 0x7F character.

For example, let’s encrypt characters in A-Z range:

```
#!/usr/bin/python
msg="@ABCDEFGHIJKLMNO"
print ".join(map(lambda x: chr(ord(x)^3), msg))
```

Result:
It’s like “@” and “C” characters have been swapped, and so are “B” and “a”.
Yet again, this is an interesting example of exploiting XOR properties, rather than encryption: the very same effect of preserving printable ness can be achieved while flipping any of lowest 4 bits, in any combination.
Norton Guide was popular in the epoch of MS-DOS, it was a resident program that worked as a hypertext reference manual.

Norton Guide’s databases are files with the extension .ng, the contents of which look encrypted:

![Image of Norton Guide database]

Why did we think that it’s encrypted but not compressed?

We see that the 0x1A byte (looking like “→”) occurs often, it would not be possible in a compressed file.

We also see long parts that consist only of Latin letters, and they look like strings in an unknown language.

Figure 9.2: Very typical look
Since the 0x1A byte occurs so often, we can try to decrypt the file, assuming that it’s encrypted by the simplest XOR-encryption.

If we apply XOR with the 0x1A constant to each byte in Hiew, we can see familiar English text strings:

![Figure 9.3: Hiew XORing with 0x1A](image)

XOR encryption with one single constant byte is the simplest possible encryption method, which is, nevertheless, encountered sometimes.

Now we understand why the 0x1A byte is occurring so often: because there are so many zero bytes and they were replaced by 0x1A in encrypted form.

But the constant might be different. In this case, we could try every constant in the 0..255 range and look for something familiar in the decrypted file. 256 is not so much.


**Entropy**

A very important property of such primitive encryption systems is that the information entropy of the encrypted/decrypted block is the same.

Here is my analysis in Wolfram Mathematica 10.

![Listing 9.1: Wolfram Mathematica 10](image)
What we do here is load the file, get its entropy, decrypt it, save it and get the entropy again (the same!). Mathematica also offers some well-known English language texts for analysis. So we also get the entropy of Shakespeare’s sonnets, and it is close to the entropy of the file we just analyzed. The file we analyzed consists of English language sentences, which are close to the language of Shakespeare. And the XOR-ed bitwise English language text has the same entropy. However, this is not true when the file is XOR-ed with a pattern larger than one byte. The file we analyzed can be downloaded here: http://beginners.re/examples/norton_guide/X86.NG.

One more word about base of entropy

Wolfram Mathematica calculates entropy with base of $e$ (base of the natural logarithm), and the UNIX `ent` utility\(^1\) uses base 2. So we set base 2 explicitly in `Entropy` command, so Mathematica will give us the same results as the `ent` utility.

\(^1\)http://www.fourmilab.ch/random/
9.1.3 Simplest possible 4-byte XOR encryption

If a longer pattern was used for XOR-encryption, for example a 4 byte pattern, it’s easy to spot as well. For example, here is the beginning of the kernel32.dll file (32-bit version from Windows Server 2008):

Figure 9.4: Original file
Here it is “encrypted” with a 4-byte key:

Figure 9.5: “Encrypted” file

It’s very easy to spot the recurring 4 symbols.

Indeed, the header of a PE-file has a lot of long zero areas, which are the reason for the key to become visible.
Here is the beginning of a PE-header in hexadecimal form:

![PE-header in hexadecimal form]

**Figure 9.6: PE-header**
It’s easy to spot that the key is the following 4 bytes: \texttt{8C 61 D2 63}.

With this information, it’s easy to decrypt the whole file.

So it is important to keep in mind these properties of PE-files: 1) PE-header has many zero-filled areas; 2) all PE-sections are padded with zeros at a page boundary (4096 bytes), so long zero areas are usually present after each section.

Some other file formats may contain long zero areas.

It’s typical for files used by scientific and engineering software.

For those who want to inspect these files on their own, they are downloadable here: \url{http://go.yurichev.com/17352}.

\textbf{Exercise}

- \url{http://challenges.re/50}
9.1.4 Simple encryption using XOR mask

I've found an old interactive fiction game while diving deep into if-archive:

The New Castle v3.5 - Text/Adventure Game in the style of the original Infocom (tm) type games, Zork, Collosal Cave (Adventure), etc. Can you solve the mystery of the abandoned castle?

Shareware from Software Customization.


There is a file inside (named castle.dbf) which is clearly encrypted, but not by a real crypto algorithm, nor it's compressed, this is something rather simpler. I wouldn't even measure entropy level (9.2 on page 929) of the file, because I'm sure it's low. Here is how it looks like in Midnight Commander:

![Figure 9.8: Encrypted file in Midnight Commander](image)

The encrypted file can be downloaded here: https://beginners.re/current-tree/ff/XOR/mask_1/files/castle.dbf.bz2.

Will it be possible to decrypt it without accessing to the program, using just this file?

There is a clearly visible pattern of repeating string. If a simple encryption by XOR mask was applied, such repeating strings is a prominent signature, because, probably, there were a long lacunas of zero bytes, which, in turn, are present in many executable files as well as in binary data files.

Here I'll dump the file's beginning using `xxd` UNIX utility:

```
...
0000080: 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e 1d 61 .a.c.w.iubgv.~....a
0000090: 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e 7a az..rn..}}c~wf.z
00000a0: 75 50 02 4a 31 71 31 33 5c 27 08 5c 51 74 3e 39 uP.J1q13\'.\'Qt>9
00000b0: 50 2e 28 72 24 4b 38 21 4l 09 37 38 3b 51 41 2c P.(r$K8!l.78;QA,1
00000c0: 1c 3c 77 5d 27 5a 1c 7c 6a 10 14 68 77 08 6d 1a .<7}].Z.|j..hw.m.
00000d0: 6a 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e 1d j.a.c.w.iubgv.~....a
00000e0: 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e 7a az..rn..}}c~wf.z
00000f0: 75 50 02 4a 31 71 31 33 5c 27 08 5c 51 74 3e 39 uP.J1q13\'.\'Qt>9
```

2http://www.ifarchive.org/
3As in https://en.wikipedia.org/wiki/Lacuna_(manuscripts)
Let's stick at visible repeating iubgv string. By looking at this dump, we can clearly see that the period of the string occurrence is 0x51 or 81. Probably, 81 is size of block? The size of the file is 1658961, and it can be divided evenly by 81 (and there are 20481 blocks then).

Now I'll use Mathematica to analyze, are there repeating 81-byte blocks in the file? I'll split input file by 81-byte blocks and then I'll use `Tally[]` function which just counts, how many times some item has been occurred in the input list. `Tally`'s output is not sorted, so I also add `Sort[]` function to sort it by number of occurrences in descending order.

```plaintext
input = BinaryReadList["/home/dennis/.../castle.dbf"];
blocks = Partition[input, 81];
stat = Sort[Tally[blocks], #1[[2]] > #2[[2]] &]
```

And here is output:

```plaintext
...

{{80, 1, 74, 59, 113, 45, 56, 86, 52, 91, 19, 64, 60, 60, 63,
25, 38, 59, 59, 42, 14, 53, 38, 77, 66, 38, 113, 38, 75, 4, 43, 84,
63, 101, 64, 43, 79, 64, 40, 57, 16, 91, 46, 119, 69, 40, 84, 117,
9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126, 29,
97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102, 30,
122, 2, 117}, 1},
{{80, 2, 74, 49, 113, 49, 51, 92, 39, 8, 92, 81, 116, 62, 57,
80, 46, 40, 114, 36, 75, 56, 33, 76, 9, 55, 56, 59, 81, 65, 45, 28,
60, 55, 93, 39, 90, 28, 124, 106, 16, 20, 104, 119, 8, 109, 26,
106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126,
29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102,
30, 122, 2, 117}, 1}}

Tally’s output is a list of pairs, each pair has 81-byte block and number of times it has been occurred in
the file. We see that the most frequent block is the first, it has been occurred 1739 times. The second
one has been occurred 1422 times. There are others: 1012 times, 377 times, etc. 81-byte blocks which
has been occurred just once are at the end of output.
Let’s try to compare these blocks. The first and the second. Is there a function in Mathematica which
compares lists/arrays? Certainly is, but for educational purposes, I’ll use XOR operation for comparison.
Indeed: if bytes in two input arrays are identical, XOR result is 0. If they are non-equal, result will be
non-zero.
Let’s compare first block (occurred 1739 times) and the second (occurred 1422 times):
In[]:= BitXor[stat[[1]][[1]], stat[[2]][[1]]]
Out[]= {0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

They are differ only in the second byte.
Let’s compare the second block (occurred 1422 times) and the third (occurred 1012 times):
In[]:= BitXor[stat[[2]][[1]], stat[[3]][[1]]]
Out[]= {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}

They are also differ only in the second byte.
Anyway, let’s try to use the most occurred block as a XOR key and try to decrypt four first 81-byte blocks
in the file:
In[]:= key = stat[[1]][[1]]
Out[]= {80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23,
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117,
5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98,
1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99,
102, 30, 122, 2, 117}

116,
113,
104,
103,
126,

\
111,
119,
118,
119,

\
\
\
\

In[]:= ToASCII[val_] := If[val == 0, " ", FromCharacterCode[val, "PrintableASCII"]]
In[]:= DecryptBlockASCII[blk_] := Map[ToASCII[#] &, BitXor[key, blk]]
In[]:= DecryptBlockASCII[blocks[[1]]]
Out[]= {" ", " ", " ", " ", " ", " ",
", " ", " ", " ", " ", " ", " ", " ",
", " ", " ", " ", " ", " ", " ", " ",
", " ", " ", " ", " ", " ", " ", " ",
", " ", " ", " ", " ", " ", " ", " ",
", " ", " ", " ", " ", " ", " ", " ",

"
"
"
"
"
"

",
",
",
",
",
",

"
"
"
"
"
"

",
",
",
",
",
",

"
"
"
"
"
"

",
",
",
",
",
",

"
"
"
"
"
"

In[]:= DecryptBlockASCII[blocks[[2]]]

919

",
",
",
",
",
",

"
"
"
"
"
"

",
",
",
",
",
",

"
"
"
"
"
"

\
",
",
",
",
"}

"
"
"
"

\
\
\
\

", "O", \
"R", "S", " ", \
"T", "?", \
" ", " ", " ", \


(I've replaced unprintable characters by "?".)

So we see that the first and the third blocks are empty (or almost empty), but the second and the fourth has clearly visible English language words/phrases. It seems that our assumption about key is correct (at least partially). This means that the most occurred 81-byte block in the file can be found at places of lacunas of zero bytes or something like that.

Let's try to decrypt the whole file:

```
DecryptBlock[blk_] := BitXor[key, blk]
decrypted = Map[DecryptBlock[#] &, blocks];
BinaryWrite["/home/dennis/.../tmp", Flatten[decrypted]]
Close["/home/dennis/.../tmp"]
```
Figure 9.9: Decrypted file in Midnight Commander, 1st attempt

Looks like some kind of English phrases from some game, but something wrong. First of all, cases are inverted: phrases and some words are started with lowercase characters, while other characters are in upper case. Also, some phrases started with wrong letters. Take a look at the very first phrase: “eHE WEED OF CRIME BEARS BITTER FRUIT”. What is “eHE”? Isn’t “tHE” have to be here? Is it possible that our decryption key has wrong byte at this place?

Let’s look again at the second block in the file, at key and at decryption result:

<table>
<thead>
<tr>
<th>In[4] := blocks[[2]]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out[4] := {80, 2, 74, 49, 113, 49, 51, 92, 39, 8, 92, 81, 116, 62, 57, 80, 46, 40, 114, 36, 75, 56, 33, 76, 9, 55, 56, 98, 81, 65, 45, 28, 60, 55, 93, 39, 90, 28, 124, 106, 16, 20, 104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102, 30, 122, 2, 117}</td>
</tr>
<tr>
<td>In[5] := key</td>
</tr>
<tr>
<td>In[6] := BitXor[key, blocks[[2]]]</td>
</tr>
<tr>
<td>Out[6] := {0, 101, 72, 69, 0, 87, 69, 69, 68, 0, 79, 70, 0, 67, 82, 73, 77, 69, 0, 66, 69, 65, 82, 83, 0, 66, 73, 84, 84, 69, 82, 0, 70, 82, 85, 73, 84, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}</td>
</tr>
</tbody>
</table>
Encrypted byte is 2, the byte from the key is 103, \(2 \oplus 103 = 101\) and 101 is ASCII code for “e” character. What byte of a key must be equal to, so the resulting ASCII code will be 116 (for “t” character)? \(2 \oplus 116 = 118\), let’s put 118 in key at the second byte ...

\[

...and decrypt the whole file again.

![Image](image.jpg)

Figure 9.10: Decrypted file in Midnight Commander, 2nd attempt

Wow, now the grammar is correct, all phrases started with correct letters. But still, case inversion is suspicious. Why would game’s developer write them in such a manner? Maybe our key is still incorrect?

While observing ASCII table we can notice that uppercase and lowercase letter’s ASCII codes are differ in just one bit (6th bit starting at 1st, 0b100000):

![Image](image2.jpg)

Figure 9.11: 7-bit ASCII table in Emacs

6th bit set in a zero byte has decimal form of 32. But 32 is ASCII code for space!
Indeed, one can switch case just by XOR-ing ASCII character code with 32 (more about it: 3.19.3 on page 536).

It is possible that the empty lacunas in the file are not zero bytes, but rather spaces? Let’s modify XOR key one more time (I’ll XOR each byte of key by 32):

\[
\text{(* "32" is scalar and "key" is vector, but that's OK *)}
\]

\[
\text{In[]} := \text{key3} = \text{BitXor}[32, \text{key}]
\]

\[
\text{Out[]} = \{112, 86, 34, 84, 81, 70, 86, 57, 67, 40, 51, 55, 84, 93, 75, \ \\
57, 67, 77, 82, 70, 46, 89, 83, 63, 41, 85, 81, 79, 37, 36, 95, 60, \ \\
90, 69, 40, 78, 46, 50, 92, 74, 48, 52, 72, 87, 40, 77, 58, 74, 41, \ \\
65, 45, 67, 47, 87, 52, 73, 85, 66, 71, 86, 33, 94, 61, 65, 90, 49, \ \\
47, 82, 78, 35, 37, 93, 93, 67, 94, 87, 70, 62, 90, 34, 85\}
\]

\[
\text{In[]} := \text{DecryptBlock}[\text{blk}_] := \text{BitXor}[\text{key3}, \text{blk}]
\]

Let’s decrypt the input file again:

\[
\begin{array}{c}
\text{Figure 9.12: Decrypted file in Midnight Commander, final attempt}
\end{array}
\]

(Decrypted file is available here: https://beginners.re/current-tree/ff/XOR/mask_1/files/decrypted.dat.bz2.)

This is undoubtedly a correct source file. Oh, and we see numbers at the start of each block. It has to be a source of our erroneous XOR key. As it seems, the most occurred 81-byte block in the file is a block filled with spaces and containing “1” character at the place of second byte. Indeed, somehow, many blocks here are interleaved with this one. Maybe it’s some kind of padding for short phrases/messages? Other frequently occurred 81-byte blocks are also space-filled blocks, but with different digit, hence, they are differ only at the second byte.

That’s all! Now we can write an utility to encrypt the file back, and maybe modify it before.
Mathematica notebook file is downloadable here: https://beginners.re/current-tree/ff/XOR/mask_1/files/XOR_mask_1.nb.

Summary: XOR encryption like that is not robust at all. It has been intended by game's developer(s), probably, just to prevent gamer(s) to peek into internals of game, nothing else more serious. Still, encryption like that is extremely popular due to its simplicity and many reverse engineers are usually familiar with it.

9.1.5 Simple encryption using XOR mask, case II

I've got another encrypted file, which is clearly encrypted by something simple, like XOR-ing:

<table>
<thead>
<tr>
<th>/home/dennis/tmp/cipher.txt</th>
<th>0x00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000 D2 0F 70</td>
<td>IC E7 9E 8D</td>
</tr>
<tr>
<td>00000010 5C F5 D3 0D</td>
<td>DF F2 E2 BC</td>
</tr>
<tr>
<td>00000020 98 5D FC D9</td>
<td>BB 33 61 7B</td>
</tr>
<tr>
<td>00000030 14 D9 45 F8</td>
<td>E4 BC 7A 61</td>
</tr>
<tr>
<td>00000040 61 1B 8F 54</td>
<td>20 E1 DB 8B</td>
</tr>
<tr>
<td>00000050 61 7C 15 6D</td>
<td>22 A8 FE 9E</td>
</tr>
<tr>
<td>00000060 39 22 77 1B</td>
<td>52 70 88 E7</td>
</tr>
<tr>
<td>00000070 HH 76 35 73</td>
<td>80 40 3C 23</td>
</tr>
<tr>
<td>00000080 EB 7B 9A 0C</td>
<td>83 C5 40 24</td>
</tr>
<tr>
<td>00000090 E4 F7 EF 22</td>
<td>FA B6 7F E9</td>
</tr>
<tr>
<td>000000A0 DF E1 E2 AD</td>
<td>B0 11 E3 D4</td>
</tr>
<tr>
<td>000000B0 8E BF EA ED</td>
<td>9A 15 4F 41</td>
</tr>
<tr>
<td>000000C0 FD BE 93 A5</td>
<td>62 19 99 5A</td>
</tr>
<tr>
<td>000000D0 3F AF 8F 97</td>
<td>8E 0E 25 55</td>
</tr>
<tr>
<td>000000E0 33 2F AF 95</td>
<td>3F 32 7B 0E</td>
</tr>
<tr>
<td>000000F0 48 34 6F E3</td>
<td>9C EE A3 BF</td>
</tr>
<tr>
<td>00000100 CF 70 38</td>
<td>2E 08 3E 35</td>
</tr>
<tr>
<td>00000110 FF 22 29 77</td>
<td>94 80 9C 57</td>
</tr>
<tr>
<td>00000120 3F AF 8F 97</td>
<td>3F 45 9A 5A</td>
</tr>
<tr>
<td>00000130 FF 22 29 77</td>
<td>94 80 9C 57</td>
</tr>
<tr>
<td>00000140 3F AF 8F 97</td>
<td>3F 45 9A 5A</td>
</tr>
<tr>
<td>00000150 3F AF 8F 97</td>
<td>3F 45 9A 5A</td>
</tr>
<tr>
<td>00000160 2F 70 00 97</td>
<td>9A 9A 5A 4D</td>
</tr>
<tr>
<td>00000170 83 26 32 16</td>
<td>9A 1A 5A 4D</td>
</tr>
<tr>
<td>00000180 5B EE 9C 05</td>
<td>A7 80 4E 97</td>
</tr>
<tr>
<td>00000190 08 73 EE 9C 05</td>
<td>9A 1A 5A 4D</td>
</tr>
<tr>
<td>000001A0 0E F1 34</td>
<td>2C 7C 1E 89</td>
</tr>
<tr>
<td>000001B0 9E EE 9C 05</td>
<td>89 87 2C 77</td>
</tr>
<tr>
<td>000001C0 D9 AE AA 9A</td>
<td>99 7F 1C 61</td>
</tr>
<tr>
<td>000001D0 EC B9 D3 85</td>
<td>7E 80 4E 97</td>
</tr>
<tr>
<td>000001E0 20 E2 9B 7E</td>
<td>8F FF 0E 30</td>
</tr>
<tr>
<td>000001F0 36 6F F3 93</td>
<td>78 8B 3C 32</td>
</tr>
</tbody>
</table>

Figure 9.13: Encrypted file in Midnight Commander

The encrypted file can be downloaded here.

ent Linux utility reports about ~7.5 bits per byte, and this is high level of entropy (9.2 on page 929), close to compressed or properly encrypted file. But still, we clearly see some pattern, there are some blocks with size of 17 bytes, and you can see some kind of ladder, shifting by 1 byte at each 16-byte line.

It's also known that the plain text is just English language text.

Now let's assume that this piece of text is encrypted by simple XOR-ing with 17-byte key.

I tried to find some repeating 17-byte blocks using Mathematica, like I did before in my previous example (9.1.4 on page 917):

```
Listing 9.2: Mathematica

In[]:=input = BinaryReadList["/home/dennis/tmp/cipher.txt"];
```

ent Linux utility reports about ~7.5 bits per byte, and this is high level of entropy (9.2 on page 929), close to compressed or properly encrypted file. But still, we clearly see some pattern, there are some blocks with size of 17 bytes, and you can see some kind of ladder, shifting by 1 byte at each 16-byte line.

It's also known that the plain text is just English language text.

Now let's assume that this piece of text is encrypted by simple XOR-ing with 17-byte key.

I tried to find some repeating 17-byte blocks using Mathematica, like I did before in my previous example (9.1.4 on page 917):

```
Listing 9.2: Mathematica

In[]:=input = BinaryReadList["/home/dennis/tmp/cipher.txt"];
```

ent Linux utility reports about ~7.5 bits per byte, and this is high level of entropy (9.2 on page 929), close to compressed or properly encrypted file. But still, we clearly see some pattern, there are some blocks with size of 17 bytes, and you can see some kind of ladder, shifting by 1 byte at each 16-byte line.

It's also known that the plain text is just English language text.

Now let's assume that this piece of text is encrypted by simple XOR-ing with 17-byte key.
In[]:= blocks = Partition[input, 17];
In[]:= Sort[Tally[blocks], #1[[2]] > #2[[2]] &]
Out[]:= {{228, 128, 63, 58, 175, 159, 154, 232, 226, 161, 50, 97, 127, 3, 217, 80}, 1},
{{252, 171, 1, 39, 39, 143, 241, 235, 170, 91, 75, 119, 2, 152, 82}, 1},
{{241, 196, 71, 112, 56, 230, 143, 151, 234, 246, 187, 118, 44, 125, 8, 156, 17}, 1},
{{255, 206, 71, 112, 56, 231, 158, 143, 151, 234, 246, 187, 118, 44, 125, 8, 156, 17}, 1},
..., 

No luck, each 17-byte block is unique within the file and occurred only once. Perhaps, there are no 17-byte zero lacunas, or lacunas containing only spaces. It is possible indeed: such long space indentation and padding may be absent in tightly typeset text.

The first idea is to try all possible 17-byte keys and find those, which will result in readable text after decryption. Bruteforce is not an option, because there are \(2^{17}\) possible keys (~\(10^{40}\)), that's too much. But there are good news: who said we have to test 17-byte key as a whole, why can't we test each byte of key separately? It is possible indeed.

Now the algorithm is:

- try all 256 bytes for 1st byte of key;
- decrypt 1st byte of each 17-byte blocks in the file;
- are all decrypted bytes we got are printable? keep tabs on it;
- do so for all 17 bytes of key.

I've written the following Python script to check this idea:

```
Listing 9.3: Python script
each_Nth_byte=['' for _ in range(KEY_LEN)]
content=read_file(sys.argv[1])
# split input by 17-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    possible_keys=[]
    for i in range(256):
        tmp_key=chr(i)*len(each_Nth_byte[N])
        tmp=xor_strings(tmp_key,each_Nth_byte[N])
        # are all characters in tmp[] are printable?
        if is_string_printable(tmp)==False:
            continue
        possible_keys.append(i)
    print possible_keys, "len=", len(possible_keys)
```

(Full version of the source code is [here](#).)

Here is its output:

```
N= 0
[144, 145, 151] len= 3
N= 1
```

925
So there are 2 or 3 possible bytes for each byte of 17-byte key. This is much better than 256 possible bytes for each byte, but still too much. There are up to 1 million of possible keys:

Listing 9.4: Mathematica

\[
\text{In}[1]:= 3 \times 2 \times 3 \times 2 \times 2 \times 2 \times 3 \times 2 \times 2 \times 3 \times 2 \times 2 \times 2 \\
\text{Out}[1]= 995328
\]

It's possible to check all of them, but then we must check visually, if the decrypted text is looks like English language text.

Let's also take into consideration the fact that we deal with 1) natural language; 2) English language. Natural languages have some prominent statistical features. First of all, punctuation and word lengths. What is average word length in English language? Let’s just count spaces in some well-known English language texts using Mathematica.

Here is “The Complete Works of William Shakespeare” text file from Gutenberg Library:

Listing 9.5: Mathematica

\[
\text{In}[1]:= \text{input} = \text{BinaryReadList}("/home/dennis/tmp/pg100.txt"); \\
\text{In}[2]:= \text{Tally}[\text{input}]; \\
\text{Out}[2]= \{\{\text{239}, 1\}, \{\text{187}, 1\}, \{\text{191}, 1\}, \{\text{84}, 39878\}, \{\text{104}, 218875\}, \{\text{101}, 486157\}, \{\text{32}, 1285884\}, \{\text{80}, 12038\}, \{\text{114}, 269907\}, \{\text{111}, 282560\}, \{\text{106}, 2788\}, \{\text{99}, 67194\}, \{\text{116}, 291243\}, \{\text{71}, 11261\}, \{\text{117}, 115225\}, \{\text{110}, 216805\}, \{\text{98}, 46768\}, \{\text{103}, 57328\}, \{\text{69}, 42703\}, \{\text{66}, 15450\}, \{\text{107}, 29345\}, \{\text{102}, 69103\}, \{\text{67}, 21526\}, \{\text{109}, 95890\}, \{\text{112}, 46849\}, \{\text{108}, 146532\}, \{\text{87}, 16508\}, \{\text{115}, 215605\}, \{\text{105}, 199130\}, \{\text{97}, 245509\}, \{\text{83}, 34082\}, \{\text{44}, 83315\}, \{\text{121}, 85549\}, \{\text{13}, 124787\}, \{\text{10}, 124787\}, \{\text{119}, 73155\}, \{\text{100}, 134216\}, \{\text{118}, 34077\}, \{\text{46}, 78216\}, \{\text{89}, 9128\}, \{\text{45}, 8150\}, \{\text{76}, 23919\}, \{\text{42}, 73\}, \{\text{79}, 33268\}, \{\text{82}, 29040\}, \{\text{73}, 55893\}, \{\text{72}, 18486\}, \{\text{68}, 15726\}, \{\text{58}, 1843\}, \{\text{65}, 44560\}, \{\text{49}, 982\}, \{\text{50}, 373\}, \{\text{48}, 325\}, \{\text{91}, 2076\}, \{\text{35}, 3\}, \{\text{93}, 2068\}, \{\text{74}, 2071\}, \{\text{57}, 966\}, \{\text{52}, 107\}, \{\text{70}, 11770\}, \{\text{85}, 14169\}, \{\text{78}, 27393\}, \{\text{75}, 6206\}, \{\text{77}, 15887\}, \{\text{120}, 4681\}, \{\text{33}, 8840\}, \{\text{60}, 468\}, \{\text{86}, 3587\}, \{\text{51}, 343\}, \{\text{88}, 608\}, \{\text{40}, 643\}, \{\text{41}, 644\}, \{\text{62}, 440\}, \{\text{39}, 31077\}, \{\text{34}, 488\}, \{\text{59}, 17199\}, \{\text{126}, 1\}, \{\text{95}, 71\}, \{\text{113},
\]
In:
\[
\text{Length[input]/1285884 // N}
\]
Out[1]= 4.34712

There are 1285884 spaces in the whole file, and the frequency of space occurrence is 1 space per ~4.3 characters.

Now here is Alice’s Adventures in Wonderland, by Lewis Carroll from the same library:

In:
\[
\text{Length[input]/27964 // N}
\]
Out[1]= 5.99049

The result is different probably because of different formatting of these texts (maybe indentation and/or padding).

OK, so let’s assume the average frequency of space in English language is 1 space per 4.7 characters.

Now the good news again: we can measure frequency of spaces while decrypting our file gradually. Now I count spaces in each slice and throw away 1-byte keys which produce results with too small number of spaces (or too large, but this is almost impossible given so short key):

In: input = BinaryReadList["/home/dennis/tmp/pg11.txt"];
In[]: Tally[input]
Out[]={
\{239, 1\}, \{187, 1\}, \{191, 1\}, \{80, 172\}, \{114, 6398\}, \{111, 9243\}, \{106, 222\}, \{101, 15082\}, \{99, 2815\}, \{96, 1\}, \{125, 2\}, \{90, 532\}, \{124, 33\}, \{38, 21\}, \{98, 1\}, \{125, 2\}, \{37, 1\}, \{36, 2\}
\}

In[]:= Length[input]/27964 // N
Out[]= 5.99049

The result is different probably because of different formatting of these texts (maybe indentation and/or padding).

OK, so let’s assume the average frequency of space in English language is 1 space per 4.7 characters.

Now the good news again: we can measure frequency of spaces while decrypting our file gradually. Now I count spaces in each slice and throw away 1-byte keys which produce results with too small number of spaces (or too large, but this is almost impossible given so short key):

In: each_Nth_byte="\""*KEY_LEN
content=read_file(sys.argv[1])
# split input by 17-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    possible_keys=[]
    for i in range(256):
        tmp_key=chr(i)*len(each_Nth_byte[N])
        tmp=xor_strings(tmp_key,each_Nth_byte[N])
        # are all characters in tmp[] are printable?
        if is_string_printable(tmp)==False:
            continue
        # count spaces in decrypted buffer:
        spaces=tmp.count(' ')
        if spaces==0:
            continue

927
spaces_ratio = len(tmp) / spaces
if spaces_ratio < 4:
    continue
if spaces_ratio > 7:
    continue
possible_keys.append(i)
print(possible_keys, "len=", len(possible_keys))

(Full version of the source code is here.)

This reports just one single possible byte for each byte of key:

N= 0
[144] len= 1
N= 1
[160] len= 1
N= 2
[33] len= 1
N= 3
[80] len= 1
N= 4
[79] len= 1
N= 5
[143] len= 1
N= 6
[251] len= 1
N= 7
[255] len= 1
N= 8
[133] len= 1
N= 9
[131] len= 1
N= 10
[207] len= 1
N= 11
[86] len= 1
N= 12
[65] len= 1
N= 13
[18] len= 1
N= 14
[122] len= 1
N= 15
[249] len= 1
N= 16
[49] len= 1

Let's check this key in Mathematica:

Listing 9.8: Mathematica

In[] := input = BinaryReadList["/home/dennis/tmp/cipher.txt"];

In[] := blocks = Partition[input, 17];

In[] := key = {144, 160, 33, 80, 79, 143, 251, 255, 133, 131, 207, 86, 65, 18, 122, 249, 49};

In[] := EncryptBlock[blk_] := BitXor[key, blk]

In[] := encrypted = Map[EncryptBlock[#] &, blocks];

In[] := BinaryWrite["/home/dennis/tmp/plain2.txt", Flatten[encrypted]]

In[] := Close["/home/dennis/tmp/plain2.txt"]

And the plain text is:

Mr. Sherlock Holmes, who was usually very late in the mornings, save
upon those not infrequent occasions when he was up all night, was seated
at the breakfast table. I stood upon the hearth-rug and picked up the
stick which our visitor had left behind him the night before. It was a
fine, thick piece of wood, bulbous-headed, of the sort which is known as a "Penang lawyer." Just under the head was a broad silver band nearly an inch across. "To James Mortimer, M.R.C.S., from his friends of the C.C.H.," was engraved upon it, with the date "1884." It was just such a stick as the old-fashioned family practitioner used to carry--dignified, solid, and reassuring.

"Well, Watson, what do you make of it?"

Holmes was sitting with his back to me, and I had given him no sign of my occupation.

(Full version of the text is here.)

The text looks correct. Yes, I made up this example and choose well-known text of Conan Doyle, but it’s very close to what I had in my practice some time ago.

Other ideas to consider

If we would fail with space counting, there are other ideas to try:

- Take into consideration the fact that lowercase letters are much more frequent than uppercase ones.
- Frequency analysis.
- There is also a good technique to detect language of a text: trigrams. Each language has some very frequent letter triplets, these may be “the” and “tha” for English. Read more about it: N-Gram-Based Text Categorization, http://code.activestate.com/recipes/326576/. Interestingly enough, trigrams detection can be used when you decrypt a ciphertext gradually, like in this example (you just have to test 3 adjacent decrypted characters).

For non-Latin writing systems encoded in UTF-8, things may be easier. For example, Russian text encoded in UTF-8 has each byte interleaved with 0xD0/0xD1 byte. It is because Cyrillic characters are placed in 4th block of Unicode table. Other writing systems has their own blocks.

9.1.6 Homework

An ancient text adventure for MS-DOS, developed in the end of 1980’s. To conceal game information from player, data files, most likely, XOR-ed with something: https://beginners.re/homework/XOR_crypto_1/destiny.zip. Try to get into...

9.2 Information entropy

Entropy: The quantitative measure of disorder, which in turn relates to the thermodynamic functions, temperature, and heat.

Dictionary of Applied Math for Engineers and Scientists

For the sake of simplification, I would say, information entropy is a measure, how tightly some piece of data can be compressed. For example, it is usually not possible to compress already compressed archive file, so it has high entropy. On the other hand, 1MiB of zero bytes can be compressed to a tiny output file. Indeed, in plain English language, one million of zeros can be described just as “resulting file is one million zero bytes”. Compressed files are usually a list of instructions to decompressor, like this: “put 1000 zeros, then 0x23 byte, then 0x45 byte, then put a block of size 10 bytes which we’ve seen 500 bytes back, etc.”

Texts written in natural languages are also can be compressed tightly, because natural languages has a lot of redundancy (otherwise, a tiny typo will always lead to misunderstanding, like any toggled bit in compressed archive make decompression nearly impossible), some words are used very often, etc. In everyday speech, it’s possible to drop up to half of words and it still be recognizable.

Code for CPUs is also can be compressed, because some ISA instructions are used much more often than others. In x86, most used instructions are MOV/PUSH/CALL ( 5.11.2 on page 724).
Data compressors and ciphers tend to produce very high entropy results. Good PRNG also produce data which cannot be compressed (it is possible to measure their quality by this sign).

So, in other words, entropy is a measure which can help to probe contents of unknown data block.

### 9.2.1 Analyzing entropy in Mathematica

(This part has been first appeared in my blog at 13-May-2015. Some discussion: https://news.ycombinator.com/item?id=9545276.)

It is possible to slice a file by blocks, calculate entropy of each and draw a graph. I did this in Wolfram Mathematica for demonstration and here is a source code (Mathematica 10):

```mathematica
(* loading the file *)
input=BinaryReadList["file.bin"];  
(* setting block sizes *)
BlockSize=4096;BlockSizeToShow=256;  
(* slice blocks by 4k *)
blocks=Partition[input,BlockSize];  
(* how many blocks we've got? *)
Length[blocks]  
(* calculate entropy for each block. 2 in Entropy[] (base) is set with the intention so Entropy[] function will produce the same results as Linux ent utility does *)
entropies=Map[N[Entropy[2,#]]&,blocks];  

(* helper functions *)
fBlockToShow[input_,offset_]:=Take[input,1+offset,1+offset+BlockSizeToShow]  
fToASCII[val_]:=FromCharacterCode[val,"PrintableASCII"]  
fToHex[val_]:=IntegerString[val,16]  
fPutASCIIWindow[data_]:=Framed[Grid[Partition[Map[fToASCII,data],16]]]  
fPutHexWindow[data_]:=Framed[Grid[Partition[Map[fToHex,data],16],Alignment->Right]]

(* that will be the main knob here *)
{Slider[Dynamic[offset],{0,Length[input]-BlockSize,BlockSize}],Dynamic[BaseForm[offset,16]]}

(* main UI part *)
Dynamic[{ListLinePlot[entropies,GridLines->{{-1,offset/BlockSize,1}},Filling->Axis,AxesLabel->{"offset","entropy"}],  
CurrentBlock=fBlockToShow[input,offset];  
fPutHexWindow[CurrentBlock],  
fPutASCIIWindow[CurrentBlock]}]
```

**GeoIP ISP database**

Let's start with the GeoIP file (which assigns ISP to the block of IP addresses). This binary file GeoIPISP.dat has some tables (which are IP address ranges perhaps) plus some text blob at the end of the file (containing ISP names).

When I load it to Mathematica, I see this:
There are two parts in graph: first is somewhat chaotic, second is more steady.

0 in horizontal axis in graph means lowest entropy (the data which can be compressed very tightly, ordered in other words) and 8 is highest (cannot be compressed at all, chaotic or random in other words). Why 0 and 8? 0 means 0 bits per byte (byte as a container is not filled at all) and 8 means 8 bits per byte, i.e., the whole byte container is filled with the information tightly.

So I put slider to point in the middle of the first block, and I clearly see some array of 32-bit integers. Now I put slider in the middle of the second block and I see English text:
Indeed, this are names of ISPs. So, entropy of English text is 4.5-5.5 bits per byte? Yes, something like this. Wolfram Mathematica has some well-known English literature corpus embedded, and we can see entropy of Shakespeare’s sonnets:

\[
\text{In[8]}:= \text{Entropy}[2, \text{ExampleData[\{"Text","ShakespearesSonnets"\}]}]//N
\]

\[
\text{Out[8]}= 4.42366
\]

4.4 is close to what we’ve got (4.7-5.3). Of course, classic English literature texts are somewhat different from ISP names and other English texts we can find in binary files (debugging/logging/error messages), but this value is close.

**TP-Link WR941 firmware**

Next example. I’ve got firmware for TP-Link WR941 router:
We see here 3 blocks with empty lacunas. Then the first block with high entropy (started at address 0) is small, second (address somewhere at 0x22000) is bigger and third (address 0x123000) is biggest. I can't be sure about exact entropy of the first block, but 2nd and 3rd has very high entropy, meaning that these blocks are either compressed and/or encrypted.

I tried binwalk for this firmware file:

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>HEXADECIMAL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0</td>
<td>TP-Link firmware header, firmware version: 0.-15221.3, image version: &quot;&quot;, product ID: 0x0, product version: 155254789, kernel load address: 0x0, kernel entry point: 0x-7FFFE000, kernel offset: 4063744, kernel length: 512, rootsfs offset: 837431, rootsfs length: 1048576, bootloader offset: 2883584, bootloader length: 0</td>
</tr>
<tr>
<td>14832</td>
<td>0x39F0</td>
<td>U-Boot version string, &quot;U-Boot 1.1.4 (Jun 27 2014 - 14:56:49)&quot;</td>
</tr>
<tr>
<td>14880</td>
<td>0x3A20</td>
<td>CRC32 polynomial table, big endian</td>
</tr>
<tr>
<td>16176</td>
<td>0x3F30</td>
<td>uImage header, header size: 64 bytes, header CRC: 0x3AC66E95, created: 2014-06-27 06:56:50, image size: 34587 bytes, Data Address: 0x80010000, Entry Point: 0x80010000, data CRC: 0xDF2DBA0B, OS: Linux, CPU: MIPS, image type: Firmware Image, compression type: lzma, image name: &quot;u-boot image&quot;</td>
</tr>
<tr>
<td>16240</td>
<td>0x3F70</td>
<td>LZMA compressed data, properties: 0x5D, dictionary size: 33554432, bytes, uncompressed size: 90000 bytes</td>
</tr>
<tr>
<td>131584</td>
<td>0x20200</td>
<td>TP-Link firmware header, firmware version: 0.0.3, image version: &quot;&quot;, product ID: 0x0, product version: 155254789, kernel load address: 0x0, kernel entry point: 0x-7FFFE000, kernel offset: 3932160, kernel length: 512, rootsfs offset: 837431, rootsfs length: 1048576, bootloader offset: 2883584, bootloader length: 0</td>
</tr>
<tr>
<td>132096</td>
<td>0x20400</td>
<td>LZMA compressed data, properties: 0x5D, dictionary size: 33554432, bytes, uncompressed size: 2388212 bytes</td>
</tr>
<tr>
<td>1180160</td>
<td>0x120200</td>
<td>Squashfs filesystem, little endian, version 4.0, compression:lzma, size: 2548511 bytes, 536 inodes, blocksize: 131072 bytes, created: 2014-06-27 07:06:52</td>
</tr>
</tbody>
</table>

Indeed: there are some stuff at the beginning, but two large LZMA compressed blocks are started at 0x20400 and 0x120200. These are roughly addresses we have seen in Mathematica. Oh, and by the way, binwalk can show entropy information as well (-E option):

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>HEXADECIMAL</th>
<th>ENTROPY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0</td>
<td>Falling entropy edge (0.419187)</td>
</tr>
<tr>
<td>16384</td>
<td>0x4000</td>
<td>Rising entropy edge (0.988639)</td>
</tr>
<tr>
<td>51200</td>
<td>0xC800</td>
<td>Falling entropy edge (0.000000)</td>
</tr>
<tr>
<td>133120</td>
<td>0x20800</td>
<td>Rising entropy edge (0.987596)</td>
</tr>
<tr>
<td>968704</td>
<td>0xEC800</td>
<td>Falling entropy edge (0.508720)</td>
</tr>
<tr>
<td>1181696</td>
<td>0x120800</td>
<td>Rising entropy edge (0.989615)</td>
</tr>
<tr>
<td>3727360</td>
<td>0x38E000</td>
<td>Falling entropy edge (0.732390)</td>
</tr>
</tbody>
</table>
Rising edges are corresponding to rising edges of block on our graph. Falling edges are the points where empty lacunas are started.

Binwalk can also generate PNG graphs (-E -J):

What can we say about lacunas? By looking in hex editor, we see that these are just filled with 0xFF bytes. Why developers put them? Perhaps, because they weren’t able to calculate precise compressed blocks sizes, so they allocated space for them with some reserve.

**Notepad**

Another example is notepad.exe I’ve picked in Windows 8.1:
There is cavity at $0x19000$ (absolute file offset). I’ve opened the executable file in hex editor and found imports table there (which has lower entropy than x86-64 code in the first half of graph).

There are also high entropy block started at $0x20000$:
In hex editor I can see PNG file here, embedded in the PE file resource section (it is a large image of notepad icon). PNG files are compressed, indeed.

**Unnamed dashcam**

Now the most advanced example in this part is the firmware of some unnamed dashcam I've received from a friend:
The cavity at the very beginning is an English text: debugging messages. I checked various ISAs and I found that the first third of the whole file (with the text segment inside) is in fact MIPS (little-endian) code.

For instance, this is very distinctive MIPS function epilogue:

```
44 5F 53 50 49 5F 46 57 32 0 0 0 53 45 4D 49 44 5F 53 50 49 5F 50 53 32 0 0 0 53 45 4D 49 44 5F 53 50 49 5F 50 53 32 0 0 0 53 45 4D 49 44 5F 53 50 49 5F 50 53 32 0 0 0 53 45 4D 49
```

From our graph we can see that MIPS code has entropy of 5-6 bits per byte. Indeed, I once measured various ISAs entropy and I've got these values:

- x86: .text section of ntoskrnl.exe file from Windows 2003: 6.6
• x64: .text section of ntoskrnl.exe file from Windows 7 x64: 6.5
• ARM (thumb mode), Angry Birds Classic: 7.05
• ARM (ARM mode) Linux Kernel 3.8.0: 6.03
• MIPS (little endian), .text section of user32.dll from Windows NT 4: 6.09

So the entropy of executable code is higher than of English text, but still can be compressed.

Now the second third is started at 0xF5000. I don’t know what this is. I tried different ISAs but without success. The entropy of the block is looks even steadier than for executable one. Maybe some kind of data?

There is also a spike at ≈ 0x213000. I checked it in hex editor and I found JPEG file there (which, of course, compressed)! I also don’t know what is at the end. Let’s try Binwalk for this file:

```bash
% binwalk FW96650A.bin
```

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>HEXADECIMAL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>167698</td>
<td>0x28F12</td>
<td>Unix path: /15/20/24/25/30/60/120/240fps can be served..</td>
</tr>
<tr>
<td>280286</td>
<td>0x446DE</td>
<td>Copyright string: “Copyright (c) 2012 Novatek Microelectronic Corp.”</td>
</tr>
<tr>
<td>2169199</td>
<td>0x21196F</td>
<td>JPEG image data, JFIF standard 1.01</td>
</tr>
<tr>
<td>2300847</td>
<td>0x231BAF</td>
<td>MySQL MISAM compressed data file Version 3</td>
</tr>
</tbody>
</table>

% binwalk -E FW96650A.bin

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>HEXADECIMAL</th>
<th>ENTROPY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0</td>
<td>Falling entropy edge (0.579792)</td>
</tr>
<tr>
<td>2170880</td>
<td>0x212000</td>
<td>Rising entropy edge (0.967373)</td>
</tr>
<tr>
<td>2267136</td>
<td>0x229800</td>
<td>Falling entropy edge (0.802974)</td>
</tr>
<tr>
<td>2426880</td>
<td>0x250800</td>
<td>Falling entropy edge (0.846639)</td>
</tr>
<tr>
<td>2490368</td>
<td>0x260000</td>
<td>Falling entropy edge (0.849804)</td>
</tr>
<tr>
<td>2560000</td>
<td>0x271000</td>
<td>Rising entropy edge (0.974340)</td>
</tr>
<tr>
<td>2574336</td>
<td>0x274800</td>
<td>Rising entropy edge (0.970958)</td>
</tr>
<tr>
<td>2588672</td>
<td>0x278000</td>
<td>Falling entropy edge (0.763507)</td>
</tr>
<tr>
<td>2592768</td>
<td>0x279000</td>
<td>Rising entropy edge (0.951883)</td>
</tr>
<tr>
<td>2596864</td>
<td>0x27A000</td>
<td>Falling entropy edge (0.712814)</td>
</tr>
<tr>
<td>2600960</td>
<td>0x27B000</td>
<td>Rising entropy edge (0.968167)</td>
</tr>
<tr>
<td>2607104</td>
<td>0x27C800</td>
<td>Rising entropy edge (0.958582)</td>
</tr>
<tr>
<td>2609152</td>
<td>0x27D000</td>
<td>Rising entropy edge (0.760989)</td>
</tr>
<tr>
<td>2654288</td>
<td>0x286000</td>
<td>Rising entropy edge (0.954127)</td>
</tr>
<tr>
<td>2670592</td>
<td>0x28C000</td>
<td>Rising entropy edge (0.967883)</td>
</tr>
<tr>
<td>2676736</td>
<td>0x2BD800</td>
<td>Rising entropy edge (0.975779)</td>
</tr>
<tr>
<td>2684928</td>
<td>0x2BF800</td>
<td>Falling entropy edge (0.744369)</td>
</tr>
</tbody>
</table>

Yes, it found JPEG file and even MySQL data! But I’m not sure if it’s true—I didn’t check it yet.

It’s also interesting to try clusterization in Mathematica:
Here is an example of how Mathematica grouped various entropy values into distinctive groups. Indeed, there is something credible. Blue dots in range of 5.0-5.5 are supposedly related to English text. Yellow dots in 5.5-6 are MIPS code. A lot of green dots in 6.0-6.5 is the unknown second third. Orange dots close to 8.0 are related to compressed JPEG file. Other orange dots are supposedly related to the end of the firmware (unknown to us data).

Links

Binary files used in this part:
Wolfram Mathematica notebook file:
(all cells must be evaluated to start things working).

9.2.2 Conclusion

Information entropy can be used as a quick-n-dirty method for inspecting unknown binary files. In particular, it is a very quick way to find compressed/encrypted pieces of data. Someone say it's possible to find RSA\(^5\) (and other asymmetric cryptographic algorithms) public/private keys in executable code (keys have high entropy as well), but I didn't try this myself.

9.2.3 Tools

Handy Linux ent utility to measure entropy of a file\(^6\).


radare2 framework has \#entropy command for this.

A tool for IDA: IDAtropy\(^7\).

---

\(^5\)Rivest Shamir Adleman
\(^6\)http://www.fourmilab.ch/random/
\(^7\)https://github.com/danigargu/IDAtropy
9.2.4 A word about primitive encryption like XORing

It’s interesting that simple XOR encryption doesn’t affect entropy of data. I’ve shown this in Norton Guide example in the book (9.1.2 on page 910).

Generalizing: encryption by substitution cipher also doesn’t affect entropy of data (and XOR can be viewed as substitution cipher). The reason of that is because entropy calculation algorithm view data on byte-level. On the other hand, the data encrypted by 2 or 4-byte XOR pattern will result in another level of entropy.

Nevertheless, low entropy is usually a good sign of weak amateur cryptography (which is also used in license keys/files, etc.).

9.2.5 More about entropy of executable code

It is quickly noticeable that probably a biggest source of high-entropy in executable code are relative offsets encoded in opcodes. For example, these two consequent instructions will have different relative offsets in their opcodes, while they are in fact pointing to the same function:

```
function proc
...
function endp
...
CALL function
...
CALL function
```

Ideal executable code compressor would encode information like this: *there is a CALL to a “function” at address X and the same CALL at address Y* without necessity to encode address of the *function* twice.

To deal with this, executable compressors are sometimes able to reduce entropy here. One example is UPX: [http://sourceforge.net/p/upx/code/ci/default/tree/doc/filter.txt](http://sourceforge.net/p/upx/code/ci/default/tree/doc/filter.txt).

9.2.6 PRNG

When I run GnuPG to generate new private (secret) key, it asking for some entropy ...

```
We need to generate a lot of random bytes. It is a good idea to perform some other action (type on the keyboard, move the mouse, utilize the disks) during the prime generation; this gives the random number generator a better chance to gain enough entropy.

Not enough random bytes available. Please do some other work to give the OS a chance to collect more entropy! (Need 169 more bytes)
```

This means that good a PRNG produces long high-entropy results, and this is what the secret asymmetrical cryptographical key needs. But CPRNG® is tricky (because computer is highly deterministic device itself), so the GnuPG asking for some additional randomness from the user.

9.2.7 More examples

Here is a case where I try to calculate entropy of some blocks with unknown contents: 8.9 on page 840.

9.2.8 Entropy of various files

Entropy of random data is close to 8:

```
% dd bs=1M count=1 if=/dev/urandom | ent
Entropy = 7.999803 bits per byte.
```

This means, almost all available space inside of byte is filled with information.

256 bytes in range of 0..255 gives exact value of 8:

®Cryptographically secure PseudoRandom Number Generator
#!/usr/bin/env python
import sys
for i in range(256):
    sys.stdout.write(chr(i))

% python 1.py | ent
Entropy = 8.000000 bits per byte.

Order of bytes doesn’t matter. This means, all available space inside of byte is filled.

Entropy of any block filled with zero bytes is 0:

% dd bs=1M count=1 if=/dev/zero | ent
Entropy = 0.000000 bits per byte.

Entropy of a string consisting of a single (any) byte is 0:

% echo -n "aaaaaaaaaaaaaaaaaaa" | ent
Entropy = 0.000000 bits per byte.

Entropy of base64 string is the same as entropy of source data, but multiplied by $\frac{3}{4}$. This is because base64 encoding uses 64 symbols instead of 256.

% dd bs=1M count=1 if=/dev/urandom | base64 | ent
Entropy = 6.022068 bits per byte.

Perhaps, 6.02 is slightly bigger than 6 because padding symbols (=) spoils our statistics for a little.

Uuencode also uses 64 symbols:

% dd bs=1M count=1 if=/dev/urandom | uuencode - | ent
Entropy = 6.013162 bits per byte.

This means, any base64 and Uuencode strings can be transmitted using 6-bit bytes or characters.

Any random information in hexadecimal form has entropy of 4 bits per byte:

% openssl rand -hex $\$$(( 2**16 )) | ent
Entropy = 4.000013 bits per byte.

Entropy of randomly picked English language text from Gutenberg library has entropy ≈ 4.5. The reason of this is because English texts uses mostly 26 symbols, and $\log_2(26) = 4.7$, i.e., you would need 5-bit bytes to transmit uncompressed English texts, that would be enough (it was indeed so in teletype era).

Randomly chosen Russian language text from http://lib.ru library is F.M.Dostoevsky “Idiot”⁹, internally encoded in CP1251 encoding.

And this file has entropy of ≈ 4.98. Russian language has 33 characters, and $\log_2(33) = 5.04$. But it has unpopular and rare “ё” character. And $\log_2(32) = 5$ (Russian alphabet without this rare character)—now this close to what we’ve got.

However, the text we studying uses “ё” letter, but, probably, it’s still rarely used there.

The very same file transcoded from CP1251 to UTF-8 gave entropy of ≈ 4.23. Each Cyrillic character encoded in UTF-8 is usually encoded as a pair, and the first byte is always one of: 0xD0 or 0xD1. Perhaps, this caused bias.

Let’s generate random bits and output them as “T” and “F” characters:

# !/usr/bin/env python
import random, sys

rt=""
for i in range(102400):
    if random.randint(0,1)==1:
        rt=rt+"T"
    else:
        rt=rt+"F"
print rt

⁹http://az.lib.ru/d/dostoevskij_f_m/text_0070.shtml
Entropy is very close to 1 (i.e., 1 bit per byte).

Let's generate random decimal digits:

```python
#!/usr/bin/env python
import random, sys
rt ="
for i in range(102400):
    rt+="%d" % random.randint(0,9)
print rt
```

Entropy will be close to 3.32, indeed, this is \( \log_2(10) \).

### 9.2.9 Making lower level of entropy

The author of these lines once saw a software which stored each byte of encrypted data in 3 bytes: each has \( \approx \frac{\text{byte}}{3} \) value, so reconstructing encrypted byte back involving summing up 3 consecutive bytes. Looks absurdly.

But some people say this was done in order to conceal the very fact the data has something encrypted inside: measuring entropy of such block will show much lower level of it.

### 9.3 Millenium game save file

The “Millenium Return to Earth” is an ancient DOS game (1991), that allows you to mine resources, build ships, equip them and send them on other planets, and so on\(^{10}\).

Like many other games, it allows you to save all game state into a file.

Let's see if we can find something in it.

---

\(^{10}\)It can be downloaded for free [here](#)
So there is a mine in the game. Mines at some planets work faster, or slower on others. The set of resources is also different.

Here we can see what resources are mined at the time:

Figure 9.14: Mine: state 1

Let’s save a game state. This is a file of size 9538 bytes.

Let’s wait some “days” here in the game, and now we’ve got more resources from the mine:
Now let's try to just do binary comparison of the save files using the simple DOS/Windows FC utility:

```bash
$> FC /b 2200save.i.v1 2200SAVE.I.V2

Comparing files 2200save.i.v1 and 2200SAVE.I.V2
00000016: 0D 04
00000017: 03 04
0000001C: 1F 1E
00000146: 27 3B
00000BDA: 0E 16
00000BDC: 66 9B
00000BDE: 0E 16
00000BE0: 0E 16
00000BE6: DB 4C
00000BE7: 00 01
00000BEE: 99 E8
00000BEC: 63 C7
00000BFB: A8 2B
00000BFD: 98 1B
000000C0: A8 2D
000000C0: A8 2D
000000C7: DB 5B
000000C9: E4 A4
000000CD: 3B B8
000000CF: E8 6B
...```

The output is incomplete here, there are more differences, but we will cut result to show the most interesting.

In the first state, we have 14 “units” of hydrogen and 102 “units” of oxygen.
We have 22 and 155 “units” respectively in the second state. If these values are saved into the save file, we would see this in the difference. And indeed we do. There is 0x0E (14) at position 0xBDA and this value is 0x16 (22) in the new version of the file. This is probably hydrogen. There is 0x66 (102) at position 0xBDC in the old version and 0x9B (155) in the new version of the file. This seems to be the oxygen.

Both files are available on the website for those who wants to inspect them (or experiment) more: beginners.re.
Here is the new version of file opened in Hiew, we marked the values related to the resources mined in the game:

![Figure 9.16: Hiew: state 1](image)

Let's check each of them.
These are clearly 16-bit values: not a strange thing for 16-bit DOS software where the `int` type has 16-bit width.
Let's check our assumptions. We will write the 1234 (0x4D2) value at the first position (this must be hydrogen):

![Screen capture of Hiew software showing hexadecimal values](image1)

**Figure 9.17:** Hiew: let’s write 1234 (0x4D2) there

Then we will load the changed file in the game and took a look at mine statistics:

![Screenshot of game showing resource management](image2)

**Figure 9.18:** Let’s check for hydrogen value

So yes, this is it.
Now let’s try to finish the game as soon as possible, set the maximal values everywhere:

![Hiew: let’s set maximal values](image1)

Figure 9.19: Hiew: let’s set maximal values

0xFFFF is 65535, so yes, we now have a lot of resources:

![All resources are 65535 (0xFFFF) indeed](image2)

Figure 9.20: All resources are 65535 (0xFFFF) indeed
Let's skip some “days” in the game and oops! We have a lower amount of some resources:

Figure 9.21: Resource variables overflow

That’s just overflow.
The game’s developer supposedly didn’t think about such high amounts of resources, so there are probably no overflow checks, but the mine is “working” in the game, resources are added, hence the overflows. Apparently, it is a bad idea to be that greedy.

There are probably a lot of more values saved in this file.

So this is very simple method of cheating in games. High score files often can be easily patched like that.

More about files and memory snapshots comparing: 5.10.2 on page 717.

9.4  *fortune* program indexing file

(This part was first appeared in my blog at 25-Apr-2015.)

*fortune* is well-known UNIX program which shows random phrase from a collection. Some geeks are often set up their system in such way, so *fortune* can be called after logging on. *fortune* takes phrases from the text files laying in */usr/share/games/fortunes* (as of Ubuntu Linux). Here is example (“fortunes” text file):

```
A day for firm decisions!!!!! Or is it?
%
A few hours grace before the madness begins again.
%
A gift of a flower will soon be made to you.
%
A long-forgotten loved one will appear soon.
%
Buy the negatives at any price.
%
A tall, dark stranger will have more fun than you.
```
So it is just phrases, sometimes multiline ones, divided by percent sign. The task of fortune program is to find random phrase and to print it. In order to achieve this, it must scan the whole text file, count phrases, choose random and print it. But the text file can get bigger, and even on modern computers, this naive algorithm is a bit uneconomical to computer resources. The straightforward way is to keep binary index file containing offset of each phrase in text file. With index file, fortune program can work much faster: just to choose random index element, take offset from there, set offset in text file and read phrase from it. This is actually done in fortune program. Let’s inspect what is in its index file inside (these are .dat files in the same directory) in hexadecimal editor. This program is open-source of course, but intentionally, I will not peek into its source code.

Without any special aid we could see that there are four 4-byte elements on each 16-byte line. Perhaps, it’s our index array. I’m trying to load the whole file in Wolfram Mathematica as 32-bit integer array:

```plaintext
In[] := BinaryReadList["c:\tmp1\fortunes.dat", "UnsignedInteger32"]
```

```
Out[] = 
{33554432, 2936078336, 3137339392, 251658240, 0, 37, 0, 
721420288, 1610612736, 2399141888, 3741319168, 335609856, 1208025088, 
2868896972, 3588825216, 537001984, 398986816, 2046951424, 
3053260408, 67305472, 174502780, 1023606784, 1047213568, 3056831360, 168230912, 
1392967680, 2584150016, 4161208320, 654835712, 1493696512, 
2323557312, 3288858624, 3775397888, 4178051072, 
```

Nope, something wrong. Numbers are suspiciously big. But let’s back to od output: each 4-byte element has two zero bytes and two non-zero bytes, so the offsets (at least at the beginning of the file) are 16-bit at maximum. Probably different endianness is used in the file? Default endianness in Mathematica is little-endian, as used in Intel CPUs. Now I’m changing it to big-endian:

```plaintext
In[] := BinaryReadList["c:\tmp1\fortunes.dat", "UnsignedInteger32", 
ByteOrdering -> 1]
```

950
Yes, this is something readable. I choose random element (3066) which is 0xBFA in hexadecimal form. I'm opening 'fortunes' text file in hex editor, I'm setting 0xBFA as offset and I see this phrase:

```
% od -t x1 -c --skip-bytes=0xbfa --address-radix=x fortunes
000bfa 44 6f 20 77 68 61 74 20 63 6f 6d 65 73 20 6e 61
  D o w h a t c o m e s n a
000c0a 74 75 72 61 6c 6c 79 2e 20 20 53 65 65 74 68 65
  t u r a l l y . S e e t h e
000c1a 20 61 6e 64 20 66 75 6d 65 20 61 6e 64 20 74 68
  a n d f u m e a n d t h
```

Or:

Do what comes naturally. See the and fume and throw a tantrum.

%

Other offset are also can be checked, yes, they are valid offsets.

I can also check in Mathematica that each subsequent element is bigger than previous. I.e., elements of array are ascending. In mathematical lingo, this is called **strictly increasing monotonic function**.

As we can see, except of the very first 6 values (which is probably belongs to index file header), all numbers are in fact length of all text phrases (offset of the next phrase minus offset of the current phrase is in fact length of the current phrase).

It's very important to keep in mind that bit-endiannes can be confused with incorrect array start. Indeed, from `od` output we see that each element started with two zeros. But when shifted by two bytes in either side, we can interpret this array as little-endian:

As we can see, except of the very first 6 values (which is probably belongs to index file header), all numbers are in fact length of all text phrases (offset of the next phrase minus offset of the current phrase is in fact length of the current phrase).

It's very important to keep in mind that bit-endiannes can be confused with incorrect array start. Indeed, from `od` output we see that each element started with two zeros. But when shifted by two bytes in either side, we can interpret this array as little-endian:
If we would interpret this array as little-endian, the first element is 0x4801, second is 0x7C01, etc. High 8-bit part of each of these 16-bit values are seems random to us, and the lowest 8-bit part is seems ascending.

But I’m sure that this is big-endian array, because the very last 32-bit element of the file is big-endian (00 00 5fc4 here):

```
% od -t x1 --address-radix=x fortunes.dat
...
000660 00 00 59 0d 00 00 59 55 00 00 59 7d 00 00 59 b5
000670 00 00 59 f4 00 00 5a 35 00 00 5a 5e 00 00 5a 9c
000680 00 00 5a cb 00 00 5a f4 00 00 5b 1f 00 00 5b 3d
000690 00 00 5b 68 00 00 5b ab 00 00 5b f9 00 00 5c 49
0006a0 00 00 5c ae 00 00 5c eb 00 00 5d 34 00 00 5d 7a
0006b0 00 00 5d a3 00 00 5d f5 00 00 5e 3a 00 00 5e 67
0006c0 00 00 5e a8 00 00 5e ce 00 00 5e f7 00 00 5f 30
0006d0 00 00 5f 82 00 00 5f c4
0006d8
```

Perhaps, fortune program developer had big-endian computer or maybe it was ported from something like it.

OK, so the array is big-endian, and, judging by common sense, the very first phrase in the text file must be started at zeroth offset. So zero value should be present in the array somewhere at the very beginning. We’ve got couple of zero elements at the beginning. But the second is most appealing: 43 is going right after it and 43 is valid offset to valid English phrase in the text file.

The last array element is 0x5FC4, and there are no such byte at this offset in the text file. So the last array element is pointing behind the end of file. It’s supposedly done because phrase length is calculated as difference between offset to the current phrase and offset to the next phrase. This can be faster than traversing phrase string for percent character. But this wouldn’t work for the last element. So the dummy element is also added at the end of array.

So the first 6 32-bit integer values are supposedly some kind of header.

Oh, I forgot to count phrases in text file:

```
% cat fortunes | grep % | wc -l
432
```

The number of phrases can be present in index, but may be not. In case of very simple index files, number of elements can be easily deduced from index file size. Anyway, there are 432 phrases in the text file. And we see something very familiar at the second element (value 431). I’ve checked other files (literature.dat and riddles.dat in Ubuntu Linux) and yes, the second 32-bit element is indeed number of phrases minus 1. Why minus 1? Perhaps, this is not number of phrases, but rather the number of the last phrase (starting at zero)?

And there are some other elements in the header. In Mathematica, I’m loading each of three available files and I’m taking a look on the header:
I have no idea what other values mean, except the size of index file. Some fields are the same for all files, some are not. From my own experience, there could be:

- file signature;
- file version;
- checksum;
- some flags;
- maybe even text language identifier;
- text file timestamp, so the `fortune` program will regenerate index file if a user modified text file.

For example, Oracle .SYM files (9.5 on the next page) which contain symbols table for DLL files, also contain timestamp of corresponding DLL file, so to be sure it is still valid.

On the other hand, text file and index file timestamps can gone out of sync after archiving/unarchiving/installing/deploying/etc.

But there are no timestamp, in my opinion. The most compact way of representing date and time is UNIX time value, which is big 32-bit number. We don’t see any of such here. Other ways of representation are even less compact.

So here is algorithm, how `fortune` supposedly works:

- take number of last phrase from the second element;
- generate random number in range of 0..number_of_last_phrase;
- find corresponding element in array of offsets, take also following offset;
- output to `stdout` all characters from the text file starting at the offset until the next offset minus 2 (so to ignore terminating percent sign and character of the following phrase).

### 9.4.1 Hacking

Let's try to check some of our assumptions. I will create this text file under the path and name `/usr/share/games/fortunes/fortunes`

```
Phrase one.
%
Phrase two.
%
```

Then this `fortunes.dat` file. I take header from the original `fortunes.dat`, I changed second field (count of all phrases) to zero and I left two elements in the array: 0 and 0x1c, because the whole length of the text `fortunes` file is 28 (0x1c) bytes:
Now I run it:

```
% /usr/games/fortune
fortune: no fortune found
```

Something wrong. Let’s change the second field to 1:

```
% od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 01 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 1c
```

Now it works. It’s always shows only the first phrase:

```
% /usr/games/fortune
Phrase one.
```

Hmmm. Let’s leave only one element in array (0) without terminating one:

```
% od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 01 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 1c
```

Fortune program always shows only first phrase.

From this experiment we got to know that percent sign in text file is parsed and the size is not calculated as I deduced before, perhaps, even terminal array element is not used. However, it still can be used. And probably it was used in past?

### 9.4.2 The files

For the sake of demonstration, I still didn’t take a look in `fortune` source code. If you want to try to understand meaning of other values in index file header, you may try to achieve it without looking into source code as well. Files I took from Ubuntu Linux 14.04 are here: [http://beginners.re/examples/fortune/](http://beginners.re/examples/fortune/), hacked files are also here.

Oh, and I took the files from x64 version of Ubuntu, but array elements are still has size of 32 bit. It is because `fortune` text files are probably never exceeds 4GiB size. But if it will, all elements must have size of 64 bit so to be able to store offset to the text file larger than 4GiB.

For impatient readers, the source code of `fortune` is here: [https://launchpad.net/ubuntu/+source/fortune-mod/1:1.99.1-3.1ubuntu4](https://launchpad.net/ubuntu/+source/fortune-mod/1:1.99.1-3.1ubuntu4).

### 9.5 Oracle RDBMS: .SYM-files

When an Oracle RDBMS process experiences some kind of crash, it writes a lot of information into log files, including stack trace, like this:

```
----- Call Stack Trace -----  
calling  call  entry  argument values in hex
location type  point  (?) means dubious value
-------------------- -------- -------------------- ----------------------------
_kqvrow() 00000000
_opifch2()+2729 CALLptr 00000000 23D4B914 E47F264 1F19AE2
         1F19AE2
_-opioal8()+2832 CALLrel opifch2() 09 5 EB1C74
_opiodr()+1248 CALLReg 00000000 5E 1C EB1F0A0
         1C EB1F0A0 0
_ttcpip()+1051 CALLReg 00000000 5E 1C EB1F0A0 0
         C96C0A0 5E EB1F0A0 0 EB1ED30
_opitsk()+1404 CALL?? 00000000
_       00000000
opiino()+980 CALLrel opitsk() 0 0

11Gibibyte
```
But of course, Oracle RDBMS’s executables must have some kind of debug information or map files with symbol information included or something like that.

Windows NT Oracle RDBMS has symbol information in files with .SYM extension, but the format is proprietary. (Plain text files are good, but needs additional parsing, hence offer slower access.)

Let’s see if we can understand its format.

We will pick the shortest orawtc8.sym file that comes with the orawtc8.dll file in Oracle 8.1.7 12.

---

12 We can chose an ancient Oracle RDBMS version intentionally due to the smaller size of its modules.
By comparing the file with other .SYM files, we can quickly see that OSYM is always header (and footer), so this is maybe the file's signature.

We also see that basically, the file format is: OSYM + some binary data + zero delimited text strings + OSYM. The strings are, obviously, function and global variable names.
We will mark the OSYM signatures and strings here:

![Hiew: orawtc8.sym]

We will mark the whole strings block (except the trailing OSYM signatures) and put it into a separate file. Then we run UNIX `strings` and `wc` utilities to count the text strings:

```
strings strings_block | wc -l
```

66

So there are 66 text strings. Please note that number.

We can say, in general, as a rule, the number of anything is often stored separately in binary files. It’s indeed so, we can find the 66 value (0x42) at the file’s start, right after the OSYM signature:

```
$ hexdump -C orawtc8.sym
```

Of course, 0x42 here is not a byte, but most likely a 32-bit value packed as little-endian, hence we see 0x42 and then at least 3 zero bytes.

Why do we believe it’s 32-bit? Because, Oracle RDBMS’s symbol files may be pretty big.

The oracle.sym file for the main oracle.exe (version 10.2.0.4) executable contains 0x3A38E (238478) symbols. A 16-bit value isn’t enough here.

We can check other .SYM files like this and it proves our guess: the value after the 32-bit OSYM signature always reflects the number of text strings in the file.

957
It’s a general feature of almost all binary files: a header with a signature plus some other information about the file.

Now let’s investigate closer what this binary block is.

Using Hiew again, we put the block starting at address 8 (i.e., after the 32-bit count value) ending at the strings block, into a separate binary file.
Let's see the binary block in Hiew:

Figure 9.24: Binary block

There is a clear pattern in it.
We will add red lines to divide the block:

Figure 9.25: Binary block patterns

Hiew, like almost any other hexadecimal editor, shows 16 bytes per line. So the pattern is clearly visible: there are 4 32-bit values per line.

The pattern is visually visible because some values here (till address 0x104) are always in 0x1000xxxx form, started with 0x10 and zero bytes.

Other values (starting at 0x108) are in 0x0000xxxx form, so always started with two zero bytes.

Let’s dump the block as an array of 32-bit values:

Listing 9.9: first column is address

$ od -v -t x4 binary_block
00000000 10001000 10001080 100010f0 10001150
00000200 10001110 10001180 100011d0 10001370
00000400 10001540 10001550 10001560 10001580
00000600 100015a0 100015ac 100015b2 100015ca
00001000 100015d0 100015e0 100016b0 10001760
00001400 10001766 1000176c 10001780 100017b0
00001600 100017d0 100017e0 10001810 10001816
00002000 10002000 10002064 10002088 100020c0
00002200 10002010 10002014 10002018 1000201c
There are 132 values, that's 66*2. Probably, there are two 32-bit values for each symbol, but maybe there are two arrays? Let's see.

Values starting with 0x1000 may be addresses.

This is a .SYM file for a DLL after all, and the default base address of win32 DLLs is 0x10000000, and the code usually starts at 0x10001000.

When we open the orawtc8.dll file in IDA, the base address is different, but nevertheless, the first function is:

```
.text:60351000 sub_60351000 proc near
.text:60351000
.text:60351000 arg_0 = dword ptr 8
.text:60351000 arg_4 = dword ptr 0Ch
.text:60351000 arg_8 = dword ptr 10h
.text:60351000
.text:60351000 push ebp
.text:60351001 mov ebp, esp
.text:60351003 mov eax, dword_60353014
.text:60351008 cmp eax, 0xFFFFFFFFh
.text:6035100B jnz short loc_6035104F
.text:6035100D mov ecx, hModule
.text:60351013 xor eax, eax
.text:60351015 cmp ecx, 0xFFFFFFFFh
.text:60351018 mov dword_60353014, eax
.text:6035101D jnz short loc_60351031
.text:6035101F call sub_603510F0
.text:60351024 mov ecx, eax
.text:60351026 mov eax, dword_60353014
.text:6035102B mov hModule, ecx
.text:60351031.text:60351031 loc_60351031: ; CODE XREF: sub_60351000+1D
.text:60351031 test ecx, ecx
.text:60351033 jbe short loc_6035104F
.text:60351035 push offset ProcName ; "ax_reg"
.text:6035103A push ecx ; hModule
.text:6035103B call ds:GetProcAddress ...
```

Wow, "ax_reg" string sounds familiar.

It's indeed the first string in the strings block! So the name of this function seems to be “ax_reg”.

The second function is:
The "ax_unreg" string is also the second string in the strings block!

The starting address of the second function is 0x60351080, and the second value in the binary block is 10001080. So this is the address, but for a DLL with the default base address.

We can quickly check and be sure that the first 66 values in the array (i.e., the first half of the array) are just function addresses in the DLL, including some labels, etc. Well, what's the other part of array then? The other 66 values that start with 0x0000? These seem to be in range [0...0x3F8]. And they do not look like bitfields: the series of numbers is increasing.

The last hexadecimal digit seems to be random, so, it's unlikely the address of something (it would be divisible by 4 or maybe 8 or 0x10 otherwise).

Let's ask ourselves: what else Oracle RDBMS's developers would save here, in this file?

Quick wild guess: it could be the address of the text string (function name).

It can be quickly checked, and yes, each number is just the position of the first character in the strings block.

This is it! All done.

We will write an utility to convert these .SYM files into IDA script, so we can load the .idc script and it sets the function names:

```c
#include <stdio.h>
#include <stdint.h>
#include <io.h>
#include <assert.h>
#include <malloc.h>
#include <fcntl.h>
#include <string.h>

int main (int argc, char *argv[])
{
    uint32_t sig, cnt, offset;
    uint32_t *d1, *d2;
    int    h, i, remain, file_len;
    char   *d3;
    uint32_t array_size_in_bytes;

    assert (argv[1]); // file name
    assert (argv[2]); // additional offset (if needed)
```
```c
// additional offset
assert (sscanf (argv[2], "%X", &offset)==1);

// get file length
assert ((f=fopen (argv[1], _O_RDONLY | _O_BINARY, 0))!=-1);
assert ((file_len=lseek (h, 0, SEEK_END))!=-1);
assert (lseek (h, 0, SEEK_SET)!=-1);

// read signature
assert (read (h, &sig, 4)==4);
// read count
assert (sig==0x4D59534F); // OSYM

// skip timedatestamp (for 11g)
// lseek (h, 4, 1);

array_size_in_bytes=cnt*sizeof(uint32_t);

// load symbol addresses array
d1=(uint32_t*)malloc (array_size_in_bytes);
assert (d1);
assert (read (h, d1, array_size_in_bytes)==array_size_in_bytes);

// load string offsets array
d2=(uint32_t*)malloc (array_size_in_bytes);
assert (d2);
assert (read (h, d2, array_size_in_bytes)==array_size_in_bytes);

// calculate strings block size
remain=file_len-(8+4)-(cnt*8);

// load strings block
assert (d3=(char*)malloc (remain));
assert (read (h, d3, remain)==remain);

printf ("#include <idc.idc>\n\n");
printf ("static main() {\n");

for (i=0; i<cnt; i++)
    printf ("\tMakeName(0x%08X, "%s"), offset + d1[i], &d3[d2[i]])\n",

printf ("}\n");

close (h);
free (d1); free (d2); free (d3);
}

Here is an example of its work:

```c
#include <idc.idc>

static main() {  
    MakeName(0x60351000, "_ax_reg");
    MakeName(0x60351080, "_ax_unreg");
    MakeName(0x603510F0, "_loaddll");
    MakeName(0x60351150, "_wtcsrin0");
    MakeName(0x60351160, "_wtcsrin");
    MakeName(0x603511D0, "_wtclkm");
    MakeName(0x60351370, "_wtcstu");
    ...
}
```

The example files were used in this example are here: beginners.re.
Oh, let’s also try Oracle RDBMS for win64. There has to be 64-bit addresses instead, right?
The 8-byte pattern is visible even easier here:

So yes, all tables now have 64-bit elements, even string offsets!
The signature is now 0SYMAM64, to distinguish the target platform, apparently.
This is it!
Here is also library which has functions to access Oracle RDBMS.SYM-files: GitHub.

9.6 Oracle RDBMS: .MSB-files

When working toward the solution of a problem, it always helps if you know the answer.

Murphy’s Laws, Rule of Accuracy

This is a binary file that contains error messages with their corresponding numbers. Let’s try to understand its format and find a way to unpack it.
There are Oracle RDBMS error message files in text form, so we can compare the text and packed binary files ¹³.

This is the beginning of the ORAUS.MSG text file with some irrelevant comments stripped:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Text Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>&quot;normal, successful completion&quot;</td>
</tr>
<tr>
<td>00017</td>
<td>&quot;unique constraint (%s.%s) violated&quot;</td>
</tr>
<tr>
<td>00018</td>
<td>&quot;maximum number of sessions exceeded&quot;</td>
</tr>
<tr>
<td>00019</td>
<td>&quot;maximum number of session licenses exceeded&quot;</td>
</tr>
<tr>
<td>00020</td>
<td>&quot;maximum number of processes (%s) exceeded&quot;</td>
</tr>
<tr>
<td>00021</td>
<td>&quot;session attached to some other process; cannot switch session&quot;</td>
</tr>
<tr>
<td>00022</td>
<td>&quot;invalid session ID; access denied&quot;</td>
</tr>
<tr>
<td>00023</td>
<td>&quot;session references process private memory; cannot detach session&quot;</td>
</tr>
<tr>
<td>00024</td>
<td>&quot;logins from more than one process not allowed in single-process mode&quot;</td>
</tr>
<tr>
<td>00025</td>
<td>&quot;failed to allocate %s&quot;</td>
</tr>
<tr>
<td>00026</td>
<td>&quot;missing or invalid session ID&quot;</td>
</tr>
<tr>
<td>00027</td>
<td>&quot;cannot kill current session&quot;</td>
</tr>
<tr>
<td>00028</td>
<td>&quot;your session has been killed&quot;</td>
</tr>
<tr>
<td>00029</td>
<td>&quot;session is not a user session&quot;</td>
</tr>
<tr>
<td>00030</td>
<td>&quot;User session ID does not exist.&quot;</td>
</tr>
<tr>
<td>00031</td>
<td>&quot;session marked for kill&quot;</td>
</tr>
</tbody>
</table>

The first number is the error code. The second is perhaps maybe some additional flags.

¹³Open-source text files don’t exist in Oracle RDBMS for every .MSB file, so that’s why we will work on their file format
Now let's open the ORAUS.MSB binary file and find these text strings. And there are:

Figure 9.27: Hiew: first block

We see the text strings (including those from the beginning of the ORAUS.MSG file) interleaved with some binary values. By quick investigation, we can see that main part of the binary file is divided by blocks of size 0x200 (512) bytes.
Let's see the contents of the first block:

![Image](image_url)

Here we see the texts of the first messages errors. What we also see is that there are no zero bytes between the error messages. This implies that these are not null-terminated C strings. As a consequence, the length of each error message must be encoded somehow. Let's also try to find the error numbers. The ORAUS.MSG files starts with these: 0, 1, 17 (0x11), 18 (0x12), 19 (0x13), 20 (0x14), 21 (0x15), 22 (0x16), 23 (0x17), 24 (0x18)... We will find these numbers at the beginning of the block and mark them with red lines. The period between error codes is 6 bytes.

This implies that there are probably 6 bytes of information allocated for each error message.

The first 16-bit value (0xA here or 10) means the number of messages in each block: this can be checked by investigating other blocks. Indeed: the error messages have arbitrary size. Some are longer, some are shorter. But block size is always fixed, hence, you never know how many text messages can be packed in each block.

As we already noted, since these are not null-terminated C strings, their size must be encoded somewhere. The size of the first string “normal, successful completion” is 29 (0x1D) bytes. The size of the second string “unique constraint (%s.%s) violated” is 34 (0x22) bytes. We can’t find these values (0x1D or/and 0x22) in the block.

There is also another thing. Oracle RDBMS has to determine the position of the string it needs to load in the block, right? The first string “normal, successful completion” starts at position 0x1444 (if we count starting at the beginning of the file) or at 0x44 (from the block’s start). The second string “unique constraint
(\%s,\%s) violated” starts at position 0x1461 (from the file’s start) or at 0x61 (from the at the block’s start). These numbers (0x44 and 0x61) are familiar somehow! We can clearly see them at the start of the block.

So, each 6-byte block is:

- 16-bit error number;
- 16-bit zero (maybe additional flags);
- 16-bit starting position of the text string within the current block.

We can quickly check the other values and be sure our guess is correct. And there is also the last “dummy” 6-byte block with an error number of zero and starting position beyond the last error message’s last character. Probably that’s how text message length is determined? We just enumerate 6-byte blocks to find the error number we need, then we get the text string’s position, then we get the position of the text string by looking at the next 6-byte block! This way we determine the string’s boundaries! This method allows to save some space by not saving the text string’s size in the file!

It’s not possible to say it saves a lot of space, but it’s a clever trick.
Let's back to the header of .MSB-file:

Figure 9.29: Hiew: file header

Now we can quickly find the number of blocks in the file (marked by red). We can checked other .MSB-files and we see that it's true for all of them.

There are a lot of other values, but we will not investigate them, since our job (an unpacking utility) is done.

If we have to write a .MSB file packer, we would probably have to understand the meaning of the other values.
There is also a table that came after the header which probably contains 16-bit values:

![Table screenshot]

Figure 9.30: Hiew: last_errnos table

Their size can be determined visually (red lines are drawn here).

While dumping these values, we have found that each 16-bit number is the last error code for each block.

So that’s how Oracle RDBMS quickly finds the error message:

- load a table we will call last_errnos (that contains the last error number for each block);
- find a block that contains the error code we need, assuming all error codes increase across each block and across the file as well;
- load the specific block;
- enumerate the 6-byte structures until the specific error number is found;
- get the position of the first character from the current 6-byte block;
- get the position of the last character from the next 6-byte block;
- load all characters of the message in this range.

This is C program that we wrote which unpacks .MSB-files: beginners.re.

There are also the two files which were used in the example (Oracle RDBMS 11.1.0.6): beginners.re, beginners.re.
9.6.1 Summary
The method is probably too old-school for modern computers. Supposedly, this file format was developed in the mid-80’s by someone who also coded for big iron with memory/disk space economy in mind. Nevertheless, it has been an interesting and yet easy task to understand a proprietary file format without looking into Oracle RDBMS’s code.

9.7 Exercises
Try to reverse engineer of any binary files of your favorite game, including high-score files, resources, etc. There are also binary files with known structure: utmp/wtmp files, try to understand its structure without documentation.

The EXIF header in JPEG file is documented, but you can try to understand its structure without help, just shoot photos at various date/time, places, and try to find date/time and GPS location in EXIF. Try to patch GPS location, upload JPEG file to Facebook and see, how it will put your picture on the map.

Try to patch any information in MP3 file and see how your favorite MP3-player will react.

9.8 Further reading
Pierre Capillon – Black-box cryptanalysis of home-made encryption algorithms: a practical case study.
How to Hack an Expensive Camera and Not Get Killed by Your Wife.
Chapter 10

Dynamic binary instrumentation

DBI tools can be viewed as highly advanced and fast debuggers.

10.1 Using PIN DBI for XOR interception

PIN from Intel is a DBI tool. That means, it takes compiled binary and inserts your instructions in it, where you want.

Let’s try to intercept all XOR instructions. These are heavily used in cryptography, and we can try to run WinRAR archiver in encryption mode with a hope that some XOR instruction is indeed is used while encryption.

Here is the source code of my PIN tool: https://beginners.re/current-tree/DBI/XOR/files//XOR_ins.cpp.

The code is almost self-explanatory: it scans input executable file for all XOR/PXOR instructions and inserts a call to our function before each. log_info() function first checks, if operands are different (since XOR is often used just to clear register, like XOR EAX, EAX), and if they are different, it increments a counter at this EIP/RIP, so the statistics will be gathered.

I have prepared two files for test: test1.bin (30720 bytes) and test2.bin (5547752 bytes), I’ll compress them by RAR with password and see difference in statistics.

You’ll also need to turn off ASLR 1, so the PIN tool will report the same RIPS as in RAR executable.

Now let’s run it:

c:\pin-3.2-81205-msvc-windows\pin.exe -t XOR_ins.dll -- rar a -pLongPassword tmp.rar test1.bin
c:\pin-3.2-81205-msvc-windows\pin.exe -t XOR_ins.dll -- rar a -pLongPassword tmp.rar test2.bin

Now here is statistics for the test1.bin: https://beginners.re/current-tree/DBI/XOR/files//XOR_ins.out.test1. ... and for test2.bin: https://beginners.re/current-tree/DBI/XOR/files//XOR_ins.out.test2. So far, you can ignore all addresses other than ip=0x1400xxxxx, which are in other DLLs.

Now let’s see a difference: https://beginners.re/current-tree/DBI/XOR/files//XOR_ins.diff.

Some XOR instructions are executed more often for test2.bin (which is bigger) than for test1.bin (which is smaller). So these are clearly related to file size!

The first block of differences is:

< ip=0x140017b21 count=0xd84
< ip=0x140017b48 count=0x81f
< ip=0x140017b59 count=0x858
< ip=0x140017b6a count=0xc13
< ip=0x140017b7b count=0xefc
< ip=0x140017b8a count=0xefd
< ip=0x140017b92 count=0xb86
< ip=0x140017b1a count=0xf01
...
> ip=0x140017b21 count=0x9eab5

1https://stackoverflow.com/q/9560993
This is indeed some kind of loop inside of RAR.EXE:

```
.loc_140017B21:
xor r11d, [rbx]
mov r9d, [rbx+4]
add rbx, 8
mov eax, r9d
shr eax, 18h
movzx edx, al
mov eax, r9d
shr eax, 10h
movzx ecx, al
mov eax, r11d
shr eax, 8
mov r8d, [rsi+rdx*4]
oxor r8d, [rsi+rcx*4+400h]
movzx ecx, al
mov eax, r11d
shr eax, 18h
movzx r8d, [rsi+rcx*4+800h]
movzx ecx, al
mov eax, r11d
shr eax, 10h
movzx r8d, [rsi+rcx*4+1000h]
movzx ecx, al
mov eax, r11d
shr eax, 8
movzx r8d, [rsi+rcx*4+1400h]
movzx ecx, al
mov eax, r9b
movzx r8d, [rsi+rcx*4+1800h]
movzx ecx, eax
xor r8d, [rsi+rax*4+0C00h]
movzx ecx, eax
mov r11d, r8d
xor r11d, [rsi+rax*4+1C00h]
sub rdi, 1
jnz loc_140017B21
```

What does it do? No idea yet.

The next:

```
< ip=0x14002c4f1 count=0x4fce
...>
```

0x4fce is 20430, which is close to size of test1.bin (30720 bytes). 0x4463be is 4481982 which is close to size of test2.bin (5547752 bytes). Not equal, but close.

This is a piece of code with that XOR instruction:

```
.loc_14002C4EA:
movzx eax, byte ptr [r8]
shr ecx, 5
xor ecx, eax
and ecx, 7FFh
cmp [r11+rcx*4], esi
jb short loc_14002C507
cmp [r11+rcx*4], r10d
ja short loc_14002C507
inc ebx
```
Loop body can be written as:

\[
\text{state} = \text{input\_byte} \ ^\ominus \ (\text{state}<<5) \ & \ 0x7FFF\}
\]

\text{state} is then used as index in some table. Is this some kind of CRC\textsuperscript{2}? I don’t know, but this could be a checksumming routine. Or maybe optimized CRC routine? Any ideas?

The next block:

\begin{verbatim}
< ip=0x14004104a count=0x367
< ip=0x140041057 count=0x367
-->
> ip=0x14004104a count=0x24193
> ip=0x140041057 count=0x24193

.text:0000000140041039 loc_140041039:
   .text:0000000140041039   mov   rax, r10
   .text:000000014004103C   add   r10, 10h
   .text:0000000140041040   cmp   byte ptr [rcx+1], 0
   .text:0000000140041044   movdqu xmm0, xmmword ptr [rax]
   .text:0000000140041048   jz    short loc_14004104E
   .text:000000014004104A   pxor  xmm0, xmm1
   .text:000000014004104E   pxor  xmm0, xmm1
   .text:0000000140041053   movsx   r8, dword ptr [rcx+4]
   .text:0000000140041057   pxor  xmm1, xmm0
   .text:000000014004105B   cmp   r8d, 1
   .text:000000014004105F   jle   short loc_14004107C
   .text:0000000140041061   lea   rdx, [rcx+28h]
   .text:0000000140041065   lea   r9d, [r8-1]
   .text:0000000140041069   loc_140041069:
   .text:0000000140041069   movdqu xmm0, xmmword ptr [rdx]
   .text:000000014004106D   lea   rdx, [rdx+10h]
   .text:0000000140041071   aesenc  xmm1, xmm0
   .text:0000000140041076   sub   r9, 1
   .text:000000014004107A   jnz   short loc_140041069
   .text:000000014004107C

This piece has both PXOR and AESENC instructions (the last is AES\textsuperscript{3} encryption instruction). So yes, we found encryption function, RAR uses AES.

There is also another big block of almost contiguous XOR instructions:

\begin{verbatim}
< ip=0x140043e10 count=0x23006
< ip=0x140043e56 count=0x22ffd
< ip=0x140043e56 count=0x23002

But, its count is not very different during compressing/encrypting test1.bin/test2.bin. What is on these addresses?

\begin{verbatim}
.text:0000000140043E07 xor   ecx, r9d
.text:0000000140043E0A  mov   r1ld, eax
.text:0000000140043E0D  and   ecx, r10d
.text:0000000140043E10 xor   ecx, r8d
.text:0000000140043E13 rol   eax, 8
.text:0000000140043E16 and   eax, esi
.text:0000000140043E18  ror   r1ld, 8
.text:0000000140043E1C  add   edx, 5A827999h
.text:0000000140043E22  ror   r10d, 2
.text:0000000140043E26  add   r8d, 5A827999h
.text:0000000140043E29  and   r1ld, r12d

\textsuperscript{2}Cyclic redundancy check
\textsuperscript{3}Advanced Encryption Standard
\end{verbatim}
\end{verbatim}
Let's google 5A827999h constant... this looks like SHA-1! But why would RAR use SHA-1 during encryption?

Here is the answer:

In comparison, WinRAR uses its own key derivation scheme that requires (password length * 2 + \( \times 11 \)) * 4096 SHA-1 transformations. That's why it takes longer to brute-force attack \( \times \) encrypted WinRAR archives.


This is key scheduling: input password hashed many times and the hash is then used as AES key. This is why we see the count of XOR instruction is almost unchanged during we switched to bigger test file.

This is it, it took couple of hours for me to write this tool and to get at least 3 points: 1) probably checksumming; 2) AES encryption; 3) SHA-1 calculation. The first function is still unknown for me.

Still, this is impressive, because I didn't dig into RAR code (which is proprietary, of course). I didn't even peek into UnRAR source code (which is available).

The files, including test files and RAR executable I've used (win64, 5.40):

[https://beginners.re/current-tree/DBI/XOR/files/](https://beginners.re/current-tree/DBI/XOR/files/)

### 10.2 Cracking Minesweeper with PIN

In this book, I wrote about cracking Minesweeper for Windows XP: 8.4 on page 794.

The Minesweeper in Windows Vista and 7 is different: probably it was (re)written to C++, and a cell information is now stored not in global array, but rather in malloc'ed heap blocks.

This is a case when we can try PIN DBI tool.

#### 10.2.1 Intercepting all `rand()` calls

First, since Minesweeper places mines randomly, it has to call `rand()` or similar function. Let's intercept all `rand()` calls: [https://beginners.re/current-tree/DBI/minesweeper/minesweeper1.cpp](https://beginners.re/current-tree/DBI/minesweeper/minesweeper1.cpp).

Now we can run it:

```
c:\pin-3.2-81205-msvc-windows\pin.exe -t minesweeper1.dll -- C:\PATH\TO\MineSweeper.exe
```

During startup, PIN searches for all calls to `rand()` function and adds a hook right after each call. The hook is the RandAfter() function we defined: it is logging about return value and also about return address. Here is a log I got during run of standard 9*9 configuration (10 mines): [https://beginners.re/current-tree/DBI/minesweeper/minesweeper1.out.10mines](https://beginners.re/current-tree/DBI/minesweeper/minesweeper1.out.10mines). The `rand()` function was called many times from several places, but was called from 0x10002770d just 10 times. I switched Minesweeper to 16*16 configuration (40 mines) and `rand()` was called from 0x10002770d 40 times. So yes, this is our point. When I load minesweeper.exe (from Windows 7) into IDA and PDB from Microsoft website is fetched, the function which calls `rand()` at 0x10002770d called Board::placeMines().

#### 10.2.2 Replacing `rand()` calls with our function

Let's now try to replace `rand()` function with our version, let it always return zero: [https://beginners.re/current-tree/DBI/minesweeper/minesweeper2.cpp](https://beginners.re/current-tree/DBI/minesweeper/minesweeper2.cpp). During startup, PIN replaces all calls to `rand()` to calls to our function, which writes to log and returns zero. OK, I run it, and clicked on leftmost/topmost cell:
Yes, unlike Minesweeper from Windows XP, mines are places randomly after user’s click on cell, so to guarantee there is no mine at the cell user first clicked. So Minesweeper placed mines on cells other than leftmost/topmost (where I clicked).

Now I clicked on rightmost/topmost cell:

This can be some kind of practical joke? I don’t know.

I clicked on 5th cell (right at the middle) at the 1st row:

This is nice, because Minesweeper can do some correct placement even with such a broken PRNG!
10.2.3 Peeking into placement of mines

How can we get information about where mines are placed? rand()'s result is seems to be useless: it returned zero all the time, but Minesweeper somehow managed to place mines in different cells, though, lined up.

This Minesweeper also written in C++ tradition, so it has no global arrays.

Let us put ourselves in the position of programmer. It has to be loop like:

```c
for (int i; i<mines_total; i++)
{
    // get coordinates using rand()
    // put a cell: in other words, modify a block allocated in heap
};
```

How can we get information about heap block which gets modified at the 2nd step? What we need to do: 1) track all heap allocations by intercepting malloc()/realloc()/free(). 2) track all memory writes (slow). 3) intercept calls to rand().

Now the algorithm: 1) mark all heap blocks gets modified between 1st and 2nd call to rand() from 0x10002770d; 2) whenever heap block gets freed, dump its contents.

Tracking all memory writes is slow, but after 2nd call to rand(), we don’t need to track it (since we’ve got already a list of blocks of interest at this point), so we turn it off.


As it turns out, only 4 heap blocks gets modified between first two rand() calls, this is how they looks like:

```c
free(0x20aa6360)
free(): we have this block in our records, size=0x28
0x20AA6360: 36 00 00 00 4E 00 00 00-20 00 00 29 00 00 00 "6...N......
0x20AA6370: 06 00 00 00 37 00 00 00-35 00 00 19 00 00 00 "....7...5..."
0x20AA6380: 46 00 00 00 0B 00 00 00 -
```

```c
free(0x20af9d10)
free(): we have this block in our records, size=0x18
0x20AF9D10: 0A 00 00 00 0A 00 00 00-0A 00 00 00 00 00 00 00 "................
0x20AF9D20: 60 63 AA 20 00 00 00 00-`c. .... "
```

```c
free(0x20b28b20)
free(): we have this block in our records, size=0x140
0x20B28B20: 02 00 00 00 03 00 00 00-04 00 00 00 05 00 00 00 ".............
0x20B28B30: 07 00 00 00 08 00 00 00-00 00 00 00 0D 00 00 00 ".............
0x20B28B40: 0E 00 00 00 0F 00 00 00-10 00 00 00 11 00 00 00 ".............
0x20B28B50: 12 00 00 00 13 00 00 00-14 00 00 00 15 00 00 00 ".............
0x20B28B60: 16 00 00 00 17 00 00 00-18 00 00 00 1A 00 00 00 ".............
0x20B28B70: 1B 00 00 00 1C 00 00 00-1D 00 00 00 1E 00 00 00 ".............
0x20B28B80: 1F 00 00 00 20 00 00 00-21 00 00 00 22 00 00 00 ".............
0x20B28B90: 23 00 00 00 24 00 00 00-25 00 00 00 26 00 00 00 ".............
0x20B28BA0: 27 00 00 00 28 00 00 00-29 00 00 00 2A 00 00 00 ".............
0x20B28BB0: 2B 00 00 00 2C 00 00 00-2D 00 00 00 2E 00 00 00 ".............
0x20B28BC0: 2F 00 00 00 30 00 00 00-31 00 00 00 32 00 00 00 ".............
0x20B28BDD: 38 00 00 00 39 00 00 00-3A 00 00 00 3B 00 00 00 "B...9.......
0x20B28BEE: 3C 00 00 00 3D 00 00 00-3E 00 00 00 3F 00 00 00 "<...=...>...?
0x20B28BF0: 40 00 00 00 41 00 00 00-42 00 00 00 43 00 00 00 "@...A...B...C...
0x20B28BC0: 44 00 00 00 45 00 00 00-46 00 00 00 47 00 00 00 "D...E...G...H...
0x20B28BC1: 49 00 00 00 4A 00 00 00-4B 00 00 00 4C 00 00 00 "I...J...K...L...
0x20B28BC2: 4D 00 00 00 4E 00 00 00-4F 00 00 00 50 00 00 00 "M...O...P...P...
0x20B28BC3: 50 00 00 00 52 00 00 00-53 00 00 00 54 00 00 00 "P...P...P...P...
0x20B28BC4: 50 00 00 00 50 00 00 00-50 00 00 00 50 00 00 00 "P...P...P...P...
0x20B28BC5: 50 00 00 00 50 00 00 00-50 00 00 00 50 00 00 00 "P...P...P...P..."
```

...
We can easily see that the biggest blocks (with size 0x28 and 0x140) are just arrays of values up to ≈ 0x50. Wait... 0x50 is 80 in decimal representation. And 9*9=81 (standard minesweeper configuration).

After quick investigation, I’ve found that each 32-bit element is indeed cell coordinate. A cell is represented using a single number, it’s a number inside of 2D-array. Row and column of each mine is decoded like that: row=n/\text{WIDTH}; \text{col}=n \% \text{HEIGHT};

So when I tried to decode these two biggest blocks, I’ve got these cell maps:

```
try_to_dump_cells(). unique elements=0xa
........*
........*
........*
...........
*........
*........*
*........
*........
*........
*........
...........

try_to_dump_cells(). unique elements=0x44
*.*.*.*
*.*.*.*
*.*.*.*
*.*.*.*
*.*.*.*
*.*.*.*
*.*.*.*
*.*.*.*
```

It seems that the first block is just a list of mines placed, while the second block is a list of free cells, but, the second is somewhat out of sync with the first one, and it’s negative version of the first one coincides only partially. Nevertheless, the first map is correct - we can peek into it in log file when Minesweeper is still loaded and almost all cells are hidden, and click safely on cells marked as dots here.

So it seems, when user first clicked somewhere, Minesweeper places 10 mines, than destroys the block with a list of it (perhaps, it copies all the data to another block before?), so we can see it during free() call.

Another fact: the method Array<NodeType>::Add(NodeType) modifies blocks we observed, and is called from various places, including Board::placeMines(). But what is cool: I never got into its details, everything has been resolved using just PIN.

The files: https://beginners.re/current-tree/DBI/minesweeper.

### 10.2.4 Exercise

Try to understand how rand()'s result being converted into coordinate(s). As a practical joke, make rand() to output such results, so mines will be placed in shape of some symbol or figure.

### 10.3 Building Pin

Building Pin for Windows may be tricky. This is my working recipe.

- Unpack latest Pin to, say, C:\pin-3.7\
- Install latest Cygwin, to, say, c:\cygwin64
- Install MSVC 2015 or newer.
- Open file C:\pin-3.7\source\tools\Config\makefile.default.rules, replace mkdir -p $@ to /bin/mkdir -p $@
- (If needed) in C:\pin-3.7\source\tools\SimpleExamples\makefile.rules, add your pintool to the TEST_TOOL_ROOTS list.
10.4 Why “instrumentation”?  

Perhaps, this is a term of code profiling. There are at least two methods: 1) “sampling”: you break into running code as many times as possible (hundreds per second), and see, where it is executed at the moment; 2) “instrumentation”: compiled code is interleaved with other code, which can increment counters, etc.

Perhaps, DBI tools inherited the term?
Chapter 11

Other things

11.1 Executable files patching

11.1.1 x86 code

Frequent patching tasks are:

- One of the most frequent jobs is to disable some instruction. It is often done by filling it using byte 0x90 (NOP).
- Conditional jumps, which have an opcode like 74 xx (JZ), can be filled with two NOPs.
  It is also possible to disable a conditional jump by writing 0 at the second byte (jump offset).
- Another frequent job is to make a conditional jump to always trigger: this can be done by writing 0xEB instead of the opcode, which stands for JMP.
- A function's execution can be disabled by writing RETN (0xC3) at its beginning. This is true for all functions excluding stdcall (6.1.2 on page 725). While patching stdcall functions, one has to determine the number of arguments (for example, by finding RETN in this function), and use RETN with a 16-bit argument (0xC2).
- Sometimes, a disabled function has to return 0 or 1. This can be done by MOV EAX, 0 or MOV EAX, 1, but it's slightly verbose. A better way is XOR EAX, EAX (2 bytes 0x31 0xC0) or XOR EAX, EAX / INC EAX (3 bytes 0x31 0xC0 0x40).

A software may be protected against modifications. This protection is often done by reading the executable code and calculating a checksum. Therefore, the code must be read before protection is triggered.

This can be determined by setting a breakpoint on reading memory.
tracer has the BPM option for this.

PE executable file relocs (6.5.2 on page 751) must not to be touched while patching, because the Windows loader may overwrite your new code. (They are grayed in Hiew, for example: fig.1.22).

As a last resort, it is possible to write jumps that circumvent the relocs, or you will have to edit the relocs table.

11.2 Function arguments number statistics

I've always been interesting in what is average number of function arguments.

I've analyzed many Windows 7 32-bit DLLs (crypt32.dll, mfc71.dll, msvcr100.dll, shell32.dll, user32.dll, d3d11.dll, mshtml.dll, msxml6.dll, sqlncli11.dll, wininet.dll, mfc120.dll, msvbvm60.dll, ole32.dll, themeui.dll, wmp.dll) (because they use stdcall convention, and so it is easy to grep disassembly output just by RETN X).

- no arguments: ≈ 29%
- 1 argument: ≈ 23%
• 2 arguments: \( \approx 20\% \)
• 3 arguments: \( \approx 11\% \)
• 4 arguments: \( \approx 7\% \)
• 5 arguments: \( \approx 3\% \)
• 6 arguments: \( \approx 2\% \)
• 7 arguments: \( \approx 1\% \)

![Figure 11.1: Function arguments number statistics](image)

This is heavily dependent on programming style and may be very different for other software products.

11.3 Compiler intrinsic

A function specific to a compiler which is not an usual library function. The compiler generates a specific machine code instead of a call to it. It is often a pseudofunction for specific CPU instruction.

For example, there are no cyclic shift operations in C/C++ languages, but they are present in most CPUs. For programmer’s convenience, at least MSVC has pseudofunctions \(_\text{rotl}()\) and \(_\text{rotr}()\)\(^1\) which are translated by the compiler directly to the ROL/ROR x86 instructions.

Another example are functions to generate SSE-instructions right in the code.

Full list of MSVC intrinsics: [MSDN](https://developer.microsoft.com/en-us/)

11.4 Compiler’s anomalies

11.4.1 Oracle RDBMS 11.2 and Intel C++ 10.1

Intel C++ 10.1, which was used for Oracle RDBMS 11.2 Linux86 compilation, may emit two JZ in row, and there are no references to the second JZ. The second JZ is thus meaningless.

Listing 11.1: kdli.o from libserver11.a

```assembly
.text:08114CF1  loc_8114CF1:  ; CODE XREF:  _PG0SF539_kdlimemSer+89A
.text:08114CF1  ; _PG0SF539_kdlimemSer+3994
.text:08114CF1  8B 45 08  mov    eax, [ebp+arg_0]
```

\(^1\)MSDN
It is supposedly a code generator bug that was not found by tests, because resulting code works correctly anyway.

Another example from Oracle RDBMS 11.1.0.6.0 for win32.

11.4.2 MSVC 6.0

Just found in some old code:

The first FXCH instruction swaps ST(0) and ST(1), the second do the same, so both do nothing. This is a program uses MFC42.dll, so it could be MSVC 6.0, 5.0 or maybe even MSVC 4.2 from 1990s.

This pair do nothing, so it probably wasn't caught by MSVC compiler tests. Or maybe I wrong?

11.4.3 ftol2() in MSVC 2012

Just found this in ftol2() standard C/C++ library function (float-to-long conversion routine) in Microsoft Visual Studio 2012.

The first FXCH instruction swaps ST(0) and ST(1), the second do the same, so both do nothing. This is a program uses MFC42.dll, so it could be MSVC 6.0, 5.0 or maybe even MSVC 4.2 from 1990s.

This pair do nothing, so it probably wasn't caught by MSVC compiler tests. Or maybe I wrong?
Note two identical FSTP-s (float store with pop) at the end. First I thought it was compiler anomaly (I’m collecting such cases just as someone do with butterflies), but it seems, it’s handwritten assembler piece, in msvcrtd.lib there is an object file with this function in it, and we can find this string in it: f:\dd\vctools\crt_bld\SELF_X86\crt\prebuild\tran\i386\ftol2.asm — that was probably a path to the file on developer’s computer where msvcrtd.lib was built.

So, bug, text editor-induced typo, or feature?

11.4.4 Summary

Other compiler anomalies here in this book: 1.28.2 on page 313, 3.10.3 on page 493, 3.18.7 on page 531, 1.26.7 on page 299, 1.18.4 on page 146, 1.28.5 on page 330.

Such cases are demonstrated here in this book, to show that such compilers errors are possible and sometimes one should not to rack one’s brain while thinking why did the compiler generate such strange code.
11.5 Itanium

Although almost failed, Intel Itanium (IA64) is a very interesting architecture.

While OOE CPUs decides how to rearrange their instructions and execute them in parallel, EPIC\textsuperscript{2} was an attempt to shift these decisions to the compiler: to let it group the instructions at the compile stage.

This resulted in notoriously complex compilers.

Here is one sample of IA64 code: simple cryptographic algorithm from the Linux kernel:

```c
#define TEA_ROUNDS 32
#define TEA_DELTA 0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const u8 *src)
{
    u32 y, z, n, sum = 0;
    u32 k0, k1, k2, k3;
    struct tea_ctx *ctx = crypto_tfm_ctx(tfm);
    const __le32 *in = (const __le32 *)src;
    __le32 *out = (__le32 *)dst;

    y = le32_to_cpu(in[0]);
    z = le32_to_cpu(in[1]);

    k0 = ctx->KEY[0];
    k1 = ctx->KEY[1];
    k2 = ctx->KEY[2];
    k3 = ctx->KEY[3];

    n = TEA_ROUNDS;

    while (n-- > 0) {
        sum += TEA_DELTA;
        y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + k1);
        z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + k3);
    }

    out[0] = cpu_to_le32(y);
    out[1] = cpu_to_le32(z);
}
```

Here is how it was compiled:

```assembly
Listing 11.4: Linux Kernel 3.2.0.4 for Itanium 2 (McKinley)

0090| tea_encrypt:
0090|08 80 80 41 00 21  adds r16 = 96, r32    // ptr to ctx->KEY[2]
0096|80 C0 82 00 42 00  adds r8 = 88, r32     // ptr to ctx->KEY[0]
009C|00 00 04 00  nop.i 0
00A0|09 18 70 41 00 21  adds r3 = 92, r32     // ptr to ctx->KEY[1]
00A6|F0 20 88 00 84 00  ld4 r15 = [r34], 4      // load z
00AC|44 06 01 84 00  adds r32 = 100, r32;;     // ptr to ctx->KEY[3]
00B0|08 98 00 20 10 10  ld4 r19 = [r16]        // r19=k2
00B6|00 01 00 00 42 40  mov r16 = r0          // r0 always contain zero
00BC|00 08 CA 00  mov.i r2 = ar.lc            // save lc register
00C0|05 70 00 44 10 10  ld4 r14 = [r34]        // load y
00C4|92 F3 CE 6B 00  movl r17 = 0xFFFFFFFF9E3779B9;; // TEA_DELTA
00C8|08 00 00 00 01 00  nop.m 0
00D0|50 01 20 00 20 00  ld4 r21 = [r8]         // r21=k0
00DC|F0 09 2A 00  mov.i ar.lc = 31            // TEA_ROUNDS is 32
00E0|0A A0 00 06 10 10  ld4 r20 = [r3];;       // r20=k1
00E4|20 01 80 00 20 00  ld4 r18 = [r32]        // r18=k3
00EC|00 00 04 00  nop.i 0
00F0| loc_F0:
00F0|09 80 40 22 00 20  add r16 = r16, r17    // r16=sum, r17=TEA_DELTA
```

\textsuperscript{2}Explicitly Parallel Instruction Computing
First of all, all IA64 instructions are grouped into 3-instruction bundles. Each bundle has a size of 16 bytes (128 bits) and consists of template code (5 bits) + 3 instructions (41 bits for each).

IDA shows the bundles as 6+6+4 bytes—you can easily spot the pattern. All 3 instructions from each bundle usually executes simultaneously, unless one of instructions has a “stop bit”.

Supposedly, Intel and HP engineers gathered statistics on most frequent instruction patterns and decided to bring bundle types (AKA “templates”): a bundle code defines the instruction types in the bundle. There are 12 of them.

For example, the zeroth bundle type is MII, which implies the first instruction is Memory (load or store), the second and third ones are I (integer instructions). Another example is the bundle of type 0x1d: MFB: the first instruction is Memory (load or store), the second one is Float (FPU instruction), and the third is Branch (branch instruction).

If the compiler cannot pick a suitable instruction for the relevant bundle slot, it may insert a NOP: you can see here the nop.i instructions (NOP at the place where the integer instruction might be) or nop.m (a memory instruction might be at this slot).

NOPs are inserted automatically when one uses assembly language manually. And that is not all. Bundles are also grouped. Each bundle may have a “stop bit”, so all the consecutive bundles with a terminating bundle which has the “stop bit” can be executed simultaneously.

In practice, Itanium 2 can execute 2 bundles at once, resulting in the execution of 6 instructions at once. So all instructions inside a bundle and a bundle group cannot interfere with each other (i.e., must not have data hazards).

If they do, the results are to be undefined. Each stop bit is marked in assembly language as two semicolons (;;) after the instruction.

So, the instructions at [90-ac] may be executed simultaneously: they do not interfere. The next group is [b0-cc].

We also see a stop bit at 10c. The next instruction at 110 has a stop bit too. This implies that these instructions must be executed isolated from all others (as in CISC).
Indeed, the next instruction at 110 uses the result from the previous one (the value in register r26), so they cannot be executed at the same time.

Apparently, the compiler was not able to find a better way to parallelize the instructions, in other words, to load CPU as much as possible, hence too much stop bits and NOPs.

Manual assembly programming is a tedious job as well: the programmer has to group the instructions manually.

The programmer is still able to add stop bits to each instructions, but this will degrade the performance that Itanium was made for.

An interesting examples of manual IA64 assembly code can be found in the Linux kernel’s sources: http://go.yurichev.com/17322.


Another very interesting Itanium feature is the speculative execution and the NaT (“not a thing”) bit, somewhat resembling NaN numbers: MSDN.

### 11.6 8086 memory model

When dealing with 16-bit programs for MS-DOS or Win16 (8.8.3 on page 835 or 3.34.5 on page 648), we can see that the pointers consist of two 16-bit values. What do they mean? Oh yes, that is another weird MS-DOS and 8086 artifact.

8086/8088 was a 16-bit CPU, but was able to address 20-bit address in RAM (thus being able to access 1MB of external memory).

The external memory address space was divided between RAM (640KB max), ROM, windows for video memory, EMS cards, etc.

Let’s also recall that 8086/8088 was in fact an inheritor of the 8-bit 8080 CPU.

The 8080 has a 16-bit memory space, i.e., it was able to address only 64KB.

And probably because of reason of old software porting5, 8086 can support many 64KB windows simultaneously, placed within the 1MB address space.

This is some kind of a toy-level virtualization.

All 8086 registers are 16-bit, so to address more, special segment registers (CS, DS, ES, SS) were introduced.

Each 20-bit pointer is calculated using the values from a segment register and an address register pair (e.g. DS:BX) as follows:

\[
\text{real\_address} = (\text{segment\_register} \ll 4) + \text{address\_register}
\]

For example, the graphics (EGA6, VGA7) video RAM window on old IBM PC-compatibles has a size of 64KB. To access it, a value of 0xA000 has to be stored in one of the segment registers, e.g. into DS.

Then DS:0 will address the first byte of video RAM and DS:0xFFFF — the last byte of RAM.

The real address on the 20-bit address bus, however, will range from 0xA0000 to 0xAFFFF.

The program may contain hard-coded addresses like 0x1234, but the OS may need to load the program at arbitrary addresses, so it recalculates the segment register values in a way that the program does not have to care where it’s placed in the RAM.

So, any pointer in the old MS-DOS environment in fact consisted of the segment address and the address inside segment, i.e., two 16-bit values. 20-bit was enough for that, though, but we needed to recalculate the addresses very often: passing more information on the stack seemed a better space/convenience balance.

---

3Also available as http://yurichev.com/mirrors/RE/itanium.pdf
4Also available as http://phrack.org/issues/57/5.html
5The author is not 100% sure here
6Enhanced Graphics Adapter
7Video Graphics Array
By the way, because of all this it was not possible to allocate a memory block larger than 64KB.

The segment registers were reused at 80286 as selectors, serving a different function.

When the 80386 CPU and computers with bigger **RAM** were introduced, MS-DOS was still popular, so the DOS extenders emerged: these were in fact a step toward a “serious” **OS**, switching the CPU in protected mode and providing much better memory **APIs** for the programs which still needed to run under MS-DOS.

Widely popular examples include DOS/4GW (the DOOM video game was compiled for it), Phar Lap, PMODE.

By the way, the same way of addressing memory was used in the 16-bit line of Windows 3.x, before Win32.

### 11.7 Basic blocks reordering

#### 11.7.1 Profile-guided optimization

This optimization method can move some **basic blocks** to another section of the executable binary file.

Obviously, there are parts of a function which are executed more frequently (e.g., loop bodies) and less often (e.g., error reporting code, exception handlers).

The compiler adds instrumentation code into the executable, then the developer runs it with a lot of tests to collect statistics.

Then the compiler, with the help of the statistics gathered, prepares final the executable file with all infrequently executed code moved into another section.

As a result, all frequently executed function code is compacted, and that is very important for execution speed and cache usage.

An example from Oracle RDBMS code, which was compiled with Intel C++:

```
Listing 11.5: orageneric11.dll (win32)

    public _skgfsync
    _skgfsync proc near

    ; address 0x6030D86A
    db 66h
    nop
    push ebp
    mov ebp, esp
    mov edx, [ebp+0Ch]
    test edx, edx
    jz short loc_6030D884
    mov eax, [edx+30h]
    test eax, 400h
    jnz __VInfreq__skgfsync ; write to log
    continue:
    mov eax, [ebp+8]
    mov edx, [ebp+10h]
    mov dword ptr [eax], 0
    lea eax, [edx+0Fh]
    and eax, 0FFFFFFFCh
    mov ecx, [eax]
    cmp ecx, 45726963h
    jnz error ; exit with error
    mov esp, ebp
    pop ebp
    ret

    _skgfsync endp

    ...

    ; address 0x60B953F0

    __VInfreq__skgfsync:
    mov eax, [edx]
    test eax, eax
    jz continue
    mov ecx, [ebp+10h]
```
The distance of addresses between these two code fragments is almost 9 MB.
All infrequently executed code was placed at the end of the code section of the DLL file, among all function parts.
This part of the function was marked by the Intel C++ compiler with the VInfreq prefix.
Here we see that a part of the function that writes to a log file (presumably in case of error or warning or something like that) which was probably not executed very often when Oracle’s developers gathered statistics (if it was executed at all).
The writing to log basic block eventually returns the control flow to the “hot” part of the function.
Another “infrequent” part is the basic block returning error code 27050.
In Linux ELF files, all infrequently executed code is moved by Intel C++ into the separate text.unlikely section, leaving all “hot” code in the text.hot section.
From a reverse engineer’s perspective, this information may help to split the function into its core and error handling parts.

11.8 My experience with Hex-Rays 2.2.0

11.8.1 Bugs
There are couple of bugs.
First of all, Hex-Rays is getting lost when FPU instructions are interleaved (by compiler codegenerator) with others.
For example, this:

```
f        proc    near
    lea    eax, [esp+4]
    fild   dword ptr [eax]
    lea    eax, [esp+8]
    fild   dword ptr [eax]
    fabs
    fcompp
    fnstsw  ax
    test   ah, 1
    jz     l01
    mov    eax, 1
    ret
l01:
    mov    eax, 2
    ret
```

...will be correctly decompiled to:

```c
signed int __cdecl f(signed int a1, signed int a2)
{
    signed int result; // eax@2
    if ( fabs((double)a2) >= (double)a1 )
        result = 2;
    else
        result = 1;
    return result;
}
```

But let's comment one of the instructions at the end:

```asm
... l01:
    ;mov eax, 2
    retn ...
```

...we getting an obvious bug:

```c
void __cdecl f(char a1, char a2)
{
    fabs((double)a2);
}
```

This is another bug:

```asm
extrn f1:dword
extrn f2:dword
f
    proc near
    fld dword ptr [esp+4]
    fadd dword ptr [esp+8]
    fst dword ptr [esp+12]
    fcomp ds:const_100
    fld dword ptr [esp+16]
    fnstsw ax
    test ah, 1
    jnz short l01
    call f1
    retn
l01:
    call f2
    retn
f
    endp ...
```

Result:

```c
int __cdecl f(float a1, float a2, float a3, float a4)
{
    double v5; // st7@1
    char v6; // c0@1
    int result; // eax@2
    v5 = a4;
    if ( v6 )
```
v6 variable has char type and if you’ll try to compile this code, compiler will warn you about variable usage before assignment.

Another bug: FPATAN instruction is correctly decompiled into atan2(), but arguments are swapped.

### 11.8.2 Odd peculiarities

Hex-Rays too often promotes 32-bit int to 64-bit one. Here is example:

```assembly
f proc near
    mov    eax, [esp+4]
    cdq
    xor    eax, edx
    sub    eax, edx
; EAX=abs(a1)
    sub    eax, [esp+8]
; EAX=EAX-a2
; EAX at this point somehow gets promoted to 64-bit (RAX)
    cdq
    xor    eax, edx
    sub    eax, edx
; EAX=abs(abs(a1)-a2)
retn
f endp
```

Result:

```c
int __cdecl f(int a1, int a2)
{
    __int64 v2; // rax@1
    v2 = abs(a1) - a2;
    return (HIDWORD(v2) ^ v2) - HIDWORD(v2);
}
```

Perhaps, this is result of CDQ instruction? I’m not sure. Anyway, whenever you see __int64 type in 32-bit code, pay attention.

This is also weird:

```assembly
f proc near
    mov    esi, [esp+4]
    lea    ebx, [esi+10h]
    cmp    esi, ebx
    jge    short l00
    cmp    esi, 1000
    jg     short l00
    mov    eax, 2
    retn
l00:
    mov    eax, 1
    retn
```

The code is correct, but needs manual intervention.

Sometimes, Hex-Rays doesn’t fold (or reduce) division by multiplication code:

```
f proc near
    mov   eax, [esp+4]
    mov   edx, 2AAAAAABh
    imul  edx
    mov   eax, edx
    retn
f endp
```

Result:

```
int __cdecl f(int a1)
{
    return (unsigned __int64)(715827883i64 * a1) >> 32;
}
```

This can be folded (rewritten) manually.

Many of these peculiarities can be solved by manual reordering of instructions, recompiling assembly code, and then feeding it to Hex-Rays again.

### 11.8.3 Silence

extrn some_func:dword

```
f proc near
    mov   ecx, [esp+4]
    mov   eax, [esp+8]
    push  eax
    call  some_func
    add   esp, 4
    ; use ECX
    mov   eax, ecx
    retn
f endp
```

Result:

```
int __cdecl f(int a1, int a2)
{
    int v2; // ecx@1
    some_func(a2);
    return v2;
}
```
v2 variable (from ECX) is lost ... Yes, this code is incorrect (ECX value doesn’t saved during call to another function), but it would be good for Hex-Rays to give a warning.

Another one:

```asm
extrn some_func:dword
f proc near
    call some_func
    jnz l01
    mov eax, 1
    ret
l01:
    mov eax, 2
    ret
f endp
```

Result:

```c
signed int f()
{
    char v0; // zf@1
    signed int result; // eax@2

    some_func();
    if ( v0 )
        result = 1;
    else
        result = 2;
    return result;
}
```

Again, warning would be great.

Anyway, whenever you see variable of `char` type, or variable which is used without initialization, this is clear sign that something went wrong and needs manual intervention.

### 11.8.4 Comma

Comma in C/C++ has a bad fame, because it can lead to a confusing code.

Quick quiz, what does this C/C++ function return?

```c
int f()
{
    return 1, 2;
}
```

It’s 2: when compiler encounters comma-expression, it generates code which executes all sub-expressions, and returns value of the last sub-expression.

I’ve seen something like that in production code:

```c
if (cond)
    return global_var=123, 456; // 456 is returned
else
    return global_var=789, 321; // 321 is returned
```

Apparently, programmer wanted to make code slightly shorter without additional curly brackets. In other words, comma allows to pack couple of expressions into one, without forming statement/code block inside of curly brackets.

Comma in C/C++ is close to `begin` in Scheme/Racket: [https://docs.racket-lang.org/guide/begin.html](https://docs.racket-lang.org/guide/begin.html).

Perhaps, the only widely accepted usage of comma is in `for()` statements:
char *s="hello, world";
for(int i=0; *s; s++, i++);
// i = string length

Both s++ and i++ are executed at each loop iteration.
Read more: https://stackoverflow.com/q/52550.
I’m writing all this because Hex-Rays produces (at least in my case) code which is rich with both commas and short-circuit expressions. For example, this is real output from Hex-Rays:

```c
if ( ( a >= b || (c = a, (d[a] - e) >> 2 > f) )
    { ... }
```

This is correct, it compiles and works, and let god help you to understand it. Here it is rewritten:

```c
if (cond1 || (comma_expr, cond2))
    { ... }
```

Short-circuit is effective here: first cond1 is checked, if it’s true, if() body is executed, the rest of if() expression is ignored completely. If cond1 is false, comma_expr is executed (in the previous example, a gets copied to c), then cond2 is checked. If cond2 is true, if() body gets executed, or not. In other words, if() body gets executed if cond1 is true or cond2 is true, but if the latter is true, comma_expr is also executed.

Now you can see why comma is so notorious.

**A word about short-circuit.** A common beginner’s misconception is that sub-conditions are checked in some unspecified order, which is not true. In a | b | c expression, a, b and c gets evaluated in unspecified order, so that is why || has also been added to C/C++, to apply short-circuit explicitly.

### 11.8.5 Data types

Data types is a problem for decompilers.

Hex-Rays can be blind to arrays in local stack, if they weren’t set correctly before decompilation. Same story about global arrays.

Another problem is too big functions, where a single slot in local stack can be used by several variables across function’s execution. It’s not a rare case when a slot is used for int-variable, then for pointer, then for float-variable. Hex-Rays correctly decompiles it: it creates a variable with some type, then cast it to another type in various parts of functions. This problem has been solved by me by manual splitting big function into several smaller. Just make local variables as global ones, etc, etc. And don’t forget about tests.

### 11.8.6 Long and messed expressions

Sometimes, during rewriting, you can end up with long and hard to understand expressions in if() constructs, like:

```c
if (((! (v38 && v30 <= 5 && v27 != -1)) && (! (v38 && v30 <= 5) && v27 != -1)) || (v24 >= 5 || v26 && v25)
    { ... }
```

Wolfram Mathematica can minimize some of them, using BooleanMinimize[] function:

```wolfram
In[1]:= BooleanMinimize[(! (v38 && v30 <= 5 && v27 != -1)) && v38 && v30 <= 5 && v25 == 0]
Out[1]:= v38 && v25 == 0 && v27 == -1 && v30 <= 5
```

There is even better way, to find common subexpressions:
Mathematica has added two new variables: Compile\$1 and Compile\$2, values of which are to be used several times in expression. So we can add two additional variables.

### 11.8.7 De Morgan’s laws and decompilation

Sometimes a compiler’s optimizer can use De Morgan’s laws to make code shorter/faster.

For example, this:

```c
void f(int a, int b, int c, int d)
{
    if (a>0 && b>0)
        printf ("both a and b are positive\n");
    else if (c>0 && d>0)
        printf ("both c and d are positive\n");
    else
        printf ("something else\n");
}

... looks pretty innocent, when compiled by optimizing GCC 5.4.0 x64:

```
else if ( c <= 0 || d <= 0 )
{
    result = puts("something else");
}
else
{
    result = puts("both c and d are positive");
    return result;
}

The c <= 0 || d <= 0 expression is inversion of c>0 && d>0 since A∪B = A∩B and A∩B = A∪B, in other words, !(cond1 || cond2) == !cond1 && !cond2 and !(cond1 && cond2) == !cond1 || !cond2. These rules are worth to be kept in mind, since this compiler optimization is used heavily almost everywhere.

Sometimes it’s good idea to invert a condition, in order to understand a code better. This is a piece of a real code decompiled by Hex-Rays:

```c
for (int i=0; i<12; i++)
{
    if (v1[i-12] != 0.0 || v1[i] != 0.0)
    {
        v108=min(v108, (float)v0[i*24 -2]);
        v113=max(v113, (float)v0[i*24]);
    }
}
```

... it can be rewritten like:

```c
for (int i=0; i<12; i++)
{
    if (v1[i-12] == 0.0 && v1[i] == 0.0)
        continue;
    v108=min(v108, (float)v0[i*24 -2]);
    v113=max(v113, (float)v0[i*24]);
}
```

Which is better? I don’t know yet, but for better understanding, it’s great to take a look on both.

### 11.8.8 My plan

- Split big functions (and don’t forget about tests). Sometimes it’s very helpful to form new functions out of big loop bodies.
- Check/set data type of variables, arrays, etc.
- If you see odd result, dangling variable (which used before initialization), try to swap instructions manually, recompile it and feed to Hex-Rays again.

### 11.8.9 Summary

Nevertheless, quality of Hex-Rays 2.2.0 is very, very good. It makes life way easier.

### 11.9 Cyclomatic complexity

The term is used to measure complexity of a function. Complex functions are usually evil, because they are hard to maintain, hard to test, etc.

There are several heuristics to measure it.

For example, we can find in Linux kernel coding style\(^8\):

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you’re screwed anyway, and should fix your program.

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it’s OK to have a longer function.

However, if you have a complex function, and you suspect that a less-than-gifted first-year high-school student might not even understand what the function is all about, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can ask the compiler to in-line them if you think it’s performance-critical, and it will probably do a better job of it than you would have done).

Another measure of the function is the number of local variables. They shouldn’t exceed 5-10, or you’re doing something wrong. Re-think the function, and split it into smaller pieces.

A human brain can generally easily keep track of about 7 different things, anything more and it gets confused. You know you’re brilliant, but maybe you’d like to understand what you did 2 weeks from now.

In JPL Institutional Coding Standard for the C Programming Language:\n
Functions should be no longer than 60 lines of text and define no more than 6 parameters. A function should not be longer than what can be printed on a single sheet of paper in a standard reference format with one line per statement and one line per declaration. Typically, this means no more than about 60 lines of code per function. Long lists of function parameters similarly compromise code clarity and should be avoided.

Each function should be a logical unit in the code that is understandable and verifiable as a unit. It is much harder to understand a logical unit that spans multiple screens on a computer display or multiple pages when printed. Excessively long functions are often a sign of poorly structured code.

Now let’s back to cyclomatic complexity.

Without diving deep into graph theory: there are basic blocks and links between them. For example, this is how IDA shows BB’s and links (as arrows). Just click space and you’ll see this: 1.18 on page 88. Each BB is also called vertex or node in graph theory. Each link - edge.

There are at least two popular ways to calculate cyclomatic complexity: 1) edges - nodes + 2 2) edges - nodes + number of exits (RET instructions)

As of IDA example below, there are 4 BBs, so that is 4 nodes. But there are also 4 links and 1 return instruction. By 1st rule, this is 2, by the second: 1.

The bigger the number, the more complex your function and things go from bad to worse. As you can see, additional exit (return instructions) make things even worse, as well as additional links between nodes (including additional goto’s).

I wrote the simple IDAPython script (https://beginners.re/current-tree/other/cyclomatic/cyclomatic.py) to measure it. Here is result for Linux kernel 4.11 (most complex functions in it):

```
1829c0 do_check edges=937 nodes=574 rets=1 E-N+2=365 E-N+rets=364
2effe8 ext4_fill_super edges=862 nodes=568 rets=1 E-N+2=296 E-N+rets=295
5d92e0 wm5110_readable_register edges=661 nodes=369 rets=2 E-N+2=294 E-N+rets=294
277650 do_blockdev_direct_IO edges=771 nodes=507 rets=1 E-N+2=266 E-N+rets=265
10f7c0 load_module edges=711 nodes=465 rets=1 E-N+2=248 E-N+rets=247
787730 dev_ethtool edges=559 nodes=315 rets=1 E-N+2=246 E-N+rets=245
84e440 do_ipv6_setsockopt edges=468 nodes=237 rets=1 E-N+2=233 E-N+rets=232
72c3c0 mmc_init_card edges=593 nodes=365 rets=1 E-N+2=230 E-N+rets=229
...\n```

9https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf
10Basic Block
This is source code of some of them: do_check(), ext4_fill_super(), do_blockdev_direct_IO(), do_jit().

Most complex functions in Windows 7 ntoskrnl.exe file:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Edges</th>
<th>Nodes</th>
<th>Rets</th>
<th>E-N+2</th>
<th>E-N+rets</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub_140569400</td>
<td>3070</td>
<td>1889</td>
<td>1</td>
<td>1183</td>
<td>1182</td>
</tr>
<tr>
<td>MmAccessFault</td>
<td>2256</td>
<td>1424</td>
<td>1</td>
<td>834</td>
<td>833</td>
</tr>
<tr>
<td>FsRtlMdlReadCompleteDevEx</td>
<td>1241</td>
<td>752</td>
<td>1</td>
<td>491</td>
<td>490</td>
</tr>
<tr>
<td>MmProbeAndLockPages</td>
<td>623</td>
<td>362</td>
<td>1</td>
<td>361</td>
<td>361</td>
</tr>
<tr>
<td>ExpQuerySystemInformation</td>
<td>995</td>
<td>671</td>
<td>1</td>
<td>326</td>
<td>325</td>
</tr>
<tr>
<td>MmProbeAndLockSelectedPages</td>
<td>551</td>
<td>326</td>
<td>1</td>
<td>325</td>
<td>325</td>
</tr>
<tr>
<td>NtSetInformationProcess</td>
<td>586</td>
<td>296</td>
<td>1</td>
<td>295</td>
<td>295</td>
</tr>
</tbody>
</table>

From a bug hunter’s standpoint, complex functions are prone to have bugs, so an attention should be paid to them.


Chapter 12

Books/blogs worth reading

12.1 Books and other materials

12.1.1 Reverse Engineering


Also, Kris Kaspersky’s books.

12.1.2 Windows

- Mark Russinovich, *Microsoft Windows Internals*
- Peter Ferrie – The “Ultimate” Anti-Debugging Reference

Blogs:

- Microsoft: Raymond Chen
- nynaeve.net

12.1.3 C/C++

- C++11 standard
- Marshall Cline, C++ FAQ
- Dennis Yurichev, C/C++ programming language notes

---

2[^2]: Also available as [http://go.yurichev.com/17274](http://go.yurichev.com/17274)
3[^3]: Also available as [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf)
4[^4]: Also available as [http://agner.org/optimize/optimizing_cpp.pdf](http://agner.org/optimize/optimizing_cpp.pdf)
5[^5]: Also available as [http://go.yurichev.com/17291](http://go.yurichev.com/17291)
6[^6]: Also available as [http://yurichev.com/C-book.html](http://yurichev.com/C-book.html)
12.1.4 x86 / x86-64

- Intel manuals
- AMD manuals

Somewhat outdated, but still interesting to read:

Michael Abrash, *Graphics Programming Black Book*, 1997 (he is known for his work on low-level optimization for such projects as Windows NT 3.1 and id Quake).

12.1.5 ARM

- ARM manuals
- [ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile, (2013)]

12.1.6 Assembly language

Richard Blum — Professional Assembly Language.

12.1.7 Java


12.1.8 UNIX


12.1.9 Programming in general

- Henry S. Warren, *Hacker’s Delight*, (2002). Some people say tricks and hacks from the book are not relevant today because they were good only for RISC CPUs, where branching instructions are expensive. Nevertheless, these can help immensely to understand boolean algebra and what all the mathematics near it.

---

7 Also available as https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf
9 Also available as http://developer.amd.com/resources/developer-guides-manuals/
10 Also available as http://agner.org/optimize/microarchitecture.pdf
11 Also available as http://www.agner.org/optimize/calling_conventions.pdf
12 Also available as https://github.com/jagregory/abrash-black-book
15 Also available as http://go.yurichev.com/17273
16 Also available as https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf; http://docs.oracle.com/javase/specs/jvms/se7/html/
12.1.10 Cryptography

- (Free) lvh, *Crypto 101*\(^{17}\)
- (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*\(^{18}\).

\(^{17}\)Also available as [https://www.crypto101.io/](https://www.crypto101.io/)

\(^{18}\)Also available as [https://crypto.stanford.edu/~dabo/cryptobook/](https://crypto.stanford.edu/~dabo/cryptobook/)
Chapter 13

Communities

There are two excellent RE\textsuperscript{1}-related subreddits on reddit.com: reddit.com/r/ReverseEngineering/ and reddit.com/r/remath (on the topics for the intersection of RE and mathematics).

There is also a RE part of the Stack Exchange website: reverseengineering.stackexchange.com.

On IRC there are ##re and ##asm channels on FreeNode\textsuperscript{2}.

\textsuperscript{1}Reverse Engineering

\textsuperscript{2}freenode.net
Afterword
13.1 Questions?

Do not hesitate to mail any questions to the author: <first_name @ last_name . com> or <first_name . last_name @ gmail . com>. Do you have any suggestion on new content for the book? Please do not hesitate to send any corrections (including grammar (you see how horrible my English is?)), etc.

The author is working on the book a lot, so the page and listing numbers, etc., are changing very rapidly. Please do not refer to page and listing numbers in your emails to me. There is a much simpler method: make a screenshot of the page, in a graphics editor underline the place where you see the error, and send it to the author. He’ll fix it much faster. And if you familiar with git and \LaTeX you can fix the error right in the source code:

GitHub.

Do not worry to bother me while writing me about any petty mistakes you found, even if you are not very confident. I’m writing for beginners, after all, so beginners’ opinions and comments are crucial for my job.
Appendix
.1 x86

.1.1 Terminology

Common for 16-bit (8086/80286), 32-bit (80386, etc.), 64-bit.

byte 8-bit. The DB assembly directive is used for defining variables and arrays of bytes. Bytes are passed in the 8-bit part of registers: AL/BL/CL/CL/AL/BL/CH/DH/SIL/DIL/R*L.

word 16-bit. DW assembly directive —”. Words are passed in the 16-bit part of the registers: AX/BX/CX/DX/SI/DI/R*W.

double word (“dword”) 32-bit. DD assembly directive —”. Double words are passed in registers (x86) or in the 32-bit part of registers (x64). In 16-bit code, double words are passed in 16-bit register pairs.

quad word (“qword”) 64-bit. DQ assembly directive —”. In 32-bit environment, quad words are passed in 32-bit register pairs.

tbyte (10 bytes) 80-bit or 10 bytes (used for IEEE 754 FPU registers).

paragraph (16 bytes)—term was popular in MS-DOS environment.

Data types of the same width (BYTE, WORD, DWORD) are also the same in Windows API.

.1.2 General purpose registers

It is possible to access many registers by byte or 16-bit word parts.

It is all inheritance from older Intel CPUs (up to the 8-bit 8080) still supported for backward compatibility. Older 8-bit CPUs (8080) had 16-bit registers divided by two.

Programs written for 8080 could access the low byte part of 16-bit registers, high byte part or the whole 16-bit register.

Perhaps, this feature was left in 8086 as a helper for easier porting.

This feature is usually not present in RISC CPUs.

Registers prefixed with R- appeared in x86-64, and those prefixed with E-—in 80386.

Thus, R-registers are 64-bit, and E-registers—32-bit.

8 more GPR’s were added in x86-64: R8-R15. .

N.B.: In the Intel manuals the byte parts of these registers are prefixed by L, e.g.: R8L, but IDA names these registers by adding the B suffix, e.g.: R8B.

RAX/EAX/AX/AL

<table>
<thead>
<tr>
<th></th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AKA accumulator. The result of a function is usually returned via this register.

RBX/EBX/BX/BL

<table>
<thead>
<tr>
<th></th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### RCX/ECX/CX/CL

<table>
<thead>
<tr>
<th>Byte number:</th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCX×8</td>
<td></td>
<td>ECX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CH</td>
<td>CL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AKA** counter: in this role it is used in REP prefixed instructions and also in shift instructions (SHL/SHR/RxL/RxR).

### RDX/EDX/DX/DL

<table>
<thead>
<tr>
<th>Byte number:</th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDX×8</td>
<td></td>
<td>EDX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DH</td>
<td>DL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### RSI/ESI/SI/SIL

<table>
<thead>
<tr>
<th>Byte number:</th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSI×8</td>
<td></td>
<td>ESI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SIL×8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AKA** “source index”. Used as source in the instructions REP MOVsx, REP CMPSx.

### RDI/EDI/DI/DIL

<table>
<thead>
<tr>
<th>Byte number:</th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDI×8</td>
<td></td>
<td>EDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIL×8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**AKA** “destination index”. Used as a pointer to the destination in the instructions REP MOVsx, REP STOSx.

### R8/R8D/R8W/R8L

<table>
<thead>
<tr>
<th>Byte number:</th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td></td>
<td>R8D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R8W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R8L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### R9/R9D/R9W/R9L

<table>
<thead>
<tr>
<th>Byte number:</th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>R9</td>
<td></td>
<td>R9D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R9W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R9L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10/R10D/R10W/R10L</td>
<td>Byte number:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7th</td>
<td>6th</td>
<td>5th</td>
<td>4th</td>
<td>3rd</td>
<td>2nd</td>
<td>1st</td>
<td>0th</td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>R10D</td>
<td>R10W</td>
<td>R10L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R11/R11D/R11W/R11L</th>
<th>Byte number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th</td>
<td>6th</td>
</tr>
<tr>
<td>R11</td>
<td>R11D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R12/R12D/R12W/R12L</th>
<th>Byte number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th</td>
<td>6th</td>
</tr>
<tr>
<td>R12</td>
<td>R12D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R13/R13D/R13W/R13L</th>
<th>Byte number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th</td>
<td>6th</td>
</tr>
<tr>
<td>R13</td>
<td>R13D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R14/R14D/R14W/R14L</th>
<th>Byte number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th</td>
<td>6th</td>
</tr>
<tr>
<td>R14</td>
<td>R14D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R15/R15D/R15W/R15L</th>
<th>Byte number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th</td>
<td>6th</td>
</tr>
<tr>
<td>R15</td>
<td>R15D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RSP/ESP/SP/SPL</th>
<th>Byte number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7th</td>
<td>6th</td>
</tr>
<tr>
<td>RSP</td>
<td>ESP</td>
</tr>
</tbody>
</table>

AKA stack pointer. Usually points to the current stack except in those cases when it is not yet initialized.
RBP/EBP/BP/BPL

<table>
<thead>
<tr>
<th>Byte number:</th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RBP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EBP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AKA frame pointer. Usually used for local variables and accessing the arguments of the function. More about it: (1.12.1 on page 67).

RIP/EIP/IP

<table>
<thead>
<tr>
<th>Byte number:</th>
<th>7th</th>
<th>6th</th>
<th>5th</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st</th>
<th>0th</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIPx64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AKA “instruction pointer”\(^3\). Usually always points to the instruction to be executed right now. Cannot be modified, however, it is possible to do this (which is equivalent):

```
MOV EAX, ...
JMP EAX
```

Or:

```
PUSH value
RET
```

CS/DS/ES/SS/FS/GS

16-bit registers containing code selector (CS), data selector (DS), stack selector (SS).

FS in win32 points to TLS, GS took this role in Linux. It is made so for faster access to the TLS and other structures like the TIB.
In the past, these registers were used as segment registers (11.6 on page 986).

Flags register

AKA EFLAGS.

\(^3\)Sometimes also called “program counter”
<table>
<thead>
<tr>
<th>Bit (mask)</th>
<th>Abbreviation (meaning)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (1)</td>
<td>CF (Carry)</td>
<td>The CLC/STC/CMC instructions are used for setting/resetting/toggling this flag.</td>
</tr>
<tr>
<td>2 (4)</td>
<td>PF (Parity)</td>
<td>Setting to 0 if the last operation's result is equal to 0. [1.25.7 \text{ on page 231}].</td>
</tr>
<tr>
<td>4 (0x10)</td>
<td>AF (Adjust)</td>
<td>Unused solely for work with BCD-numbers.</td>
</tr>
<tr>
<td>6 (0x40)</td>
<td>ZF (Zero)</td>
<td>Setting to 0 if the last operation’s result is equal to 0.</td>
</tr>
<tr>
<td>7 (0x80)</td>
<td>SF (Sign)</td>
<td>-</td>
</tr>
<tr>
<td>8 (0x100)</td>
<td>TF (Trap)</td>
<td>Used for debugging. If turned on, an exception is to be generated after each instruction’s execution.</td>
</tr>
<tr>
<td>9 (0x200)</td>
<td>IF (Interrupt enable)</td>
<td>Are interrupts enabled. The CLI/STI instructions are used for setting/resetting the flag.</td>
</tr>
<tr>
<td>10 (0x400)</td>
<td>DF (Direction)</td>
<td>A direction is set for the REP MOVsx/CMPSx/LODSx/SCASx instructions. The CLD/STD instructions are used for setting/resetting the flag. See also: [3.26 \text{ on page 624}].</td>
</tr>
<tr>
<td>11 (0x800)</td>
<td>OF (Overflow)</td>
<td>-</td>
</tr>
<tr>
<td>12, 13 (0x3000)</td>
<td>IOPL (I/O privilege level)</td>
<td>-</td>
</tr>
<tr>
<td>14 (0x4000)</td>
<td>NT (Nested task)</td>
<td>-</td>
</tr>
<tr>
<td>16 (0x10000)</td>
<td>RF (Resume)</td>
<td>Used for debugging. The CPU ignores the hardware breakpoint in DRx if the flag is set.</td>
</tr>
<tr>
<td>17 (0x20000)</td>
<td>VM (Virtual 8086 mode)</td>
<td>-</td>
</tr>
<tr>
<td>18 (0x40000)</td>
<td>AC (Alignment check)</td>
<td>-</td>
</tr>
<tr>
<td>19 (0x80000)</td>
<td>VIF (Virtual interrupt)</td>
<td>-</td>
</tr>
<tr>
<td>20 (0x100000)</td>
<td>VIP (Virtual interrupt pending)</td>
<td>-</td>
</tr>
<tr>
<td>21 (0x200000)</td>
<td>ID (Identification)</td>
<td>-</td>
</tr>
</tbody>
</table>

All the rest flags are reserved.

### 1.3 FPU registers

8 80-bit registers working as a stack: ST(0)-ST(7). N.B.: IDA calls ST(0) as just ST. Numbers are stored in the IEEE 754 format.

**long double** value format:

\[
\begin{array}{cccccc}
S & \text{exponent} & I & \text{mantissa or fraction} & 0
\end{array}
\]

\[S — \text{sign}, I — \text{integer part}\]

**Control Word**

Register controlling the behavior of the **FPU**.
<table>
<thead>
<tr>
<th>Bit</th>
<th>Abbreviation (meaning)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>IM (Invalid operation Mask)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>DM (Denormalized operand Mask)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ZM (Zero divide Mask)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>OM (Overflow Mask)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>UM (Underflow Mask)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>PM (Precision Mask)</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>IEM (Interrupt Enable Mask)</td>
<td>Exceptions enabling, 1 by default (disabled)</td>
</tr>
</tbody>
</table>
| 8, 9| PC (Precision Control) | 00 — 24 bits (REAL4)  
|     |                        | 10 — 53 bits (REAL8)  
|     |                        | 11 — 64 bits (REAL10) |
| 10, 11| RC (Rounding Control) | 00 — (by default) round to nearest  
|      |                        | 01 — round toward $-\infty$  
|      |                        | 10 — round toward $+\infty$  
|      |                        | 11 — round toward 0 |
| 12  | IC (Infinity Control) | 0 — (by default) treat $+\infty$ and $-\infty$ as unsigned  
|     |                        | 1 — respect both $+\infty$ and $-\infty$ |

The PM, UM, OM, ZM, DM, IM flags define if to generate exception in the case of a corresponding error.

**Status Word**

Read-only register.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Abbreviation (meaning)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>B (Busy)</td>
<td>Is FPU do something (1) or results are ready (0)</td>
</tr>
<tr>
<td>14</td>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>13, 12, 11</td>
<td>TOP</td>
<td>points to the currently zeroth register</td>
</tr>
<tr>
<td>10</td>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>C0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>IR (Interrupt Request)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>SF (Stack Fault)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>P (Precision)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>U (Underflow)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>O (Overflow)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Z (Zero)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>D (Denormalized)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>I (Invalid operation)</td>
<td></td>
</tr>
</tbody>
</table>

The SF, P, U, O, Z, D, I bits signal about exceptions.

About the C3, C2, C1, C0 you can read more here: ([1.25.7 on page 231](#)).

**N.B.:** When ST(x) is used, the FPU adds $x$ to TOP (by modulo 8) and that is how it gets the internal register’s number.

**Tag Word**

The register has current information about the usage of numbers registers.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Abbreviation (meaning)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15, 14</td>
<td>tag(7)</td>
</tr>
<tr>
<td>13, 12</td>
<td>tag(6)</td>
</tr>
<tr>
<td>11, 10</td>
<td>tag(5)</td>
</tr>
<tr>
<td>9, 8</td>
<td>tag(4)</td>
</tr>
<tr>
<td>7, 6</td>
<td>tag(3)</td>
</tr>
<tr>
<td>5, 4</td>
<td>tag(2)</td>
</tr>
<tr>
<td>3, 2</td>
<td>tag(1)</td>
</tr>
<tr>
<td>1, 0</td>
<td>tag(0)</td>
</tr>
</tbody>
</table>

Each tag contains information about a physical FPU register (R(x)), not logical (ST(x)).

For each tag:

- 00 — The register contains a non-zero value
• 01 — The register contains 0
• 10 — The register contains a special value (NAN\(^4\), \(\infty\), or denormal)
• 11 — The register is empty

### .1.4 SIMD registers

**MMX registers**

8 64-bit registers: MM0..MM7.

**SSE and AVX registers**

SSE: 8 128-bit registers: XMM0..XMM7. In the x86-64 8 more registers were added: XMM8..XMM15. AVX is the extension of all these registers to 256 bits.

### .1.5 Debugging registers

Used for hardware breakpoints control.

• DR0 — address of breakpoint #1
• DR1 — address of breakpoint #2
• DR2 — address of breakpoint #3
• DR3 — address of breakpoint #4
• DR6 — a cause of break is reflected here
• DR7 — breakpoint types are set here

**DR6**

<table>
<thead>
<tr>
<th>Bit (mask)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (1)</td>
<td>B0 — breakpoint #1 has been triggered</td>
</tr>
<tr>
<td>1 (2)</td>
<td>B1 — breakpoint #2 has been triggered</td>
</tr>
<tr>
<td>2 (4)</td>
<td>B2 — breakpoint #3 has been triggered</td>
</tr>
<tr>
<td>3 (8)</td>
<td>B3 — breakpoint #4 has been triggered</td>
</tr>
<tr>
<td>13 (0x2000)</td>
<td>BD — modification attempt of one of the DRx registers. (\text{may be raised if GD is enabled})</td>
</tr>
<tr>
<td>14 (0x4000)</td>
<td>BS — single step breakpoint (TF flag has been set in EFLAGS). Highest priority. Other bits may also be set.</td>
</tr>
<tr>
<td>15 (0x8000)</td>
<td>BT (task switch flag)</td>
</tr>
</tbody>
</table>

N.B. A single step breakpoint is a breakpoint which occurs after each instruction. It can be enabled by setting TF in EFLAGS (\.1.2 on page 1008).

**DR7**

Breakpoint types are set here.

\(^4\)Not a Number
<table>
<thead>
<tr>
<th>Bit (mask)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (1)</td>
<td>L0 — enable breakpoint #1 for the current task</td>
</tr>
<tr>
<td>1 (2)</td>
<td>G0 — enable breakpoint #1 for all tasks</td>
</tr>
<tr>
<td>2 (4)</td>
<td>L1 — enable breakpoint #2 for the current task</td>
</tr>
<tr>
<td>3 (8)</td>
<td>G1 — enable breakpoint #2 for all tasks</td>
</tr>
<tr>
<td>4 (0x10)</td>
<td>L2 — enable breakpoint #3 for the current task</td>
</tr>
<tr>
<td>5 (0x20)</td>
<td>G2 — enable breakpoint #3 for all tasks</td>
</tr>
<tr>
<td>6 (0x40)</td>
<td>L3 — enable breakpoint #4 for the current task</td>
</tr>
<tr>
<td>7 (0x80)</td>
<td>G3 — enable breakpoint #4 for all tasks</td>
</tr>
<tr>
<td>8 (0x100)</td>
<td>LE — not supported since P6</td>
</tr>
<tr>
<td>9 (0x200)</td>
<td>GE — not supported since P6</td>
</tr>
<tr>
<td>13 (0x2000)</td>
<td>GD — exception is to be raised if any MOV instruction tries to modify one of the DRx registers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16,17 (0x30000)</td>
<td>breakpoint #1: R/W — type</td>
</tr>
<tr>
<td>18,19 (0xC0000)</td>
<td>breakpoint #1: LEN — length</td>
</tr>
<tr>
<td>20,21 (0x300000)</td>
<td>breakpoint #2: R/W — type</td>
</tr>
<tr>
<td>22,23 (0xC00000)</td>
<td>breakpoint #2: LEN — length</td>
</tr>
<tr>
<td>24,25 (0x3000000)</td>
<td>breakpoint #3: R/W — type</td>
</tr>
<tr>
<td>26,27 (0xC0000000)</td>
<td>breakpoint #3: LEN — length</td>
</tr>
<tr>
<td>28,29 (0x300000000)</td>
<td>breakpoint #4: R/W — type</td>
</tr>
<tr>
<td>30,31 (0xC000000000)</td>
<td>breakpoint #4: LEN — length</td>
</tr>
</tbody>
</table>

The breakpoint type is to be set as follows (R/W):
- 00 — instruction execution
- 01 — data writes
- 10 — I/O reads or writes (not available in user-mode)
- 11 — on data reads or writes

N.B.: breakpoint type for data reads is absent, indeed.

Breakpoint length is to be set as follows (LEN):
- 00 — one-byte
- 01 — two-byte
- 10 — undefined for 32-bit mode, eight-byte in 64-bit mode
- 11 — four-byte

### .1.6 Instructions

Instructions marked as (M) are not usually generated by the compiler: if you see one of them, it is probably a hand-written piece of assembly code, or a compiler intrinsic (11.3 on page 981).

Only the most frequently used instructions are listed here. You can read 12.1.4 on page 999 for a full documentation.

Do you have to know all instruction's opcodes by heart? No, only those which are used for code patching (11.1.1 on page 980). All the rest of the opcodes don’t need to be memorized.

#### Prefixes

**LOCK** forces CPU to make exclusive access to the RAM in multiprocessor environment. For the sake of simplification, it can be said that when an instruction with this prefix is executed, all other CPUs in a multiprocessor system are stopped. Most often it is used for critical sections, semaphores, mutexes. Commonly used with ADD, AND, BTR, BTS, CMPXCHG, OR, XADD, XOR. You can read more about critical sections here (6.5.4 on page 777).

**REP** is used with the MOVSx and STOSx instructions: execute the instruction in a loop, the counter is located in the CX/ECX/RCX register. For a detailed description, read more about the MOVSx (.1.6 on page 1015) and STOSx (.1.6 on page 1016) instructions.

The instructions prefixed by REP are sensitive to the DF flag, which is used to set the direction.
REPE/REPNE (AKA REPZ/REPNZ) used with CMPSx and SCASx instructions: execute the last instruction in a loop, the count is set in the CX/ECX/RCX register. It terminates prematurely if ZF is 0 (REPE) or if ZF is 1 (REPNE).

For a detailed description, you can read more about the CMPSx (1.6 on page 1018) and SCASx (1.6 on page 1016) instructions.

Instructions prefixed by REPE/REPNE are sensitive to the DF flag, which is used to set the direction.

Most frequently used instructions

These can be memorized in the first place.

ADC (add with carry) add values, increment the result if the CF flag is set. ADC is often used for the addition of large values, for example, to add two 64-bit values in a 32-bit environment using two ADD and ADC instructions. For example:

```c
; work with 64-bit values: add val1 to val2.
; .lo means lowest 32 bits, .hi means highest.
ADD val1.lo, val2.lo
ADC val1.hi, val2.hi ; use CF that was set or cleared at the previous instruction
```

One more example: 1.34 on page 392.

ADD add two values

AND logical “and”

CALL call another function:

```c
PUSH address_after_CALL_instruction; JMP label
```

CMP compare values and set flags, the same as SUB but without writing the result

DEC decrement. Unlike other arithmetic instructions, DEC doesn’t modify CF flag.

IMUL signed multiply IMUL often used instead of MUL, read more about it: 2.2.1 on page 450.

INC increment. Unlike other arithmetic instructions, INC doesn’t modify CF flag.

JAE AKA JNC: jump if above or equal (unsigned): CF=0

JA AKA JNBE: jump if greater (unsigned): CF=0 and ZF=0

JBE jump if lesser or equal (unsigned): CF=1 or ZF=1

JB AKA JC: jump if below (unsigned): CF=1

JC AKA JB: jump if CF=1

JE AKA JZ: jump if equal or zero: ZF=1

JGE jump if greater or equal (signed): SF=OF

JG jump if greater (signed): ZF=0 and SF=OF

JLE jump if lesser or equal (signed): ZF=1 or SF≠OF

JL jump if lesser (signed): SF≠OF

JNAE AKA JC: jump if not above or equal (unsigned) CF=1

JNA jump if not above (unsigned) CF=1 and ZF=1

JNBE jump if not below or equal (unsigned): CF=0 and ZF=0

JNB AKA JNC: jump if not below (unsigned): CF=0

JNC AKA JAE: jump CF=0 synonymous to JNB.

JNE AKA JNZ: jump if not equal or not zero: ZF=0

1013
JNGE jump if not greater or equal (signed): SF≠OF
JNG jump if not greater (signed): ZF=1 or SF≠OF
JNLE jump if not lesser (signed): ZF=0 and SF=OF
JNL jump if not lesser (signed): SF=OF
JNO jump if not overflow: OF=0
JNS jump if SF flag is cleared
JNZ AKA JNE: jump if not equal or not zero: ZF=0
JO jump if overflow: OF=1
JPO jump if PF flag is cleared (Jump Parity Odd)
JP AKA JPE: jump if PF flag is set
JS jump if SF flag is set
JZ AKA JE: jump if equal or zero: ZF=1

LAHF copy some flag bits to AH:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>4</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF</td>
<td>ZF</td>
<td>AF</td>
<td>PF</td>
<td>CF</td>
</tr>
</tbody>
</table>

This instruction is often used in FPU-related code.

LEAVE equivalent of the MOV ESP, EBX and POP EBP EBP instruction pair — in other words, this instruction sets the stack pointer (ESP) back and restores the EBP register to its initial state.

LEA (Load Effective Address) form an address

This instruction was intended not for summing values and multiplication but for forming an address, e.g., for calculating the address of an array element by adding the array address, element index, with multiplication of element size³.

So, the difference between MOV and LEA is that MOV forms a memory address and loads a value from memory or stores it there, but LEA just forms an address.

But nevertheless, it is can be used for any other calculations.

LEA is convenient because the computations performed by it does not alter CPU flags. This may be very important for OOE processors (to create less data dependencies).

Aside from this, starting at least at Pentium, LEA instruction is executed in 1 cycle.

```c
int f(int a, int b)
{
    return a*8+b;
};
```

Listing 1: Optimizing MSVC 2010

```assembly
_a$ = 8 ; size = 4
_b$ = 12 ; size = 4
_f PROC
    mov eax, DWORD PTR _b$[esp-4]
    mov ecx, DWORD PTR _a$[esp-4]
    lea eax, DWORD PTR [eax+ecx*8]
    ret 0
_f ENDP
```

Intel C++ uses LEA even more:

```c
int f1(int a)
{
    return a*13;
};
```

³See also: wikipedia
Listing 2: Intel C++ 2011

```
    _f1    PROC NEAR
  mov    ecx, DWORD PTR [4+esp] ; ecx = a
  lea    edx, DWORD PTR [ecx+ecx*8] ; edx = a*9
  lea    eax, DWORD PTR [edx+ecx*4] ; eax = a*9 + a*4 = a*13
  ret
```

These two instructions performs faster than one IMUL.

**MOVSB/MOVSW/MOVSD/MOVSQ** copy byte/ 16-bit word/ 32-bit word/ 64-bit word from the address which is in SI/ESI/RSI into the address which is in DI/EDI/RDI.

Together with the REP prefix, it is to be repeated in a loop, the count is to be stored in the CX/ECX/RCX register: it works like memcpy() in C. If the block size is known to the compiler in the compile stage, memcpy() is often inlined into a short code fragment using REP MOVSx, sometimes even as several instructions.

The memcpy(EDI, ESI, 15) equivalent is:

```
; copy 15 bytes from ESI to EDI
CLD  ; set direction to forward
MOV ECX, 3
REP MOVSD  ; copy 12 bytes
MOVSW  ; copy 2 more bytes
MOVSB  ; copy remaining byte
```

( Supposedly, it works faster than copying 15 bytes using just one REP MOVSB).

**MOVZX** load and clear all other bitsi see also: (1.23.1 on page 201)

**NOT** op1: $op1 = \neg op1$. logical inversion Important feature—the instruction doesn’t change flags.

**OR** logical “or”

**POP** get a value from the stack: value=SS:[ESP]; ESP=ESP+4 (or 8)

**PUSH** push a value into the stack: ESP=ESP-4 (or 8); SS:[ESP]=value

**RET** return from subroutine: POP tmp; JMP tmp.

In fact, RET is an assembly language macro, in Windows and *NIX environment it is translated into RETN (“return near”) or, in MS-DOS times, where the memory was addressed differently (11.6 on page 986), into RETF (“return far”).

RET can have an operand. Then it works like this:

POP tmp; ADD ESP op1; JMP tmp. RET with an operand usually ends functions in the stdcall calling convention, see also: 6.1.2 on page 725.

**SAHF** copy bits from AH to CPU flags:

1015
This instruction is often used in FPU-related code.

**SBB (subtraction with borrow)** subtract values, decrement the result if the CF flag is set. SBB is often used for subtraction of large values, for example, to subtract two 64-bit values in 32-bit environment using two SUB and SBB instructions. For example:

```assembly
; work with 64-bit values: subtract val2 from val1.
; .lo means lowest 32 bits, .hi means highest.
SUB val1.lo, val2.lo
SBB val1.hi, val2.hi ; use CF that was set or cleared at the previous instruction
```

One more example: 1.34 on page 392.

**SCASB/SCASW/SCASD/SCASQ (M)** compare byte/ 16-bit word/ 32-bit word/ 64-bit word that’s stored in AX/EAX/RAX with a variable whose address is in DI/EDI/RDI. Set flags as CMP does.

This instruction is often used with the REPNE prefix: continue to scan the buffer until a special value stored in AX/EAX/RAX is found. Hence “NE” in REPNE: continue to scan while the compared values are not equal and stop when equal.

It is often used like the strlen() C standard function, to determine an ASCIIZ string’s length:

Example:

```assembly
lea    edi, string
mov    ecx, 0FFFFFFFFh ; scan 2^32-1 bytes, i.e., almost infinitely
xor    eax, eax       ; 0 is the terminator
repne scasb
add    edi, 0FFFFFFFFh ; correct it

; now EDI points to the last character of the ASCIIZ string.

; lets determine string length
; current ECX = -1-strlen
not    ecx
dec    ecx

; now ECX contain string length
```

If we use a different AX/EAX/RAX value, the function acts like the memchr() standard C function, i.e., it finds a specific byte.

**SHL** shift value left

**SHR** shift value right:

![Shift Diagram](image)

These instructions are frequently used for multiplication and division by $2^n$. Another very frequent application is processing bit fields: 1.28 on page 303.

**SHRD** op1, op2, op3: shift value in op2 right by op3 bits, taking bits from op1.

Example: 1.34 on page 392.

**STOSB/STOSW/STOSD/STOSQ** store byte/ 16-bit word/ 32-bit word/ 64-bit word from AX/EAX/RAX into the address which is in DI/EDI/RDI.
Together with the REP prefix, it is to be repeated in a loop, the counter is in the CX/ECX/RCX register: it works like memset() in C. If the block size is known to the compiler on compile stage, memset() is often inlined into a short code fragment using REP MOVsx, sometimes even as several instructions.

memset(EDI, 0xAA, 15) equivalent is:

```assembly
; store 15 0xAA bytes to EDI
CLD ; set direction to forward
MOV EAX, 0AAAAAAAAh
MOV ECX, 3
REP STOSD ; write 12 bytes
STOSW ; write 2 more bytes
STOSB ; write remaining byte
```

(Supposedly, it works faster than storing 15 bytes using just one REP STOSB).

**SUB** subtract values. A frequently occurring pattern is SUB reg, reg, which implies zeroing of reg.

**TEST** same as AND but without saving the result, see also: 1.28 on page 303

**XOR** op1, op2: XOR\(^\text{e}\) values. \(op1 = op1 \oplus op2\). A frequently occurring pattern is XOR reg, reg, which implies zeroing of reg. See also: 2.6 on page 456.

**Less frequently used instructions**

**BSF** bit scan forward, see also: 1.36.2 on page 417

**BSR** bit scan reverse

**BSWAP** (byte swap), change value endianness.

**BTC** bit test and complement

**BTR** bit test and reset

**BTS** bit test and set

**BT** bit test

**CBW/CWD/CWDE/CDQ/CDQE** Sign-extend value:

- **CBW** convert byte in AL to word in AX
- **CWD** convert word in AX to doubleword in DX:AX
- **CWDE** convert word in AX to doubleword in EAX
- **CDQ** convert doubleword in EAX to quadword in EDX:EAX
- **CDQE** (x64) convert doubleword in EAX to quadword in RAX

These instructions consider the value’s sign, extending it to high part of the newly constructed value. See also: 1.34.5 on page 401.

Interestingly to know these instructions was initially named as SEX (Sign EXTend), as Stephen P. Morse (one of Intel 8086 CPU designers) wrote in [Stephen P. Morse, *The 8086 Primer*, (1980)]^7:

> The process of stretching numbers by extending the sign bit is called sign extension. The 8086 provides instructions (Fig. 3.29) to facilitate the task of sign extension. These instructions were initially named SEX (sign extend) but were later renamed to the more conservative CBW (convert byte to word) and CWD (convert word to double word).

**CLD** clear DF flag.

**CLI** (M) clear IF flag.

**CMC** (M) toggle CF flag

**CMOVcc** conditional MOV: load if the condition is true. The condition codes are the same as in the Jcc instructions (.1.6 on page 1013).

---

\(^{e}\)eXclusive OR

^7Also available as [https://archive.org/details/The8086Primer](https://archive.org/details/The8086Primer)
CMPSB/CMPSW/CMPSD/CMPSQ (M) compare byte/ 16-bit word/ 32-bit word/ 64-bit word from the address which is in SI/ESI/RSI with the variable at the address stored in DI/EDI/RDI. Set flags as CMP does.

Together with the REP prefix, it is to be repeated in a loop, the counter is stored in the CX/ECX/RCX register, the process will run until the ZF flag is zero (e.g., until the compared values are equal to each other, hence “E” in REPE).

It works like memcmp() in C.

Example from the Windows NT kernel (WRK v1.2):

Listing 3: base\ntos\rtl\i386\movemem.asm

; ULONG
; RtlCompareMemory (;
; IN PVOID Source1,
; IN PVOID Source2,
; IN ULONG Length
;)
;
; Routine Description:
;
; This function compares two blocks of memory and returns the number of bytes that compared equal.
;
; Arguments:
;
; Source1 (esp+4) - Supplies a pointer to the first block of memory to compare.
;
; Source2 (esp+8) - Supplies a pointer to the second block of memory to compare.
;
; Length (esp+12) - Supplies the Length, in bytes, of the memory to be compared.
;
; Return Value:
;
; The number of bytes that compared equal is returned as the function value. If all bytes compared equal, then the length of the original block of memory is returned.
;
;--
RcmSource1 equ [esp+12]
RcmSource2 equ [esp+16]
RcmLength equ [esp+20]

CODE_ALIGNMENT
cPublicProc_RtlCompareMemory,3
cPublicFpo 3,0

    push    esi            ; save registers
    push    edi            ;
    cld              ; clear direction
    mov    esi,RcmSource1 ; (esi) -> first block to compare
    mov    edi,RcmSource2 ; (edi) -> second block to compare

    ; Compare dwords, if any.
    ;
    rcm10: mov    ecx,RcmLength ; (ecx) = length in bytes
            shr    ecx,2 ; (ecx) = length in dwords
            jz     rcm20 ; no dwords, try bytes
            repe   cmpsd ; compare dwords
            jnz   rcm40 ; mismatch, go find byte

    ; Compare residual bytes, if any.
    ;

1018
rcm20:  mov   ecx,RcmLength          ; (ecx) = length in bytes
          and  ecx,3                ; (ecx) = length mod 4
          jz   rcm30                ; 0 odd bytes, go do dwords
          repe cmpsb                ; compare odd bytes
          jnz  rcm50                ; mismatch, go report how far we got

; All bytes in the block match.

rcm30:   mov   eax,RcmLength          ; set number of matching bytes
          pop  edi                  ; restore registers
          pop  esi
          stdRET _RtlCompareMemory

; When we come to rcm40, esi (and edi) points to the dword after the
; one which caused the mismatch. Back up 1 dword and find the byte.
; Since we know the dword didn't match, we can assume one byte won't.

rcm40:   sub   esi,4                ; back up
          sub   edi,4                ; back up
          mov   ecx,5                ; ensure that ecx doesn't count out
          repe cmpsb                ; find mismatch byte

; When we come to rcm50, esi points to the byte after the one that
; did not match, which is TWO after the last byte that did match.

rcm50:   dec   esi                  ; back up
          sub   esi,RcmSource1        ; compute bytes that matched
          mov   eax,esi
          pop  edi                  ; restore registers
          pop  esi
          stdRET _RtlCompareMemory

stdENDP _RtlCompareMemory

N.B.: this function uses a 32-bit word comparison (CMPSD) if the block size is a multiple of 4, or
per-byte comparison (CMPSB) otherwise.

CPUID  get information about the CPU’s features. see also: (1.30.6 on page 366).

DIV  unsigned division

IDIV  signed division

INT  (M): INT x is analogous to PUSHF; CALL dword ptr [x*4] in 16-bit environment. It was widely
used in MS-DOS, functioning as a syscall vector. The registers AX/BX/CX/DX/SI/DI were filled with
the arguments and then the flow jumped to the address in the Interrupt Vector Table (located at the
beginning of the address space). It was popular because INT has a short opcode (2 bytes) and the
program which needs some MS-DOS services is not bother to determine the address of the service’s
entry point. The interrupt handler returns the control flow to caller using the IRET instruction.

The most busy MS-DOS interrupt number was 0x21, serving a huge part of its API. See also: [Ralf
Brown Ralf Brown’s Interrupt List], for the most comprehensive interrupt lists and other MS-DOS
information.

In the post-MS-DOS era, this instruction was still used as syscall both in Linux and Windows (6.3 on
page 738), but was later replaced by the SYSENTER or SYSCALL instructions.

INT3  (M): this instruction is somewhat close to INT, it has its own 1-byte opcode (0xCC), and is actively
used while debugging. Often, the debuggers just write the 0xCC byte at the address of the breakpoint
to be set, and when an exception is raised, the original byte is restored and the original instruction
at this address is re-executed.
As of Windows NT, an EXCEPTION_BREAKPOINT exception is to be raised when the CPU executes this
instruction. This debugging event may be intercepted and handled by a host debugger, if one is
loaded. If it is not loaded, Windows offers to run one of the registered system debuggers. If MSVS\textsuperscript{8} is installed, its debugger may be loaded and connected to the process. In order to protect from reverse engineering, a lot of anti-debugging methods check integrity of the loaded code.

MSVC has compiler intrinsic for the instruction: \_\_\_\_debugbreak()\textsuperscript{9}.

There is also a win32 function in kernel32.dll named DebugBreak()\textsuperscript{10}, which also executes INT 3.

IN (M) input data from port. The instruction usually can be seen in OS drivers or in old MS-DOS code, for example (8.8.3 on page 835).

IRET: was used in the MS-DOS environment for returning from an interrupt handler after it was called by the INT instruction. Equivalent to POP tmp; POPF; JMP tmp.

LOOP (M) decrement CX/ECX/RCX, jump if it is still not zero.

LOOP instruction was often used in DOS-code which works with external devices. To add small delay, this was done:

\begin{verbatim}
    MOV CX, nnnn
    LOOP LABEL
\end{verbatim}

Drawback is obvious: length of delay depends on CPU speed.

OUT (M) output data to port. The instruction usually can be seen in OS drivers or in old MS-DOS code, for example (8.8.3 on page 835).

POPA (M) restores values of (R|E)DI, (R|E)SI, (R|E)BP, (R|E)BX, (R|E)DX, (R|E)CX, (R|E)AX registers from the stack.

POPCNT population count. Counts the number of 1 bits in the value.

See: 2.7 on page 460.

POPF restore flags from the stack (AKA EFLAGS register)

PUSHA (M) pushes the values of the (R|E)AX, (R|E)CX, (R|E)DX, (R|E)BX, (R|E)BP, (R|E)SI, (R|E)DI registers to the stack.

PUSHF push flags (AKA EFLAGS register)

RCL (M) rotate left via CF flag:

\begin{verbatim}
    7 6 5 4 3 2 1 0
    CF
\end{verbatim}

\begin{verbatim}
    CF 7 6 5 4 3 2 1 0
\end{verbatim}

RCL (M) rotate right via CF flag:

\begin{verbatim}
    7 6 5 4 3 2 1 0
    CF
\end{verbatim}

\begin{verbatim}
    CF 7 6 5 4 3 2 1 0
\end{verbatim}

ROL/ROR (M) cyclic shift

ROL: rotate left:

\begin{verbatim}
    7 6 5 4 3 2 1 0
    CF
\end{verbatim}

\begin{verbatim}
    CF 7 6 5 4 3 2 1 0
\end{verbatim}

ROR: rotate right:

\begin{verbatim}
    7 6 5 4 3 2 1 0
    CF
\end{verbatim}

\begin{verbatim}
    CF 7 6 5 4 3 2 1 0
\end{verbatim}

\textsuperscript{8}Microsoft Visual Studio
\textsuperscript{9}MSDN
\textsuperscript{10}MSDN
Despite the fact that almost all CPUs have these instructions, there are no corresponding operations in C/C++, so the compilers of these PLs usually do not generate these instructions.

For the programmer’s convenience, at least MSVC has the pseudofunctions (compiler intrinsics) \_rotl() and \_rotr()\(^\text{11}\), which are translated by the compiler directly to these instructions.

SAL Arithmetic shift left, synonymous to SHL

SAR Arithmetic shift right

Hence, the sign bit always stays at the place of the MSB.

SETcc op: load 1 to operand (byte only) if the condition is true or zero otherwise. The condition codes are the same as in the Jcc instructions ( .1.6 on page 1013).

STC (M) set CF flag

STD (M) set DF flag. This instruction is not generated by compilers and generally rare. For example, it can be found in the ntoskrnl.exe Windows kernel file, in the hand-written memory copy routines.

STI (M) set IF flag

SYSCALL (AMD) call syscall ( 6.3 on page 738)

SYSENTER (Intel) call syscall ( 6.3 on page 738)

UD2 (M) undefined instruction, raises exception. Used for testing.

XCHG (M) exchange the values in the operands

This instruction is rare: compilers don’t generate it, because starting at Pentium, XCHG with address in memory in operand executes as if it has LOCK prefix ([Michael Abrash, Graphics Programming Black Book, 1997chapter 19]). Perhaps, Intel engineers did so for compatibility with synchronizing primitives. Hence, XCHG starting at Pentium can be slow. On the other hand, XCHG was very popular in assembly language programmers. So if you see XCHG in code, it can be a sign that this piece of code is written manually. However, at least Borland Delphi compiler generates this instruction.

FPU instructions

-R suffix in the mnemonic usually implies that the operands are reversed, -P suffix implies that one element is popped from the stack after the instruction’s execution, -PP suffix implies that two elements are popped.

-P instructions are often useful when we do not need the value in the FPU stack to be present anymore after the operation.

FABS replace value in ST(0) by absolute value in ST(0)

FADD op: ST(0)=op+ST(0)

FADD ST(0), ST(i): ST(0)=ST(0)+ST(i)

FADDP ST(1)=ST(0)+ST(1); pop one element from the stack, i.e., the values in the stack are replaced by their sum

FCHS ST(0)=-ST(0)

FCOM compare ST(0) with ST(1)

FCOM op: compare ST(0) with op

FCOMP compare ST(0) with ST(1); pop one element from the stack

\(^{11}\)MSDN
FCOMPP  compare ST(0) with ST(1); pop two elements from the stack
FDIVR  op: ST(0)=op/ST(0)
FDIVR ST(i), ST(j): ST(i)=ST(j)/ST(i)
FDIVR op: ST(0)=op/ST(0); pop one element from the stack
FDIVRP ST(i), ST(j): ST(i)=ST(j)/ST(i); pop one element from the stack
FDIV op: ST(0)=ST(0)/op
FDIV ST(i), ST(j): ST(i)=ST(i)/ST(j)
FDIVP ST(1)=ST(0)/ST(1); pop one element from the stack, i.e., the dividend and divisor values in the stack are replaced by quotient
FILD  op: convert integer and push it to the stack.
FIST  op: convert ST(0) to integer op
FISTP op: convert ST(0) to integer op; pop one element from the stack
FLD1 push 1 to stack
FLDCW op: load FPU control word ( .1.3 on page 1009) from 16-bit op.
FLDZ push zero to stack
FLD op: push op to the stack.
FMUL op: ST(0)=ST(0)*op
FMULST(i), ST(j): ST(i)=ST(i)*ST(j)
FMULP op: ST(0)=ST(0)*op; pop one element from the stack
FMULP ST(i), ST(j): ST(i)=ST(i)*ST(j); pop one element from the stack
FSINCOS : tmp=ST(0); ST(1)=sin(tmp); ST(0)=cos(tmp)
FSQRT : $ST(0) = \sqrt{ST(0)}$
FSTCW op: store FPU control word ( .1.3 on page 1009) into 16-bit op after checking for pending exceptions.
FNSTCW op: store FPU control word ( .1.3 on page 1009) into 16-bit op.
FSTSW op: store FPU status word ( .1.3 on page 1010) into 16-bit op after checking for pending exceptions.
FNSTSW op: store FPU status word ( .1.3 on page 1010) into 16-bit op.
FST op: copy ST(0) to op
FSTP op: copy ST(0) to op; pop one element from the stack
FSUBR op: ST(0)=op-ST(0)
FSUBR ST(0), ST(i): ST(0)=ST(i)-ST(0)
FSUBRP ST(1)=ST(0)-ST(1); pop one element from the stack, i.e., the value in the stack is replaced by the difference
FSUB op: ST(0)=ST(0)-op
FSUB ST(0), ST(i): ST(0)=ST(0)-ST(i)
FSUBP ST(1)=ST(1)-ST(0); pop one element from the stack, i.e., the value in the stack is replaced by the difference
FUCOM ST(i): compare ST(0) and ST(i)
FUCOM compare ST(0) and ST(1)
FUCOMP compare ST(0) and ST(1); pop one element from stack.
FUCOMPP compare ST(0) and ST(1); pop two elements from stack.

The instructions perform just like FCOM, but an exception is raised only if one of the operands is SNaN, while QNaN numbers are processed smoothly.

FXCH ST(i) exchange values in ST(0) and ST(i)
**FXCH** exchange values in ST(0) and ST(1)

**Instructions having printable ASCII opcode**

(In 32-bit mode).

These can be suitable for shellcode construction. See also: 8.14.1 on page 887.

<table>
<thead>
<tr>
<th>ASCII character</th>
<th>hexadecimal code</th>
<th>x86 instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
<td>XOR</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>XOR</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>XOR</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>XOR</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>XOR</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>XOR</td>
</tr>
<tr>
<td>7</td>
<td>37</td>
<td>AAA</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td>CMP</td>
</tr>
<tr>
<td>9</td>
<td>39</td>
<td>CMP</td>
</tr>
<tr>
<td>:</td>
<td>3a</td>
<td>CMP</td>
</tr>
<tr>
<td>;</td>
<td>3b</td>
<td>CMP</td>
</tr>
<tr>
<td>&lt;</td>
<td>3c</td>
<td>CMP</td>
</tr>
<tr>
<td>=</td>
<td>3d</td>
<td>CMP</td>
</tr>
<tr>
<td>?</td>
<td>3f</td>
<td>AAS</td>
</tr>
<tr>
<td>@</td>
<td>40</td>
<td>INC</td>
</tr>
<tr>
<td>A</td>
<td>41</td>
<td>INC</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
<td>INC</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
<td>INC</td>
</tr>
<tr>
<td>D</td>
<td>44</td>
<td>INC</td>
</tr>
<tr>
<td>E</td>
<td>45</td>
<td>INC</td>
</tr>
<tr>
<td>F</td>
<td>46</td>
<td>INC</td>
</tr>
<tr>
<td>G</td>
<td>47</td>
<td>INC</td>
</tr>
<tr>
<td>H</td>
<td>48</td>
<td>DEC</td>
</tr>
<tr>
<td>I</td>
<td>49</td>
<td>DEC</td>
</tr>
<tr>
<td>J</td>
<td>4a</td>
<td>DEC</td>
</tr>
<tr>
<td>K</td>
<td>4b</td>
<td>DEC</td>
</tr>
<tr>
<td>L</td>
<td>4c</td>
<td>DEC</td>
</tr>
<tr>
<td>M</td>
<td>4d</td>
<td>DEC</td>
</tr>
<tr>
<td>N</td>
<td>4e</td>
<td>DEC</td>
</tr>
<tr>
<td>O</td>
<td>4f</td>
<td>DEC</td>
</tr>
<tr>
<td>P</td>
<td>50</td>
<td>PUSH</td>
</tr>
<tr>
<td>Q</td>
<td>51</td>
<td>PUSH</td>
</tr>
<tr>
<td>R</td>
<td>52</td>
<td>PUSH</td>
</tr>
<tr>
<td>S</td>
<td>53</td>
<td>PUSH</td>
</tr>
<tr>
<td>T</td>
<td>54</td>
<td>PUSH</td>
</tr>
<tr>
<td>U</td>
<td>55</td>
<td>PUSH</td>
</tr>
<tr>
<td>V</td>
<td>56</td>
<td>PUSH</td>
</tr>
<tr>
<td>W</td>
<td>57</td>
<td>PUSH</td>
</tr>
<tr>
<td>X</td>
<td>58</td>
<td>POP</td>
</tr>
<tr>
<td>Y</td>
<td>59</td>
<td>POP</td>
</tr>
<tr>
<td>Z</td>
<td>5a</td>
<td>POP</td>
</tr>
<tr>
<td>[</td>
<td>5b</td>
<td>POP</td>
</tr>
<tr>
<td>\</td>
<td>5c</td>
<td>POP</td>
</tr>
<tr>
<td>]</td>
<td>5d</td>
<td>POP</td>
</tr>
<tr>
<td>^</td>
<td>5e</td>
<td>POP</td>
</tr>
<tr>
<td>_</td>
<td>5f</td>
<td>POP</td>
</tr>
<tr>
<td>`</td>
<td>60</td>
<td>PUSHA</td>
</tr>
<tr>
<td>a</td>
<td>61</td>
<td>POPA</td>
</tr>
<tr>
<td>b</td>
<td>62</td>
<td>IMUL</td>
</tr>
<tr>
<td>c</td>
<td>63</td>
<td>IMUL</td>
</tr>
<tr>
<td>d</td>
<td>64</td>
<td>IMUL</td>
</tr>
<tr>
<td>e</td>
<td>65</td>
<td>IMUL</td>
</tr>
<tr>
<td>f</td>
<td>66</td>
<td>IMUL</td>
</tr>
<tr>
<td>g</td>
<td>67</td>
<td>IMUL</td>
</tr>
<tr>
<td>h</td>
<td>68</td>
<td>PUSH</td>
</tr>
<tr>
<td>i</td>
<td>69</td>
<td>PUSH</td>
</tr>
<tr>
<td>j</td>
<td>6a</td>
<td>PUSH</td>
</tr>
<tr>
<td>k</td>
<td>6b</td>
<td>PUSH</td>
</tr>
<tr>
<td>l</td>
<td>70</td>
<td>JO</td>
</tr>
<tr>
<td>m</td>
<td>71</td>
<td>JNO</td>
</tr>
<tr>
<td>n</td>
<td>72</td>
<td>JB</td>
</tr>
<tr>
<td>o</td>
<td>73</td>
<td>JAE</td>
</tr>
</tbody>
</table>
### 1.7 npad

It is an assembly language macro for aligning labels on a specific boundary.

That’s often needed for the busy labels to where the control flow is often passed, e.g., loop body starts. So the CPU can load the data or code from the memory effectively, through the memory bus, cache lines, etc.

Taken from listing.inc (MSVC):

By the way, it is a curious example of the different NOP variations. All these instructions have no effects whatsoever, but have a different size.

Having a single idle instruction instead of couple of NOP-s, is accepted to be better for CPU performance.
.2 ARM

.2.1 Terminology

ARM was initially developed as 32-bit CPU, so that’s why a word here, unlike x86, is 32-bit.

byte 8-bit. The DB assembly directive is used for defining variables and arrays of bytes.

halfword 16-bit. DCW assembly directive —“—.

word 32-bit. DCD assembly directive —“—.

doubleword 64-bit.
quadword 128-bit.

.2.2 Versions

- ARMv4: Thumb mode introduced.
- ARMv6: used in iPhone 1st gen., iPhone 3G (Samsung 32-bit RISC ARM 1176JZ(F)-S that supports Thumb-2)
- ARMv7: Thumb-2 was added (2003). was used in iPhone 3GS, iPhone 4, iPad 1st gen. (ARM Cortex-A8), iPad 2 (Cortex-A9), iPad 3rd gen.
- ARMv7s: New instructions added. iPhone 5, iPhone 5c, iPad 4th gen. (Apple A6).
- ARMv8: 64-bit CPU, AKA ARM64 AKA AArch64. Was used in iPhone 5S, iPad Air (Apple A7). There is no Thumb mode in 64-bit mode, only ARM (4-byte instructions).

.2.3 32-bit ARM (AArch32)

General purpose registers

- R0 — function result is usually returned using R0
- R1...R12 — GPRs
- R13 — AKA SP (stack pointer)
- R14 — AKA LR (link register)
- R15 — AKA PC (program counter)

R0-R3 are also called “scratch registers”: the function’s arguments are usually passed in them, and the values in them are not required to be restored upon the function’s exit.

Current Program Status Register (CPSR)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..4</td>
<td>M — processor mode</td>
</tr>
<tr>
<td>5</td>
<td>T — Thumb state</td>
</tr>
<tr>
<td>6</td>
<td>F — FIQ disable</td>
</tr>
<tr>
<td>7</td>
<td>I — IRQ disable</td>
</tr>
<tr>
<td>8</td>
<td>A — imprecise data abort disable</td>
</tr>
<tr>
<td>9</td>
<td>E — dataendianness</td>
</tr>
<tr>
<td>10..15</td>
<td>IT — if-then state</td>
</tr>
<tr>
<td>16..19</td>
<td>GE — greater-than-or-equal-to</td>
</tr>
<tr>
<td>20..23</td>
<td>DNM — do not modify</td>
</tr>
<tr>
<td>24</td>
<td>J — Java state</td>
</tr>
<tr>
<td>27</td>
<td>Q — sticky overflow</td>
</tr>
<tr>
<td>28</td>
<td>V — overflow</td>
</tr>
<tr>
<td>29</td>
<td>C — carry/borrow/extend</td>
</tr>
<tr>
<td>30</td>
<td>Z — zero bit</td>
</tr>
<tr>
<td>31</td>
<td>N — negative/less than</td>
</tr>
</tbody>
</table>

VFP (floating point) and NEON registers

<table>
<thead>
<tr>
<th>0..31</th>
<th>32..64</th>
<th>65..96</th>
<th>97..127</th>
</tr>
</thead>
<tbody>
<tr>
<td>0128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>064</td>
<td>D1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>032</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
</tr>
<tr>
<td>016</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

S-registers are 32-bit, used for the storage of single precision numbers.
D-registers are 64-bit ones, used for the storage of double precision numbers.
D- and S-registers share the same physical space in the CPU—it is possible to access a D-register via the S-registers (it is senseless though).

Likewise, the NEON Q-registers are 128-bit ones and share the same physical space in the CPU with the other floating point registers.
In VFP 32 S-registers are present: S0..S31.
In VFPv2 there 16 D-registers are added, which in fact occupy the same space as S0..S31.
In VFPv3 (NEON or “Advanced SIMD”) there are 16 more D-registers, D0..D31, but the D16..D31 registers are not sharing space with any other S-registers.
In NEON or “Advanced SIMD” another 16 128-bit Q-registers were added, which share the same space as D0..D31.

.2.4 64-bit ARM (AArch64)

General purpose registers
The number of registers has been doubled since AArch32.

- X0 — function result is usually returned using X0
- X0...X7 — Function arguments are passed here.
- X8
- X9..X15 — are temporary registers, the callee function can use and not restore them.
- X16
- X17
- X18
- X19...X29 — callee function can use them, but must restore them upon exit.
- X29 — used as FP (at least GCC)
- X30 — “Procedure Link Register” AKA LR (link register).
- X31—register always contains zero AKA XZR or “Zero Register”. It’s 32-bit part is called WZR.
- SP, not a general purpose register anymore.

See also: [Procedure Call Standard for the ARM 64-bit Architecture (AArch64), (2013)]

The 32-bit part of each X-register is also accessible via W-registers (W0, W1, etc.).

<table>
<thead>
<tr>
<th>High 32-bit part</th>
<th>low 32-bit part</th>
</tr>
</thead>
<tbody>
<tr>
<td>X0</td>
<td>W0</td>
</tr>
</tbody>
</table>

.2.5 Instructions
There is a -S suffix for some instructions in ARM, indicating that the instruction sets the flags according to the result. Instructions which lacks this suffix are not modify flags. For example ADD unlike ADDS will add two numbers, but the flags will not be touched. Such instructions are convenient to use between CMP where the flags are set and, e.g. conditional jumps, where the flags are used. They are also better in terms of data dependency analysis (because less number of registers are modified during execution).

12Also available as http://go.yurichev.com/17287
### Conditional codes table

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Equal</td>
<td>$Z == 1$</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
<td>$Z == 0$</td>
</tr>
<tr>
<td><strong>CS AKA HS</strong> (Higher or Same)</td>
<td>Carry set / Unsigned, Greater than, equal</td>
<td>$C == 1$</td>
</tr>
<tr>
<td><strong>CC AKA LO</strong> (LOwer)</td>
<td>Carry clear / Unsigned, Less than</td>
<td>$C == 0$</td>
</tr>
<tr>
<td>MI</td>
<td>Minus, negative / Less than</td>
<td>$N == 1$</td>
</tr>
<tr>
<td>PL</td>
<td>Plus, positive or zero / Greater than, equal</td>
<td>$N == 0$</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow</td>
<td>$V == 1$</td>
</tr>
<tr>
<td>VC</td>
<td>No overflow</td>
<td>$V == 0$</td>
</tr>
<tr>
<td>HI</td>
<td>Unsigned higher / Greater than</td>
<td>$C == 1$ and $Z == 0$</td>
</tr>
<tr>
<td>LS</td>
<td>Unsigned lower or same / Less than or equal</td>
<td>$C == 0$ or $Z == 1$</td>
</tr>
<tr>
<td>GE</td>
<td>Signed greater than or equal / Greater than or equal</td>
<td>$N == V$</td>
</tr>
<tr>
<td>LT</td>
<td>Signed less than / Less than</td>
<td>$N ! = V$</td>
</tr>
<tr>
<td>GT</td>
<td>Signed greater than / Greater than</td>
<td>$Z == 0$ and $N == V$</td>
</tr>
<tr>
<td>LE</td>
<td>Signed less than or equal / Less than, equal</td>
<td>$Z == 1$ or $N != V$</td>
</tr>
<tr>
<td>None / AL</td>
<td>Always</td>
<td>Any</td>
</tr>
</tbody>
</table>

### .3 MIPS

#### .3.1 Registers

( O32 calling convention )

**General purpose registers GPR**

<table>
<thead>
<tr>
<th>Number</th>
<th>Pseudoname</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$ZERO</td>
<td>Always zero. Writing to this register is like NOP.</td>
</tr>
<tr>
<td>$1</td>
<td>$AT</td>
<td>Used as a temporary register for assembly macros and pseudo instructions.</td>
</tr>
<tr>
<td>$2 ...$3</td>
<td>$V0 ...$V1</td>
<td>Function result is returned here.</td>
</tr>
<tr>
<td>$4 ...$7</td>
<td>$A0 ...$A3</td>
<td>Function arguments.</td>
</tr>
<tr>
<td>$8 ...$15</td>
<td>$T0 ...$T7</td>
<td>Used for temporary data.</td>
</tr>
<tr>
<td>$16 ...$23</td>
<td>$S0 ...$S7</td>
<td>Used for temporary data*.</td>
</tr>
<tr>
<td>$24 ...$25</td>
<td>$T8 ...$T9</td>
<td>Used for temporary data.</td>
</tr>
<tr>
<td>$26 ...$27</td>
<td>$K0 ...$K1</td>
<td>Reserved for OS kernel.</td>
</tr>
<tr>
<td>$28</td>
<td>$GP</td>
<td>Global Pointer**.</td>
</tr>
<tr>
<td>$29</td>
<td>$SP</td>
<td>$SP*.</td>
</tr>
<tr>
<td>$30</td>
<td>$FP</td>
<td>$FP*.</td>
</tr>
<tr>
<td>$31</td>
<td>$RA</td>
<td>$RA.</td>
</tr>
<tr>
<td>n/a</td>
<td>PC</td>
<td>$PC.</td>
</tr>
<tr>
<td>n/a</td>
<td>HI</td>
<td>$HI.</td>
</tr>
<tr>
<td>n/a</td>
<td>LO</td>
<td>$LO.</td>
</tr>
</tbody>
</table>

**Floating-point registers**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F0 ..$F1</td>
<td>Function result returned here.</td>
</tr>
<tr>
<td>$F2 ..$F3</td>
<td>Not used.</td>
</tr>
<tr>
<td>$F4 ..$F11</td>
<td>Used for temporary data.</td>
</tr>
<tr>
<td>$F12 ..$F15</td>
<td>First two function arguments.</td>
</tr>
<tr>
<td>$F16 ..$F19</td>
<td>Used for temporary data.</td>
</tr>
<tr>
<td>$F20 ..$F31</td>
<td>Used for temporary data*.</td>
</tr>
</tbody>
</table>

* — **Callee** must preserve the value.

** — **Callee** must preserve the value ( except in PIC code).

*** — accessible using the MFHI and MFL0 instructions.
### 3.2 Instructions

There are 3 kinds of instructions:

- **R-type**: those which have 3 registers. R-instruction usually have the following form:

  \[ \text{instruction destination, source1, source2} \]

  One important thing to keep in mind is that when the first and second register are the same, IDA may show the instruction in its shorter form:

  \[ \text{instruction destination/source1, source2} \]

  That somewhat reminds us of the Intel syntax for x86 assembly language.

- **I-type**: those which have 2 registers and a 16-bit immediate value.

- **J-type**: jump/branch instructions, have 26 bits for encoding the offset.

### Jump instructions

What is the difference between B- instructions (BEQ, B, etc.) and J- ones (JAL, JALR, etc.)?

The B-instructions have an I-type, hence, the B-instructions’ offset is encoded as a 16-bit immediate. JR and JALR are R-type and jump to an absolute address specified in a register. J and JAL are J-type, hence the offset is encoded as a 26-bit immediate.

In short, B-instructions can encode a condition (B is in fact pseudo instruction for \( \text{BEQ } \$0, \$0, \text{LABEL} \)), while J-instructions can’t.

### 4 Some GCC library functions

<table>
<thead>
<tr>
<th>name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>__divdi3</td>
<td>signed division</td>
</tr>
<tr>
<td>__moddi3</td>
<td>getting remainder (modulo) of signed division</td>
</tr>
<tr>
<td>__udivdi3</td>
<td>unsigned division</td>
</tr>
<tr>
<td>__umoddi3</td>
<td>getting remainder (modulo) of unsigned division</td>
</tr>
</tbody>
</table>

### 5 Some MSVC library functions

ll in function name stands for “long long”, e.g., a 64-bit data type.

<table>
<thead>
<tr>
<th>name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_alldiv</td>
<td>signed division</td>
</tr>
<tr>
<td>_allmul</td>
<td>multiplication</td>
</tr>
<tr>
<td>_allrem</td>
<td>remainder of signed division</td>
</tr>
<tr>
<td>_allshl</td>
<td>shift left</td>
</tr>
<tr>
<td>_allshr</td>
<td>signed shift right</td>
</tr>
<tr>
<td>_aulldiv</td>
<td>unsigned division</td>
</tr>
<tr>
<td>_aullrem</td>
<td>remainder of unsigned division</td>
</tr>
<tr>
<td>_aullshr</td>
<td>unsigned shift right</td>
</tr>
</tbody>
</table>

Multiplication and shift left procedures are the same for both signed and unsigned numbers, hence there is only one function for each operation here.

The source code of these function can be found in the installed MSVS, in VC/crt/src/intel/*.asm.

### 6 Cheatsheets

#### 6.1 IDA

Hot-keys cheatsheet:
<table>
<thead>
<tr>
<th>key</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space</td>
<td>switch listing and graph view</td>
</tr>
<tr>
<td>C</td>
<td>convert to code</td>
</tr>
<tr>
<td>D</td>
<td>convert to data</td>
</tr>
<tr>
<td>A</td>
<td>convert to string</td>
</tr>
<tr>
<td>*</td>
<td>convert to array</td>
</tr>
<tr>
<td>U</td>
<td>undefine</td>
</tr>
<tr>
<td>O</td>
<td>make offset of operand</td>
</tr>
<tr>
<td>H</td>
<td>make decimal number</td>
</tr>
<tr>
<td>R</td>
<td>make char</td>
</tr>
<tr>
<td>B</td>
<td>make binary number</td>
</tr>
<tr>
<td>Q</td>
<td>make hexadecimal number</td>
</tr>
<tr>
<td>N</td>
<td>rename identifier</td>
</tr>
<tr>
<td>?</td>
<td>calculator</td>
</tr>
<tr>
<td>G</td>
<td>jump to address</td>
</tr>
<tr>
<td>:</td>
<td>add comment</td>
</tr>
<tr>
<td>Ctrl-X</td>
<td>show references to the current function, label, variable (incl. in local stack)</td>
</tr>
<tr>
<td>X</td>
<td>show references to the function, label, variable, etc.</td>
</tr>
<tr>
<td>Alt-I</td>
<td>search for constant</td>
</tr>
<tr>
<td>Ctrl-I</td>
<td>search for the next occurrence of constant</td>
</tr>
<tr>
<td>Alt-B</td>
<td>search for byte sequence</td>
</tr>
<tr>
<td>Ctrl-B</td>
<td>search for the next occurrence of byte sequence</td>
</tr>
<tr>
<td>Alt-T</td>
<td>search for text (including instructions, etc.)</td>
</tr>
<tr>
<td>Ctrl-T</td>
<td>search for the next occurrence of text</td>
</tr>
<tr>
<td>Alt-P</td>
<td>edit current function</td>
</tr>
<tr>
<td>Enter</td>
<td>jump to function, variable, etc.</td>
</tr>
<tr>
<td>Esc</td>
<td>get back</td>
</tr>
<tr>
<td>Num -</td>
<td>fold function or selected area</td>
</tr>
<tr>
<td>Num +</td>
<td>unhide function or area</td>
</tr>
</tbody>
</table>

Function/area folding may be useful for hiding function parts when you realize what they do. This is used in myscript\textsuperscript{13} for hiding some often used patterns of inline code.

\subsection{OllyDbg}

Hot-keys cheatsheet:

<table>
<thead>
<tr>
<th>hot-key</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>F7</td>
<td>trace into</td>
</tr>
<tr>
<td>F8</td>
<td>step over</td>
</tr>
<tr>
<td>F9</td>
<td>run</td>
</tr>
<tr>
<td>Ctrl-F2</td>
<td>restart</td>
</tr>
</tbody>
</table>

\subsection{MSVC}

Some useful options which were used through this book.

<table>
<thead>
<tr>
<th>option</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/O1</td>
<td>minimize space</td>
</tr>
<tr>
<td>/Ob0</td>
<td>no inline expansion</td>
</tr>
<tr>
<td>/Ox</td>
<td>maximum optimizations</td>
</tr>
<tr>
<td>/GS-</td>
<td>disable security checks (buffer overflows)</td>
</tr>
<tr>
<td>/Fa(file)</td>
<td>generate assembly listing</td>
</tr>
<tr>
<td>/Zi</td>
<td>enable debugging information</td>
</tr>
<tr>
<td>/Zp(n)</td>
<td>pack structs on ( n )-byte boundary</td>
</tr>
<tr>
<td>/MD</td>
<td>produced executable will use MSVCR*.DLL</td>
</tr>
</tbody>
</table>

Some information about MSVC versions: \textbf{5.1.1 on page 692}.

\subsection{GCC}

Some useful options which were used through this book.

\textsuperscript{13}GitHub
<table>
<thead>
<tr>
<th>option</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Os</td>
<td>code size optimization</td>
</tr>
<tr>
<td>-O3</td>
<td>maximum optimization</td>
</tr>
<tr>
<td>-regparm=</td>
<td>how many arguments are to be passed in registers</td>
</tr>
<tr>
<td>-o file</td>
<td>set name of output file</td>
</tr>
<tr>
<td>-g</td>
<td>produce debugging information in resulting executable</td>
</tr>
<tr>
<td>-S</td>
<td>generate assembly listing file</td>
</tr>
<tr>
<td>-masm=intel</td>
<td>produce listing in Intel syntax</td>
</tr>
<tr>
<td>-fno-inline</td>
<td>do not inline functions</td>
</tr>
</tbody>
</table>

### 6.5 GDB

Some of commands we used in this book:

<table>
<thead>
<tr>
<th>option</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>break filename.c:number</td>
<td>set a breakpoint on line number in source code</td>
</tr>
<tr>
<td>break function</td>
<td>set a breakpoint on function</td>
</tr>
<tr>
<td>break *address</td>
<td>set a breakpoint on address</td>
</tr>
<tr>
<td>b</td>
<td>run</td>
</tr>
<tr>
<td>p variable</td>
<td>print value of variable</td>
</tr>
<tr>
<td>run</td>
<td>continue execution</td>
</tr>
<tr>
<td>c</td>
<td>print stack</td>
</tr>
<tr>
<td>bt</td>
<td>set Intel syntax</td>
</tr>
<tr>
<td>disas function</td>
<td>disassemble current function</td>
</tr>
<tr>
<td>disas function,+50</td>
<td>disassemble function</td>
</tr>
<tr>
<td>disas $eip,+0x10</td>
<td>disassemble portion</td>
</tr>
<tr>
<td>disas/r</td>
<td>disassemble with opcodes</td>
</tr>
<tr>
<td>info registers</td>
<td>print all registers</td>
</tr>
<tr>
<td>info float</td>
<td>print FPU-registers</td>
</tr>
<tr>
<td>info locals</td>
<td>dump local variables (if known)</td>
</tr>
<tr>
<td>x/w ...</td>
<td>dump memory as 32-bit word</td>
</tr>
<tr>
<td>x/w $rdi</td>
<td>dump memory as 32-bit word</td>
</tr>
<tr>
<td>x/10w ...</td>
<td>dump 10 memory words</td>
</tr>
<tr>
<td>x/s ...</td>
<td>dump memory as string</td>
</tr>
<tr>
<td>x/i ...</td>
<td>dump memory as code</td>
</tr>
<tr>
<td>x/10c ...</td>
<td>dump 10 characters</td>
</tr>
<tr>
<td>x/b ...</td>
<td>dump bytes</td>
</tr>
<tr>
<td>x/h ...</td>
<td>dump 16-bit halfwords</td>
</tr>
<tr>
<td>x/g ...</td>
<td>dump giant (64-bit) words</td>
</tr>
<tr>
<td>finish</td>
<td>execute till the end of function</td>
</tr>
<tr>
<td>next</td>
<td>next instruction (don't dive into functions)</td>
</tr>
<tr>
<td>step</td>
<td>next instruction (dive into functions)</td>
</tr>
<tr>
<td>set step-mode on</td>
<td>do not use line number information while stepping</td>
</tr>
<tr>
<td>frame n</td>
<td>switch stack frame</td>
</tr>
<tr>
<td>info break</td>
<td>list of breakpoints</td>
</tr>
<tr>
<td>del n</td>
<td>delete breakpoint</td>
</tr>
<tr>
<td>set args ...</td>
<td>set command-line arguments</td>
</tr>
</tbody>
</table>
Acronyms Used
OS  Operating System........................................................................................................xv

OOP  Object-Oriented Programming............................................................................542

PL  Programming Language............................................................................................xiii

PRNG  Pseudorandom Number Generator...................................................................viii

ROM  Read-Only Memory...............................................................................................81

ALU  Arithmetic Logic Unit..........................................................................................25

PID  Program/process ID............................................................................................799

LF  Line Feed (10 or 'n' in C/C++)............................................................................524

CR  Carriage Return (13 or 'r' in C/C++) ....................................................................524

LIFO  Last In First Out...............................................................................................29

MSB  Most Significant Bit.........................................................................................315

LSB  Least Significant Bit

NSA  National Security Agency...............................................................................460

CFB  Cipher Feedback...............................................................................................845

CSPRNG  Cryptographically Secure Pseudorandom Number Generator................846

ABI  Application Binary Interface..............................................................................15

RA  Return Address....................................................................................................21

PE  Portable Executable.............................................................................................5

SP  stack pointer. SP/ESP/RSP in x86/x64. SP in ARM............................................18

DLL  Dynamic-Link Library........................................................................................748

PC  Program Counter. IP/EIP/RIP in x86/64. PC in ARM..........................................19

LR  Link Register..........................................................................................................6

IDA  Interactive Disassembler and Debugger developed by Hex-Rays......................6

IAT  Import Address Table.........................................................................................748

INT  Import Name Table..............................................................................................748

RVA  Relative Virtual Address....................................................................................748
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VA</td>
<td>Virtual Address</td>
<td>748</td>
</tr>
<tr>
<td>OEP</td>
<td>Original Entry Point</td>
<td>738</td>
</tr>
<tr>
<td>MSVC</td>
<td>Microsoft Visual C++</td>
<td></td>
</tr>
<tr>
<td>MSVS</td>
<td>Microsoft Visual Studio</td>
<td>1020</td>
</tr>
<tr>
<td>ASLR</td>
<td>Address Space Layout Randomization</td>
<td>607</td>
</tr>
<tr>
<td>MFC</td>
<td>Microsoft Foundation Classes</td>
<td>752</td>
</tr>
<tr>
<td>TLS</td>
<td>Thread Local Storage</td>
<td>280</td>
</tr>
<tr>
<td>AKA</td>
<td>Also Known As</td>
<td>29</td>
</tr>
<tr>
<td>CRT</td>
<td>C Runtime library</td>
<td>10</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td>xv</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
<td>855</td>
</tr>
<tr>
<td>FPU</td>
<td>Floating-Point Unit</td>
<td>v</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computing</td>
<td>19</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
<td>2</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
<td>744</td>
</tr>
<tr>
<td>RTTI</td>
<td>Run-Time Type Information</td>
<td>557</td>
</tr>
<tr>
<td>BSS</td>
<td>Block Started by Symbol</td>
<td>24</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data</td>
<td>194</td>
</tr>
<tr>
<td>BSOD</td>
<td>Blue Screen of Death</td>
<td>739</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management Systems</td>
<td>xiii</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
<td>ix</td>
</tr>
<tr>
<td>HPC</td>
<td>High-Performance Computing</td>
<td>517</td>
</tr>
<tr>
<td>SEH</td>
<td>Structured Exception Handling</td>
<td>36</td>
</tr>
<tr>
<td>ELF</td>
<td>Executable File format widely used in *NIX systems including Linux</td>
<td>79</td>
</tr>
<tr>
<td>TIB</td>
<td>Thread Information Block</td>
<td>280</td>
</tr>
<tr>
<td>PIC</td>
<td>Position Independent Code</td>
<td>537</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td></td>
</tr>
<tr>
<td>NAN</td>
<td>Not a Number</td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
<td></td>
</tr>
<tr>
<td>BEQ</td>
<td>(PowerPC, ARM) Branch if Equal</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>(PowerPC, ARM) Branch if Not Equal</td>
<td></td>
</tr>
<tr>
<td>BLR</td>
<td>(PowerPC) Branch to Link Register</td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>eXclusive OR</td>
<td></td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller Unit</td>
<td></td>
</tr>
<tr>
<td>RAM</td>
<td>Random-Access Memory</td>
<td></td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
<td></td>
</tr>
<tr>
<td>EGA</td>
<td>Enhanced Graphics Adapter</td>
<td></td>
</tr>
<tr>
<td>VGA</td>
<td>Video Graphics Array</td>
<td></td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
<td></td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
<td></td>
</tr>
<tr>
<td>ASCIIZ</td>
<td>ASCII Zero (null-terminated ASCII string)</td>
<td></td>
</tr>
<tr>
<td>IA64</td>
<td>Intel Architecture 64 (Itanium)</td>
<td></td>
</tr>
<tr>
<td>EPIC</td>
<td>Explicitly Parallel Instruction Computing</td>
<td></td>
</tr>
<tr>
<td>OOE</td>
<td>Out-of-Order Execution</td>
<td></td>
</tr>
<tr>
<td>MSDN</td>
<td>Microsoft Developer Network</td>
<td></td>
</tr>
<tr>
<td>STL</td>
<td>(C++) Standard Template Library</td>
<td></td>
</tr>
<tr>
<td>PODT</td>
<td>(C++) Plain Old Data Type</td>
<td></td>
</tr>
<tr>
<td>HDD</td>
<td>Hard Disk Drive</td>
<td></td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Memory</td>
<td></td>
</tr>
<tr>
<td>WRK</td>
<td>Windows Research Kernel</td>
<td></td>
</tr>
<tr>
<td>GPR</td>
<td>General Purpose Registers</td>
<td></td>
</tr>
<tr>
<td>SSDT</td>
<td>System Service Dispatch Table</td>
<td></td>
</tr>
<tr>
<td>RE</td>
<td>Reverse Engineering</td>
<td></td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>EOF</td>
<td>End of File</td>
<td>85</td>
</tr>
<tr>
<td>DIY</td>
<td>Do It Yourself</td>
<td>614</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
<td>605</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
<td>444</td>
</tr>
<tr>
<td>MIME</td>
<td>Multipurpose Internet Mail Extensions</td>
<td>444</td>
</tr>
<tr>
<td>DBI</td>
<td>Dynamic Binary Instrumentation</td>
<td>523</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
<td>623</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
<td>623</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
<td>4</td>
</tr>
<tr>
<td>IV</td>
<td>Initialization Vector</td>
<td>x</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest Shamir Adleman</td>
<td>939</td>
</tr>
<tr>
<td>CPRNG</td>
<td>Cryptographically secure PseudoRandom Number Generator</td>
<td>940</td>
</tr>
<tr>
<td>GiB</td>
<td>Gibibyte</td>
<td>954</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic redundancy check</td>
<td>974</td>
</tr>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
<td>974</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collector</td>
<td>612</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated development environment</td>
<td>373</td>
</tr>
<tr>
<td>BB</td>
<td>Basic Block</td>
<td>996</td>
</tr>
</tbody>
</table>
Glossary

anti-pattern Generally considered as bad practice. 31, 76, 461

arithmetic mean a sum of all values divided by their count. 518

atomic operation “ατοµoς” stands for “indivisible” in Greek, so an atomic operation is guaranteed not to be interrupted by other threads. 633, 778

basic block a group of instructions that do not have jump/branch instructions, and also don’t have jumps inside the block from the outside. In IDA it looks just like as a list of instructions without empty lines. 683, 987, 988

callee A function being called by another. 32, 46, 66, 86, 97, 99, 101, 418, 462, 544, 644, 725-728, 730, 731, 1028

caller A function calling another. 6–8, 10, 26, 46, 86, 97, 98, 100, 108, 155, 418, 471, 544, 725, 727, 728, 731

compiler intrinsic A function specific to a compiler which is not an usual library function. The compiler generates a specific machine code instead of a call to it. Often, it’s a pseudofunction for a specific CPU instruction. Read more: (11.3 on page 981). 1019

CP/M Control Program for Microcomputers: a very basic disk OS used before MS-DOS. 888

decrement Decrease by 1. 18, 183, 184, 202, 437, 718, 1013, 1015, 1020

dongle Dongle is a small piece of hardware connected to LPT printer port (in past) or to USB. 818

endianness Byte order. 21, 78, 343, 1017

GiB Gibibyte: $2^{30}$ or 1024 mebibytes or 1073741824 bytes. 15

heap usually, a big chunk of memory provided by the OS so that applications can divide it by themselves as they wish. malloc()/free() work with the heap. 30, 344, 560, 562, 563, 575, 577, 592, 593, 623, 747, 748

increment Increase by 1. 16, 19, 184, 187, 202, 208, 324, 327, 437, 1013

integral data type usual numbers, but not a real ones. may be used for passing variables of boolean data type and enumerations. 230

jump offset a part of the JMP or Jcc instruction’s opcode, to be added to the address of the next instruction, and this is how the new PC is calculated. May be negative as well. 93, 132, 1013

kernel mode A restrictions-free CPU mode in which the OS kernel and drivers execute. cf. user mode. 1039

leaf function A function which does not call any other function. 27, 31

link register (RISC) A register where the return address is usually stored. This makes it possible to call leaf functions without using the stack, i.e., faster. 31, 819, 1026, 1027

loop unwinding It is when a compiler, instead of generating loop code for $n$ iterations, generates just $n$ copies of the loop body, in order to get rid of the instructions for loop maintenance. 186
name mangling used at least in C++, where the compiler needs to encode the name of class, method and argument types in one string, which will become the internal name of the function. You can read more about it here: 3.21.1 on page 542. 542, 693, 694

NaN not a number: a special cases for floating point numbers, usually signaling about errors. 232, 254, 986

NEON AKA “Advanced SIMD”—SIMD from ARM. 1026
Nop “no operation”, idle instruction. 718
NTAPI API available only in the Windows NT line. Largely not documented by Microsoft. 786

padding Padding in English language means to stuff a pillow with something to give it a desired (bigger) form. In computer science, padding means to add more bytes to a block so it will have desired size, like \(2^n\) bytes.. 701

PDB (Win32) Debugging information file, usually just function names, but sometimes also function arguments and local variables names. 692, 750, 786, 787, 794, 795, 871
POKE BASIC language instruction for writing a byte at a specific address. 718

Product Multiplication result. 97, 223, 226, 405, 430, 450

Quotient Division result. 216, 219, 221, 222, 226, 429, 497, 519

Real number numbers which may contain a dot. this is float and double in C/C++. 216

Register allocator The part of the compiler that assigns CPU registers to local variables. 201, 305, 418

Reverse engineering act of understanding how the thing works, sometimes in order to clone it. iv, 1019

Security cookie A random value, different at each execution. You can read more about it here: 1.26.3 on page 279. 769

Stack frame A part of the stack that contains information specific to the current function: local variables, function arguments, RA, etc.. 67, 97, 98, 478, 769

Stack pointer A register pointing to a place in the stack. 9, 11, 19, 29, 34, 42, 54, 55, 73, 99, 544, 644, 725–728, 1007, 1014, 1026, 1033

Stdout standard output. 20, 34, 155

tail call It is when the compiler (or interpreter) transforms the recursion (tail recursion) into an iteration for efficiency. 481

Thunk function Tiny function with a single role: call another function. 22, 41, 389, 819, 828

Tracer My own simple debugging tool. You can read more about it here: 7.2.1 on page 781. 188–190, 541, 610, 696, 706, 709, 765, 773, 814, 815, 873, 879, 883, 885, 902, 980

User mode A restricted CPU mode in which it all application software code is executed. cf. kernel mode. 835, 1038

Windows NT Windows NT, 2000, XP, Vista, 7, 8, 10. 289, 416, 643, 699, 739, 749, 777, 890, 1019

Word data type fitting in GPR. In the computers older than PCs, the memory size was often measured in words rather than bytes.. 443–446, 451, 566, 624

Xoring often used in the English language, which implying applying the XOR operation. 769, 831, 834
C language elements
C99, 108
  bool,  303
  restrict, 514
  variable length arrays, 282
Comma, 992
const, 9, 81, 468
for, 183, 483
if, 123, 154
Pointers, 66, 73, 109, 381, 418, 596
Post-decrement, 437
Post-increment, 437
Pre-decrement, 437
Pre-increment, 437
ptrdiff_t, 613
return, 10, 86, 107
Short-circuit, 525, 527, 993
switch, 153, 154, 163
while, 200

C standard library
alloca(), 34, 282, 462, 760
assert(), 288, 703
atexit(), 565
atoi(), 500, 862
close(), 742
exit(), 471
fread(), 620
free(), 462, 592
fwrite(), 620
getenv(), 863
localtime(), 652
localtime_r(), 352
longjmp, 625
longjmp(), 155
malloc(), 345, 462, 592
memchr(), 1016
memcmp(), 624
memcpy(), 12, 66, 511, 624, 1015
memmove(), 624
memset(), 264, 510, 883, 1016
open(), 742
pow(), 228
puts(), 20
qsort(), 381, 470
rand(), 336, 695, 792, 794, 813, 839
realloc(), 462
realloc(), 462
scanf(), 66
setjmp, 625
srand(), 514
strcat(), 448, 471, 507, 742
strcmp(), 12, 509, 840
strcpy(), 200, 414, 509, 526, 1016
strstr(), 470
strtok(), 211
toupper(), 533
va_arg, 518
va_list, 522
vprintf, 522
write(), 620

C++, 874
C++11, 575, 734
exceptions, 760
ostream, 557
References, 558
RTTI, 557
STL, 692
   std::forward_list, 575
   std::list, 566
   std::map, 582
   std::set, 582
   std::string, 559
   std::vector, 575
C11, 734
Callbacks, 381
Canary, 279
cdecl, 42, 725
Chess, 459
Cipher Feedback mode, 845
clusterization, 938
COFF, 826
column-major order, 290
Compiler intrinsic, 35, 450, 981
Compiler’s anomalies, 146, 299, 313, 330, 493, 531, 981
Core dump, 606
Cray, 405, 446, 457, 460
CRC32, 462, 482
CRT, 744, 765
CryptoMiniSat, 425
CryptoPP, 724, 843
Cygwin, 693, 696, 754, 782
Data general Nova, 216
De Morgan’s laws, 994
DEC Alpha, 404
DES, 405, 418
dlopen(), 742
dlsym(), 742
dmalloc, 606
Donald E. Knuth, 446
DOSBox, 890
DosBox, 709
double, 217, 730
Doubly linked list, 458, 566
dtruss, 781
Duff’s device, 494
Dynamically loaded libraries, 21
Edsger W. Dijkstra, 593
EICAR, 887
ELF, 79
Entropy, 911, 929
Error messages, 702
fastcall, 14, 33, 65, 305, 726
fetchmail, 444
FidoNet, 701
FILETIME, 402
FIXUP, 816
float, 217, 730
Forth, 676
FORTRAN, 22
Fortran, 290, 514, 593, 693
FreeBSD, 705
Function epilogue, 28, 54, 56, 135, 361, 708
Function prologue, 10, 28, 31, 55, 279, 708
Fused multiply–add, 102, 103
Fuzzing, 506
Garbage collector, 612, 677
GCC, 693, 1029, 1030
GDB, 27, 47, 51, 278, 389, 390, 781, 1030
GeoIP, 930
GHex, 780
Glibc, 389, 624, 738
Global variables, 76
GNU Scientific Library, 356
GnuPG, 940
GraphViz, 611
grep usage, 190, 260, 692, 706, 709, 872
Hash functions, 462
HASP, 705
Heartbleed, 623, 850
Heisenbug, 630, 638
Hex-Rays, 107, 197, 296, 301, 613, 637, 988
Hiew, 92, 132, 153, 697, 702, 750, 751, 754, 780, 980
Honeywell 6070, 444
ICQ, 718
IDA, 86, 153, 376, 514, 686, 699, 780, 781, 962, 1029
var ?, 55, 73
IEEE 754, 217, 315, 374, 425, 1005
Inline code, 192, 312, 506, 548, 579
Integer overflow, 105
Intel
   8080, 208
   8086, 208, 312, 835
   Memory model, 650, 986
   8253, 889
   80286, 835, 987
   80386, 312, 987
   80486, 217
   FPU, 217
Intel 4004, 443
Intel C++, 9, 406, 981, 987, 1014
Intel syntax, 12, 18
iPod/iPhone/iPad, 18
Itanium, 404, 984
JAD, 5
Java, 445, 654
John Carmack, 524
JPEG, 938
jumpable, 167, 175
Keil, 18
kernel panic, 738
kernel space, 738
LAPACK, 22
LARGE_INTEGER, 402
LD_PRELOAD, 742
Linker, 81, 542
Linux, 306, 739, 874
   libc.so.6, 305, 389
LISP, 599
LLDB, 781
LLVM, 18
1042
long double, 217
Loop unwinding, 186
LZMA, 933

Mac OS Classic, 818
Mac OS X, 782
Mathematica, 593, 804
MD5, 462, 704
memfrob(), 843
Memoization, 805
MFC, 751, 863
Microsoft, 402
Microsoft Word, 623
MIDI, 704
MinGW, 693, 901
minifloat, 439
MIPS, 2, 712, 723, 749, 819, 937
Branch delay slot, 8
Global Pointer, 24, 296
Instructions
ADD, 105
ADDIU, 25, 84, 85
ADDU, 105
AND, 314
BC1F, 263
BC1T, 263
BEQ, 96, 137
BLTZ, 141
BNE, 137
BNEZ, 177
BREAK, 499
C.LT.D, 263
J, 6, 8, 25
JAL, 105
JALR, 25, 105
JR, 166
LB, 197
LBU, 197
LI, 441
LUI, 25, 84, 85, 317, 441
LW, 25, 74, 85, 166, 441
MFHI, 105, 500, 1028
MFLO, 105, 499, 1028
MTC1, 378
MULT, 105
NOR, 210
OR, 27
ORI, 314, 441
SB, 197
SLL, 177, 212, 333
SLLV, 333
SLT, 137
SLTUI, 177
SLTU, 137, 139, 177
SRL, 217
SUBU, 141
SW, 61
Load delay slot, 166
O32, 61, 65, 1027
Pseudoinstructions
B, 194
BEQZ, 139
LA, 27
LI, 8
MOVE, 25, 83
NEGU, 141
NOP, 27, 83
NOT, 210
Registers
FCCR, 263
HI, 499
LO, 499
MS-DOS, 14, 33, 280, 603, 647, 704, 709, 718, 748, 835, 887, 888, 942, 986, 1005, 1015, 1019, 1020
MSVC, 1029, 1030
MSVCRT.DLL, 901
Name mangling, 542
Native API, 749
Non-a-numbers (NaNs), 254
Notepad, 934
NSA, 460
objcdump, 376, 741, 754, 780
octet, 444
OEP, 748, 754
OOP
Polymorphism, 542
opaque predicate, 538
OpenMP, 630, 695
OpenSSL, 623, 850
OpenWatcom, 693, 727
Oracle RDBMS, 9, 405, 702, 757, 874, 882, 883, 954, 964, 981, 987
Page (memory), 416
Pascal, 697
PDP-11, 437
PGP, 701
Phrack, 701
Pin, 523
PNG, 936
position-independent code, 18, 739
PowerPC, 2, 24, 818
Propagating Cipher Block Chaining, 855
Punched card, 264
puts() instead of printf(), 20, 71, 106, 133
Python, 523, 593
ctypes, 733
Qt, 14
Quake, 524
Quake III Arena, 380
Racket, 992
rada.re, 13
Radare, 781
radare2, 939
rafind2, 780
RAID4, 457
RAM, 81
Raspberry Pi, 18
ReactOS, 763
Recursion, 29, 30, 481
FPU, 1009
Instructions
AAA, 1024
AAS, 1024
ADC, 394, 647, 1013
ADD, 9, 42, 97, 501, 647, 1013
ADDSD, 426
ADDSS, 436
ADDrcc, 143
AESDECE, 844
AESENC, 844
AESKEYGENASSIST, 846
AND, 10, 304, 307, 320, 334, 368, 1013, 1017
BSF, 417, 1017
BSR, 1017
BSWAP, 461, 1017
BT, 1017
BTC, 316, 1017
BTR, 316, 778, 1017
BTS, 316, 1017
CALL, 9, 30, 720, 753, 856, 929, 1013
CBW, 449, 1017
CDQ, 401, 449, 1017
CDQE, 449, 1017
CLD, 1017
CLI, 1017
CMC, 1017
CMOVcc, 135, 143, 144, 147, 150, 462, 1017
CMP, 86, 470, 1013, 1024
CMPSB, 704, 1017
CMPSD, 1017
CMPSQ, 1017
CMPSW, 1017
COMISD, 434
COMISS, 436
CPUID, 366, 1019
CWD, 449, 647, 899, 1017
CWDE, 449, 1017
DEC, 202, 1013, 1024
DIV, 449, 1019
DIVSD, 426, 707
FABS, 1021
FADD, 1021
FADDP, 219, 225, 1021
FATRET, 329, 330
FCHS, 1021
FCMOVcc, 256
FCOM, 243, 254, 1021
FCOMP, 231, 1021
FCOMPP, 1021
FDIV, 218, 706, 1021
FDIVP, 219, 1021
FDIVR, 225, 1021
FDIVRP, 1021
FDUP, 676
FILD, 1021
FIST, 1022
FISTP, 1022
FLD, 229, 231, 1022
FLD1, 1022
FLDCW, 1022
FLDZ, 1022
FMUL, 219, 1022
FMULP, 1022
FNSTCW, 1022
FNSTSW, 231, 254, 1022
FSCALE, 379
FSINCOS, 1022
FSQRT, 1022
FST, 1022
FSTCW, 1022
FSTP, 229, 1022
FSTSW, 1022
FSUB, 1022
FSUBP, 1022
FSUBR, 1022
FSUBRP, 1022
FUCOM, 254, 1022
FUCOMP, 256
FUCOMP, 1022
FUCOMPP, 254, 1022
FWAIT, 217
FXCH, 982, 1022
IDIV, 449, 497, 1019
IMUL, 97, 299, 449, 450, 599, 1013, 1024
IN, 720, 835, 889, 1019
INC, 202, 980, 1013, 1024
INT, 33, 888, 1019
INT3, 696
IRET, 1019, 1020
JA, 124, 255, 449, 1013, 1024
JAE, 124, 1013, 1024
JB, 124, 449, 1013, 1024
JBE, 124, 1013, 1024
JC, 1013
Jcc, 96, 146
JCXZ, 1013
JE, 155, 1013, 1024
JECXZ, 1013
JG, 124, 449, 1013
JGE, 124, 1013
JL, 124, 449, 1013
JLE, 124, 1013
JMP, 30, 41, 54, 753, 980, 1013
JNA, 1013
JNAE, 1013
JNB, 1013
JNBE, 255, 1013
JNC, 1013
JNE, 86, 124, 1013, 1024
JNG, 1013
JNGE, 1013
JNL, 1013
JNLE, 1013
JNO, 1013, 1024
JNS, 1013, 1024
JNZ, 1013
JO, 1013, 1024
JP, 232, 1013, 1024
JPO, 1013
JRCXZ, 1013
JS, 1013, 1024
JZ, 94, 155, 981, 1013
LAHF, 1014
LEA, 67, 99, 347, 472, 484, 501, 729, 790, 856, 1014
LEAVE, 11, 1014
LES, 840, 898
LOCK, 777
LODSB, 889
LOOP, 183, 199, 708, 898, 1020
MAXSD, 434
MOV, 8, 10, 12, 510, 511, 720, 751, 856, 929, 980, 1015
MOVSDA, 408
MOVQ, 408
MOVS, 1015
MOVS, 433, 512, 1015
 MOVSX, 433
 MOVSQ, 1015
 MOVSS, 436
 MOVSX, 1015
 MOVSX, 200, 208, 360-362, 449, 1015
 MOVSXD, 283
 MOVZX, 201, 345, 819, 1015
 MUL, 449, 450, 599, 1015
 MULSD, 426
 NEG, 504, 1015
 NOP, 484, 980, 1015, 1024
 NOT, 206, 208, 1015
 OR, 307, 526, 1015
 OUT, 720, 835, 1020
PADD, 408
PCMPEQ, 417
PLMULHW, 405
PLMULLD, 405
 PMOVMSKB, 417
 POP, 9, 29, 30, 1015, 1024
 POPA, 1020, 1024
 POPCNT, 1020
 POPF, 889, 1020
 PUSHA, 1020, 1024
 PUSHA, 1020, 1024
 PXOR, 417
 RCL, 708, 1020
 RCR, 1020
 RET, 6, 7, 10, 30, 279, 544, 644, 980, 1015
 ROL, 330, 981, 1020
 ROR, 981, 1020
 SAHF, 254, 1015
 SAL, 636, 1020
 SAR, 334, 449, 517, 636, 898, 1021
 SBB, 394, 1015
 SCASB, 890, 1016
 SCASD, 1016
 SCASQ, 1016
 SCASM, 1016
SET, 467
 SETcc, 137, 201, 255, 1021
 SHL, 212, 265, 334, 636, 1016
 SHR, 216, 334, 368, 636, 1016
 SHRD, 400, 1016
 STC, 1021
 STD, 1021
 STI, 1021
 STOSB, 496, 1016
 STOSD, 1016
 STOSQ, 510, 1016
 STOSW, 1016
 SUB, 10, 86, 155, 470, 501, 1013, 1017
SYSCALL, 1019, 1021
SYSENTER, 739, 1019, 1021
TEST, 200, 304, 307, 334, 1017
UD2, 1021
XADD, 778
XCHG, 1015, 1021
XOR, 10, 86, 206, 517, 707, 831, 980, 1017, 1024
MMX, 404
Prefixes
LOCK, 778, 1012
REP, 1012, 1015, 1016
REPE/REPNE, 1012
REPNE, 1016
Registers
AF, 444
AH, 1014, 1015
CS, 986
DF, 624
DR6, 1011
DR7, 1011
DS, 986
EAX, 86, 106
EBP, 67, 97
ECX, 542
ES, 898, 986
ESP, 42, 67
Flags, 86, 126, 1008
FS, 735
GS, 280, 735, 738
JMP, 173
RIP, 741
SS, 986
ZF, 86, 304
SSE, 405
SSE2, 405
x86-64, 14, 15, 50, 66, 72, 93, 99, 418, 425, 721, 727, 741, 1005, 1011
Xcode, 18
XML, 701, 840
XOR, 845
Z80, 444
zlib, 625, 843
Zobrist hashing, 459
ZX Spectrum, 454