

Parte I

Padrões de códigos

Quando o autor desse livro começou a aprender C e depois C++, ele costumava escrever pequenos pedaços de código, compilar e então o olhar na sua saída em assembly. Isso acabou tornando muito fácil para o seu entendimento sobre o que estava acontecendo no código que ele escreveu. Ele fez isso tantas vezes, que a relação entre o código em C/C++ e o que o compilador produzia ficou gravada em sua mente. É fácil imaginar a aparência e função de um rascunho em C. Algumas vezes essa técnica pode ser útil para outras pessoas.

Às vezes compiladores antigos serão usados aqui com o objetivo de conseguir o menor (ou mais simples) pedaço de código possível.

Níveis de otimização e informações de depuração

O código-fonte pode ser compilado por um número diferente de compiladores com vários níveis de otimização. Um compilador típico tem por volta de três desses níveis, onde o nível zero representa que a otimização está desabilitada. A otimização também pode ser relacionada com o tamanho ou velocidade do código. Um compilador não-otimizado é mais rápido e produz um código de mais fácil compreensão (embora detalhado), enquanto um compilador com otimização é mais lento e tenta produzir códigos que executam mais rápido (mas não necessariamente mais compactos). Em adição aos níveis de otimização e direção, um compilador pode incluir no arquivo de saída algumas informações de depuração, dessa maneira produzindo código para fácil depuração. Uma das características de um código “depurado” é que ele pode conter ligações entre cada linha do código-fonte e o endereço do respectivo código de máquina. Compiladores otimizadores, por outro lado, tendem a produzir saídas onde linhas completas do código-fonte podem ser otimizadas e apresentadas de uma maneira completamente diferente, muitas vezes ainda nem estando presente no código de máquina resultante. Com a engenharia reversa podemos obter quaisquer versões, simplesmente porque alguns desenvolvedores ativam as otimizações do compilador e outros não. Por causa disso, nós tentaremos trabalhar em ambos exemplos de depuração e versões de lançamento dos códigos demonstrados nesse livro, quando possível.

Capítulo 1

Uma breve introdução a CPU

A **CPU** é o dispositivo que executa o código de máquina do qual consiste um programa.

Um glossário resumido:

Instrução : Um comando primário da **CPU**. Os exemplos mais simples incluem: mover informação entre os registradores, trabalhar com memória, operações primárias de aritmética. Como regra, cada **CPU** tem sua própria arquitetura do conjunto de instruções (**ISA**¹).

Código de máquina : Código que a **CPU** processa diretamente, cada instrução geralmente é codificada por vários bytes.

Linguagem assembly : Códigos mnemônicos e algumas extensões como macros que tem por intenção facilitar a vida do programador.

Registrador da CPU : Cada **CPU** tem um conjunto fixo de registradores de propósito geral (**GPR**²). Aproximadamente 8 na arquitetura x86, 16 na x86-64, 16 na ARM. A maneira mais fácil de entender um registrador é imaginá-lo como uma variável temporário sem tipo. Imagine que você está trabalhando em uma linguagem de alto nível e pudesse usar somente oito variáveis de 32-bit (ou 64). Ainda assim uma gama de coisas podem ser feitas usando somente estes!

Você pode pensar por quê há a necessidade dessa diferença entre código de máquina e linguagens de programação de alto nível. A resposta está no fato de humanos e CPUs não se parecerem nada – é muito mais fácil para um humano usar uma linguagem de alto nível como C/C++, Java, Python, etc., mas para a CPU é mais fácil usar um nível muito mais baixo de abstração. Talvez seja possível inventar uma CPU que pode executar códigos em linguagem de alto nível, mas ela seria muitas vezes mais complexa que as CPUs que conhecemos hoje. De uma maneira similar, é muito mais inconveniente para humanos escreverem em linguagem assembly, devido ao fato dela ser tão baixo nível e difícil de se escrever sem cometer um alto número de erros irritantes. O programa que converte linguagem de alto nível em código assembly é chamado de compilador.

¹Instruction Set Architecture

²General Purpose Registers

Capítulo 2

A mais simples função

A função mais simples possível é indiscutivelmente aquela que simplesmente retorna um valor constante:

Aqui está:

Listing 2.1:

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

Vamos compilar!

2.1 x86

Aqui está o que ambos compiladores, GCC com otimização e MSVC produzem na plataforma x86:

Listing 2.2: Com otimização GCC/MSVC (saída do assembly)

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

Há somente duas instruções: a primeira coloca o valor 123 no registrador EAX, que é usado por convenção para guardar o valor de retorno e a segunda é a RET, que retorna a execução para onde a função foi chamada.

O resultado será obtido no registrador EAX.

Capítulo 3

Vamos usar o famoso exemplo do livro “The C programming Language”[\[Ker88\]](#):

```
#include <stdio.h>

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

3.1 x86

3.1.1 MSVC

Vamos compilar esse código no MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(A opção /Fa instrui o compilador para gerar o arquivo de listagem em assembly)

Listing 3.1: 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
```

O compilador gerou o arquivo 1.obj, que está ligado a 1.exe. No nosso caso, o arquivo contém dois segmentos: CONST (para informações que são constantes) e _TEXT (para o código).

A string hello, word em C/C++ tem seu tipo `const char []`[\[Str13, p176, 7.3.2\]](#), mas não tem um nome. O compilador precisa lidar com essa string de alguma maneira, definindo então o nome de \$SG3830 para ela.

Assim, o código pode ser reescrito da seguinte maneira:

```
#include <stdio.h>

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

int main()
```

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

Vamos voltar para a listagem em assembly. Como podemos ver, a string é delimitada por um byte de valor zero, o que é padrão para strings em C/C++. Mais sobre strings em C/C++: [25.1.1 on page 106](#).

No segmento de código `_TEXT`, só há uma função por enquanto: `main()`. A função `main()` começa com um código como cabeçalho e termina com outro como rodapé (quase como qualquer outra função)¹.

Depois do cabeçalho da função, podemos ver a chamada para a função `printf()`: `CALL _printf`. Antes da chamada, o endereço da string (ou um ponteiro para o mesmo) contendo nossa saudação (“Hello, world!”) é colocado na stack com a ajuda a instrução `PUSH`.

Quando o a função `printf()` retorna o controle para a função `main()`, o endereço da string (ou o ponteiro para a mesma) ainda está na stack. Como não precisamos mais dela, o apontador da stack (registrador `ESP`) precisa ser corrigido.

`ADD ESP, 4` significa adicionar 4 para o valor do registrador `ESP`.

Mas por que 4? Como esse é um programa de 32-bits, nós precisamos exatamente 4 bytes para endereço passando pela stack. `ADD ESP, 4` é equivalente a um `POP` mas sem precisar de nenhum registrador².

Pelos mesmos motivos, alguns compiladores (como o Intel C++ Compiler) podem emitir `POP ECX` ao invés de `ADD` (esse padrão pode ser observado no código do Oracle RDBMS pois ele é compilado com o Intel C++ Compiler). Essa instrução tem quase o mesmo efeito mas o conteúdo de `ECX` seria apagado. O Intel C++ provavelmente usa `POP ECX` pois o opcode é menor do que `ADD ESP, x` (1 byte para `POP` ao invés de 3 para `ADD`).

Aqui está um exemplo do uso de `POP` ao invés de `ADD` do Oracle RDBMS:

Listing 3.2: Oracle RDBMS 10.2 Linux (app.o file)

```
.text:0800029A      push    ebx
.text:0800029B      call   qksfroChild
.text:080002A0      pop     ecx
```

Depois de chamar `printf()`, o código original em C/C++ contém a declaração `return 0` – `return 0` como o resultado da função `main()`.

No código gerado, isso é implementado pela instrução `XOR EAX, EAX`.

`XOR` é a condição lógica “ou exclusivo”³ que os compiladores geralmente usam ao invés de `MOV EAX, 0` – de novo por causa de um pequeno decréscimo no número de bytes necessários (2 bytes para `XOR` contra 5 para a instrução `MOV`).

Alguns compiladores também usam `SUB EAX, EAX`, que significa, `SUB`trair o valor contido em `EAX` do valor em `EAX`, que, em qualquer caso, resultará em zero.

A última instrução `RET` retorna o controle para onde a função foi chamada. Geralmente, isso é código C/C++ `CRT`⁴, que retorna o controle para o sistema operacional.

3.2 x86-64

3.2.1 MSVC–x86-64

Vamos tentar também o MSVC 64-bits:

Listing 3.3: 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
```

¹Brazilian Portuguese text placeholder ([4 on page 8](#)).

²TBT: CPU flags worden echter wel aangepast

³[wikipedia](#)

⁴C runtime library

No x86-64, todos os registradores foram estendidos para 64-bits e agora seus nomes contêm um R- no prefixo. A fim de diminuir a frequência com que a stack (pilha) é usada (em outras palavras, para acessar memória externa/cache menos vezes), existe uma maneira popular de passar argumentos para funções através dos registradores (*fastcall*). Por exemplo, uma parte dos argumentos da função é passada nos registradores, o resto pela stack. No Win64, 4 argumentos de funções são passados através dos registradores RCX, RDX, R8, R9. Que é o que nós vemos, um ponteiro para a string para o printf() não é passado pela stack, mas no registrador RCX. Os ponteiros são 64-bits agora, então, eles são passados através dos registradores de 64-bits (que tem prefixo R-). Entretanto, para compatibilidade, ainda é possível acessar partes de 32-bits, usando o prefixo E-. É assim que os registradores RAX/EAX/AX/AL se parecem no x86-64:

Brazilian Portuguese text placeholder	
RAX ^{x64}	
EAX	
AX	
AH	AL

A função `main()` retorna um valor do tipo inteiro, que em C/C++ é melhor para compatibilidade com versões anteriores e portabilidade, de 32-bits, por isso o registrador EAX é limpo no final da função (a parte de 32-bits do registrador) ao invés de RAX. Há também 40 bytes alocados na pilha local. Que é chamado de “shadow space”, o qual falaremos mais tarde: [8.2.1 on page 29](#).

3.3 Conclusão

A principal diferença entre os códigos em x86/ARM e x64/ARM64 é que o ponteiro para a string é agora 64-bits de tamanho. De fato, CPUs modernas agora são de 64-bits devido a redução do custo da memória e a demanda mais alta devido a aplicações mais modernas. Nós podemos adicionar muito mais memória nos nossos computadores do que ponteiros de 32-bits são capazes de endereçar. Como tal, todos os ponteiros são agora 64-bits.

Capítulo 4

Cabeçalhos e rodapés de funções

Um cabeçalho de uma função é uma sequência de instruções no começo da função. Ele geralmente se parece com algo como o código a seguir:

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

O que essas instruções fazem: salvam o valor no registrador EBP, mudam o valor de EBP para o valor em ESP e então aloca espaço na pilha para variáveis locais.

O valor em EBP continua o mesmo depois do período de execução da função e é para ser usado para variáveis locais e acessos de argumentos. Para o mesmo propósito pode ser usado o ESP, mas considerando que ele muda com o tempo, essa abordagem não é muito conveniente.

O rodapé da função libera o espaço alocado na pilha, retorna o valor do registrador EBP de volta ao seu estado inicial e retorna o controle de volta para a chamada da função:

```
mov     esp, ebp
pop     ebp
ret     0
```

Os cabeçalhos e rodapés das funções geralmente são detectados na desassemblagem para a delimitação das funções.

Capítulo 5

Pilha

A pilha é uma das estruturas mais fundamentais na ciência da computação.¹

Tecnicamente, é só um bloco de memória junto com os registradores ESP ou RSP em x86 e x64, ou o SP² no ARM, como um ponteiro para aquele bloco.

As instruções mais frequente para o acesso da pilha são PUSH e POP (em ambos x86 e x64). PUSH subtrai de ESP/RSP/SP 4 no modo 32-bits (ou 8 no modo 64-bits) e então escreve o conteúdo desse operando único para o endereço de memória apontado por ESP/RSP/SP.

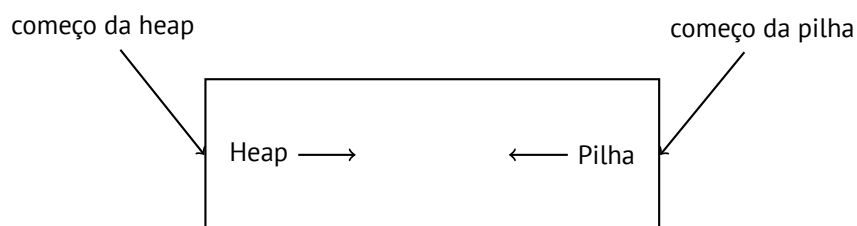
POP é a operação reversa: recupera a informação da localização de memória que é apontada por SP, carrega a mesma no operando da instrução (geralmente um registrador) e então adiciona 4 (ou 8) para o ponteiro da pilha.

Depois da alocação da pilha, o ponteiro aponta para o fundo da pilha. PUSH decrementa o ponteiro da pilha e POP incrementa. O fundo da pilha está na verdade no começo do bloco de memória alocado para ela. Pode parecer estranho, mas é a maneira como é feita.

5.1 Por que a pilha “cresce” para trás?

Intuitivamente, nós podemos pensar que a pilha cresce para frente, em direção a endereços mais altos, como qualquer outra estrutura de informação.

O motivo da pilha crescer para trás é provavelmente histórico. Quando os computadores era grandes e ocupavam um cômodo todo, era mais fácil dividir a memória em duas partes, uma para a ‘heap’ e outra para a pilha. Logicamente, era desconhecido o quão grande a heap e a pilha seriam durante a execução do programa, então essa solução era a mais simples possível.



No [RT74] nós podemos ler:

A parte relacionada ao usuário é dividida em três segmentos lógicos. O segmento de texto do programa começa na localização 0 no espaço virtual de endereçamento. Durante a execução, esse segmento é protegido para não ser reescrito e uma única cópia dele é compartilhado entre todos os processos executando o mesmo programa. Começando no limite de 8Kbytes acima do segmento de texto do programa no espaço de endereçamento virtual começa um segmento de informação gravável, não compartilhável e de um tamanho que pode ser estendido por uma chamada do sistema. Começando no endereço mais alto no espaço de endereçamento virtual está a pilha, que automaticamente cresce para trás conforme o ponteiro da pilha do hardware se altera.

Isso pode ser análogo a como um estudante escreve notas de duas matérias diferentes em um caderno só: as notas para a primeira matéria são escritas como de costume e as notas para a segunda são escritas do final do caderno, virando o mesmo. As anotações de uma matéria podem encontrar as da outra no meio, no caso de haver falta de espaço.

¹wikipedia.org/wiki/Call_stack

²[stack pointer](#). SP/ESP/RSP em x86/x64. SP em ARM.

5.2 Para que a pilha é usada?

5.2.1 Salvar o endereço de retorno de uma função

x86

Quando você chama outra função utilizando a instrução CALL, o endereço do ponto exato onde a instrução CALL se encontra é salvo na pilha e então um jump incondicional para o endereço no operando de CALL é executado.

A instrução CALL é equivalente a usar o par de instruções PUSH endereço_depois_chamada / JMP.

RET pega um valor da pilha e usa um jump para ele – isso é equivalente a usar POP tmp / JMP tmp.

Estourar uma stack é fácil. Só execute alguma recursão externa:

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

O compilador MSVC 2008 informa o problema:

```
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 ↵
↳ runtime stack overflow
```

...mas gera o código de qualquer maneira:

```
?f@YAXXZ PROC ; f
; File c:\tmp6\ss.cpp
; Line 2
    push    ebp
    mov     ebp, esp
; Line 3
    call   ?f@YAXXZ ; f
; Line 4
    pop     ebp
    ret     0
?f@YAXXZ ENDP ; f
```

...também, se ativarmos a otimização do compilador (opção /Ox) o código otimizado não vai estourar a pilha e funcionará *corretamente*³:

```
?f@YAXXZ PROC ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
; Line 3
    jmp     SHORT $LL3@f
?f@YAXXZ ENDP ; f
```

5.2.2 Passando argumento de funções

A maneira mais comum de se passar parâmetros no x86 é chamada «cdecl»:

```
push arg3
push arg2
push arg1
call f
add esp, 12 ; 4*3=12
```

³ironia aqui

Uma função chamada recebe seus argumentos pelo ponteiro da pilha.

Portanto, é assim que os valores dos argumentos são alocados na pilha antes da execução das primeiras instruções da função `f()`:

ESP	endereço de retorno
ESP+4	argumento#1, Marcado no IDA ⁴ como <code>arg_0</code>
ESP+8	argumento#2, Marcado no IDA como <code>arg_4</code>
ESP+0xC	argumento#3, Marcado no IDA como <code>arg_8</code>
...	...

Vale ressaltar que nada obriga o programador a passar os argumentos pela pilha. Não é um requerimento. Você pode implementar qualquer outro método usando a pilha da maneira que desejar.

Por exemplo, é possível alocar um espaço para argumentos na [heap](#), preencher e passar para a função via um ponteiro para esse bloco no registrador EAX.⁵ Isso vai funcionar, entretanto, é de senso comum no x86 e ARM a usar a pilha para esse fim.

A propósito, a função chamada não tem nenhuma informação sobre quantos argumentos foram passados. Funções em C com um número variável de argumentos (como `printf()`) determina seu número usando formatações específicas de string (que começam com o símbolo %).

Se nós escrevermos algo como:

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

`printf()` vai mostrar 1234, e então dois números aleatórios, que estariam próximos a stack.

É por isso que não é muito importante como declaramos a função `main()`: como `main()`, `main(int argc, char *argv[])` ou `main(int argc, char *argv[], char *envp[])`.

Na verdade, o código da C Runtime Library está chamando grosseiramente `main()` dessa maneira:

```
push envp
push argv
push argc
call main
...
```

Se você declarar `main()` como `main()` sem argumentos, mesmo assim eles ainda estarão presentes na pilha, mas não são usados. Se você declarar `main()` como `main(int argc, char *argv[])`, você será capaz de utilizar os primeiros dois argumentos e o terceiro vai continuar «invisível» para a sua função. Da mesma maneira, é possível declarar a `main()` como `main(int argc)` e ainda assim vai funcionar.

5.2.3 Armazenamento de variáveis locais

Uma função poderia alocar espaço na pilha para suas variáveis locais simplesmente decrementando o ponteiro da pilha. Consequentemente é muito rápido, não importando quantas variáveis locais serão definidas.

Também não é um requisito armazenar variáveis locais na pilha. Você pode armazenar variáveis locais onde você quiser, mas, tradicionalmente, é assim que é feito.

5.2.4 x86: a função `alloca()`

A função `alloca()`⁶ funciona da mesma maneira que `malloc()`, mas aloca memória diretamente na pilha.

O bloco de memória alocado não precisa ser limpo através da chamada da função `free()`, desde que o rodapé da função ([4 on page 8](#)) retorna ESP de volta para seu estado inicial e a memória alocada é simplesmente desassociada.

Sobre como a função `alloca()` é implementada, em termos simples, essa função só desloca ESP para baixo (em direção ao fundo da pilha) pelo número de bytes que você precisa e define o ESP como um ponteiro para o bloco alocado.

Vamos tentar:

⁵Por exemplo, no livro “The Art of Computer Programming” por Donald Knuth, na seção 1.4.1 dedicada a subrotinas [[Knu98](#), section 1.4.1]. Nós podemos ler que uma das maneiras de mandar argumentos para uma subrotina é simplesmente listá-los depois de uma instrução `JMP` passando o controle para a subrotina. Knuth explica que esse método era particularmente conveniente no IBM System/360.

⁶No MSVC, a implementação da função pode ser encontrada nos arquivos `alloca16.asm` e `chkstk.asm` em `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

```

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

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __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);
};

```

A função `_snprintf()` funciona exatamente como `printf()`, mas ao invés de jogar o resultado em `stdout` (terminal ou console, por exemplo), ela escreve no buffer `buf`. A função `puts()` copia o conteúdo para um `buf` do `stdout`. Lógico, essas duas chamadas de funções podem ser substituídas por um `printf()`, mas nós temos que ilustrar o uso pequeno do buffer.

MSVC

Vamos compilar (MSVC 2010):

Listing 5.1: MSVC 2010

```

...
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            ; 0000001cH
...

```

O único argumento da função `alloca()` é passado via `EAX` (ao invés de ser empurrado na pilha)⁷. Depois da chamada de `alloca()`, `ESP` aponta para o bloco de 600 bytes que nós podemos usar como memória para o array.

5.2.5 (Windows) SEH

`SEH`⁸ também são guardados na pilha (se estiverem presentes).

5.2.6 Proteção contra estouro de buffer

Mais sobre aqui ([16.2 on page 62](#)).

⁷ Isso é devido ao fato de que `alloca()` é mais nativa do compilador do que uma função normal.

Um dos motivos que se faz necessário o separamento da função ao invés de um pouco de linhas de código no código, é porque a implementação da `alloca()` no MSVC também tem código que é lido da memória que acabou de ser alocada, para deixar o sistema operacional mapear a memória física para essa região da memória virtual.

⁸Structured Exception Handling

5.2.7

Talvez, o motivo para armazenar variáveis locais e registros SEH na pilha é que eles são desvinculados automaticamente depois do fim da função, usando somente uma instrução para corrigir o ponteiro da pilha (geralmente é ADD). Argumentos de funções, como podemos dizer, são também desalocados automaticamente com o fim da função. Como contraste, tudo armazenado na memória heap tem de ser desalocado explicitamente.

5.3 Um modelo típico de pilha

Um modelo típico de pilha em um ambiente 32-bits no início de uma função, antes da execução da primeira instrução, se parece com isso:

...	...
ESP-0xC	Variável local #2, Marcado no IDA como var_8
ESP-8	Variável local #1, Marcado no IDA como var_4
ESP-4	Valor salvo de EBP
ESP	Endereço de retorno
ESP+4	argumento#1, Marcado no IDA como arg_0
ESP+8	argumento#2, Marcado no IDA como arg_4
ESP+0xC	argumento#3, Marcado no IDA como arg_8
...	...

Capítulo 6

printf() com vários argumentos

Agora vamos estender o nosso exemplo (3 on page 5), trocando `printf()` no corpo da função `main()` por isso:

```
#include <stdio.h>

int main()
{
    printf("a=%d; b=%d; c=%d", 1, 2, 3);
    return 0;
};
```

6.1 x86

6.1.1 x86: 3 argumentos

MSVC

Quando compilamos esse código com o MSVC 2010 Express temos:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

    push    3
    push    2
    push    1
    push    OFFSET $SG3830
    call    _printf
    add     esp, 16                ; 00000010H
```

Quase a mesma coisa, mas agora nós podemos ver que os argumentos da função `printf()` são empurrados na pilha na ordem reversa. O primeiro argumento é empurrado por último.

A propósito, variáveis do tipo *int* em ambientes 32-bits tem 32-bits de largura, isso é 4 bytes.

Então, nós temos quatro argumentos aqui, $4 * 4 = 16$ – eles ocupam exatamente 16 bytes na pilha: um ponteiro de 32-bits para uma string e três números do tipo *int*.

Quando o ponteiro da pilha (registrador ESP) volta para seu valor anterior pela instrução `ADD ESP, X` depois de uma chamada de função, geralmente o número de argumentos pode ser obtido simplesmente por se dividir X, o argumento da função `ADD`, por 4.

Lógico que isso é por causa da convenção de chamada do *cdecl* e somente para ambientes 32-bits.

Em certos casos onde varias funções retornam uma depois da outro, o compilador pode multiplicar as instruções «`ADD ESP, X`» em uma só, depois da última chamada:

```
push a1
push a2
call ...
...
push a1
```

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

Aqui está um exemplo do mundo real:

Listing 6.1: x86

```
.text:100113E7      push    3
.text:100113E9      call   sub_100018B0 ; (3)
.text:100113EE      call   sub_100019D0 ;
.text:100113F3      call   sub_10006A90 ;
.text:100113F8      push    1
.text:100113FA      call   sub_100018B0 ; (1)
.text:100113FF      add     esp, 8 ;
```

6.1.2 x64: 8 argumentos

Para ver como outros argumentos são passados pela pilha, vamos mudar nosso exemplo novamente aumentando o número de argumentos para 9 (`printf()` + 8 variáveis `int`):

```
#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

Como mencionado anteriormente, os primeiros 4 argumentos tem de ser passados pelos registradores RCX, RDX, R8, R9 no Win64, enquanto o resto pela pilha. Isso é exatamente o que veremos aqui. Entretanto, a instrução MOV, ao invés de PUSH, é usada para preparar a pilha, portanto os valores são armazenados de uma maneira direta.

Listing 6.2: MSVC 2012 x64

```
$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

          ; 0
          xor     eax, eax

          add     rsp, 88
          ret     0
main      ENDP
_TEXT    ENDS
END
```

Um leitor observativo pode se indagar por que são alocados 8 bytes para valores int quando 4 já é suficiente? Sim, mas lembre-se: 8 bytes são alocados para qualquer tipo de informação menor do que 64 bits. Isso é estabelecido com o objetivo de ser conveniente: é mais fácil calcular o endereço de um argumento arbitrário. Além do mais, eles são alocados em endereços de memórias alinhados. Da mesma maneira no 32-bits: 4 bytes são reservados para todos os tipos de informação.

6.2 Conclusão

Aqui está uma estrutura bem rústica da chamada da função:

Listing 6.3: x86

```
...  
PUSH  
PUSH  
PUSH  
CALL  
;
```

Listing 6.4: x64 (MSVC)

```
MOV RCX,  
MOV RDX,  
MOV R8,  
MOV R9,  
...  
PUSH  
CALL  
;
```

6.3 A propósito

A propósito, a diferença entre os argumentos passados em x86, x64, fastcall, ARM e MIPS é uma boa demonstração do fato de como a CPU é indiferente sobre como os argumentos são passados para as funções. Também é possível criar um compilador hipotético capaz de passar argumentos por alguma outra estrutura especial sem usar a pilha de nenhuma maneira.

A CPU não está ciente de convenções de chamada de funções.

Agora nós podemos também lembrar de dos programadores novatos de assembly passando argumentos para outras funções: geralmente via registradores, sem nenhuma sequência explícita, ou mesmo por variáveis globais. Logicamente, também funciona.

Capítulo 7

scanf()

Agora vamos usar a função `scanf()`.

7.1 Exemplo simples

```
#include <stdio.h>

int main()
{
    int x;
    printf ("Enter X:\n");

    scanf ("%d", &x);

    printf ("You entered %d...\n", x);

    return 0;
};
```

Não é muito inteligente usar `scanf()` para interações com o usuário nos dias de hoje. Mas nós podemos, de qualquer maneira, ilustrar passando um ponteiro para uma variável do tipo `int`.

7.1.1 Sobre ponteiros

Ponteiros são um dos conceitos mais fundamentais na ciência da computação. Com frequência, passar um array grande, estrutura ou objeto como um argumento para outra função é muito custoso, enquanto passar o endereço de onde ele está é bem mais rápido e gasta menos recursos. Ainda mais se a função chamada precisa modificar alguma coisa em um array grande ou estrutura recebida como parâmetro e retornar de volta a estrutura inteira se torna perto de absurdo fazer dessa maneira. Então a coisa mais simples a se fazer é passar o endereço do array ou estrutura para a função chamada e deixar ela fazer as mudanças necessárias.

Um ponteiro em C/C++ é somente um endereço de alguma localização de memória.

Em x86, o endereço é representado como um número de 32-bits (ele ocupa 4 bytes), enquanto no x86-64 é um número de 64-bits (ocupando 8 bytes). A propósito, essa é a razão da indignação de algumas pessoas em relação a trocar para x86-64 todos os ponteiros na arquitetura x64, exigindo o dobro de espaço, incluindo memória cache, que é um lugar “caro”.

É possível ainda se trabalhar com ponteiros sem tipos, como a função padrão em C `memcpy()`, que copia um block de uma localização de memória para outro, ela recebe como argumento dois ponteiros do tipo `void*`, uma vez que é impossível de se prever o tipo de informação que você gostaria de copiar. Tipos não são importantes, só o tamanho do bloco de memória é que importa.

Ponteiros são também largamente usados quando uma função precisa retornar mais de um valor (nós vamos voltar nisso depois).

`scanf()` é um desses casos.

Além do fato de que a função precisa indicar quantos valores foram lidos com sucesso, ela também precisa retornar todos esses valores.

Em C/C++ os tipos dos ponteiros só são necessários para checagem em tempo de compilação.

Internamente, no código compilado não tem nenhuma informação sobre os tipos de cada ponteiro.

7.1.2 x86

MSVC

Aqui está o que o resultado depois de se compilar com o MSVC 2010:

```

CONST    SEGMENT
$SG3831  DB    'Enter X:', 0aH, 00H
$SG3832  DB    '%d', 00H
$SG3833  DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC   _main
EXTRN   _scanf:PROC
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
_x$ = -4                                ; size = 4
_main   PROC
    push    ebp
    mov     ebp, esp
    push    ecx
    push    OFFSET $SG3831 ; 'Enter X:'
    call    _printf
    add     esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push    eax
    push    OFFSET $SG3832 ; '%d'
    call    _scanf
    add     esp, 8
    mov    ecx, DWORD PTR _x$[ebp]
    push    ecx
    push    OFFSET $SG3833 ; 'You entered %d...'
    call    _printf
    add     esp, 8

    ; 0
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS

```

x é uma variável local.

De acordo com os padrões de C/C++ ela só deve ser visível nessa função e não além dela. Tradicionalmente, variáveis locais são guardadas na pilha. Provavelmente há outras maneiras de alocá-las, mas no x86 é assim que é feito.

O objetivo da instrução que se segue após o cabeçalho da função, PUSH ECX, não é para salvar o valor de ECX (perceba que não há a instrução POP ECX no fim da função).

Na verdade, esse PUSH aloca 4 bytes na pilha para guardar a variável x.

x é para ser acessada com a ajuda do macro _x\$ (que é igual a -4) e o registrador EBP apontando para a posição atual.

Conforme a execução da função avança, EBP está apontando para a posição atual da pilha, sendo possível acessar variáveis locais e argumentos da função via EBP+offset.

Também é possível usar ESP para o mesmo objetivo, mas não é muito conveniente pois ele se altera com frequência. O valor de EBP pode ser visto como uma cópia do valor de ESP no começo da execução da função.

Aqui está a aparência típica de uma pilha em um ambiente de 32-bits:

...	...
EBP-8	variável local #2, Marcado no IDA como var_8
EBP-4	variável local #1, Marcado no IDA como var_4
EBP	valor salvo de EBP
EBP+4	Endereço de retorno
EBP+8	argumento#1, Marcado no IDA como arg_0
EBP+0xC	argumento#2, Marcado no IDA como arg_4
EBP+0x10	argumento#3, Marcado no IDA como arg_8
...	...

A função `scanf()` no nosso exemplo tem dois argumentos.

O primeiro é um ponteiro para a string contendo `%d` e a segunda é o endereço da variável `x`.

Primeiro, o endereço da variável `x` é carregado no registrador `EAX` pela instrução `lea eax, DWORD PTR _x$[ebp]`.

Nós podemos dizer que nesse caso, `LEA` simplesmente armazena a soma do valor em `EBP` e o macro `_x$` no registrador `EAX`.

É a mesma coisa que `lea eax, [ebp-4]`.

Então, 4 está sendo subtraído do registrador `EBP` e o resultado é carregado no registrador `EAX`. Depois, o valor em `EAX` é empurrado para dentro da pilha e o `scanf()` é chamado.

`printf()` é chamado depois disso com seu primeiro argumento – um ponteiro para a string: `You entered %d... \n`.

O segundo argumento é preparado com: `mov ecx, [ebp-4]`. A instrução armazena o valor de `x` e não o seu endereço, no registrador `ECX`.

Depois, o valor em `ECX` é armazenado na pilha e o último `printf()` é chamado.

A propósito, esse simples exemplo é a demonstração do fato de que o compilador traduz a lista de expressões em C/C++ em uma lista sequencial de instruções. Não há nada entre as expressões em C/C++, como também no código de máquina resultante, não há nada, o controle de fluxo passa de uma expressão para a outra diretamente.

7.1.3 x64

A situação aqui é parecida, mas com a diferença de que os registradores, ao invés da pilha, são usados para passar argumentos.

MSVC

Listing 7.1: MSVC 2012 x64

```

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

_TEXT SEGMENT
x$ = 32
main PROC
$LN3:
    sub    rsp, 56
    lea   rcx, OFFSET FLAT:$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

; 0
xor     eax, eax
add     rsp, 56

```

```

ret    0
main   ENDP
_TEXT ENDS

```

7.2 Variáveis globais

E se a variável `x` do último exemplo não fosse local, mas sim global? Então ela teria que ser acessível de qualquer ponto, não somente pelo corpo da função. Variáveis globais são consideradas maus hábitos, mas pelo bem do experimento, nós faremos isso.

```

#include <stdio.h>

//
int x;

int main()
{
    printf ("Enter X:\n");

    scanf ("%d", &x);

    printf ("You entered %d...\n", x);

    return 0;
};

```

7.2.1 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

```

Nesse caso, a variável `x` é definida no segmento `_DATA` e nenhuma memória é alocada na pilha local. Ela é acessada diretamente, não através da pilha. Variáveis globais não inicializadas não ocupam espaço no arquivo executável (realmente,

ninguém precisa alocar espaço para uma variável inicialmente valendo zero), mas quando alguém acessa o endereço delas, o sistema operacional vai alocar um bloco contendo somente zeros nele. ¹.

Agora vamos definir um valor para a variável:

```
int x=10; //
```

Nós temos:

```
_DATA  SEGMENT
_x     DD      0aH

...
```

Aqui nós vemos um valor 0xA do tipo DWORD (DD significa DWORD = 32 bits) para essa variável.

Se você abrir o .exe compilado no [IDA](#), você pode ver a variável x colocada no começo do segmento _DATA, e depois disso você pode ver as strings.

Se você abrir o .exe compilado no exemplo anterior no [IDA](#), onde o valor de x não foi declarado, você poderá ver algo assim:

```
.data:0040FA80 _x          dd ?          ; DATA XREF: _main+10
.data:0040FA80          ; _main+22
.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 está marcada com ? juntamente com o resto das variáveis que não precisam ser inicializadas. Isso implica que após carregar o .exe para a memória, um espaço para todas essas variáveis será alocado e preenchido com zeros [[ISO07](#), 6.7.8p10]. Mas no arquivo .exe essas variáveis não inicializadas não ocupam nenhum espaço. Isso é conveniente para arrays grandes, por exemplo.

7.2.2 MSVC: x64

Listing 7.2: 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
$LN3:

    sub     rsp, 40

    lea    rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
    call   printf
    lea    rdx, OFFSET FLAT:x
    lea    rcx, OFFSET FLAT:$SG2925 ; '%d'
    call   scanf
    mov    edx, DWORD PTR x
    lea    rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
    call   printf

    ; 0
    xor    eax, eax
```

¹TBT: That is how a VM² behaves

```

        add     rsp, 40
        ret     0
main     ENDP
_TEXT   ENDS

```

O código é quase o mesmo que no x86. Por favor, perceba que o endereço da variável `x` é passado para `scanf()` usando uma instrução `LEA`, enquanto os valores das variáveis são passadas para o segundo `printf()` usando uma instrução `MOV`. `DWORD PTR` é uma parte da linguagem assembly (sem relação com o código de máquina), indicando que o tamanho da informação da variável é de 32-bits e que a instrução `MOV` tem de ser codificada de acordo.

7.3 Checagem de resultados do scanf()

Como dito anteriormente, está meio fora de moda usar `scanf()` atualmente, mas nós temos que fazer. Precisamos ao menos testar se a função `scanf()` termina corretamente sem nenhum erro.

```

#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;
};

```

Por padrão, a função `scanf()`³ retorna o número de campos que ela leu com sucesso.

No nosso caso, se tudo ocorrer bem e o usuário entrar com um número, `scanf()` retornará 1, ou em caso de erro (ou EOF⁴) – 0.

Vamos adicionar um pouco de código em C para checar o valor de retorno de `scanf()` e imprimir uma mensagem no caso de um erro.

Isso funciona com o desejado:

```

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

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

```

7.3.1 MSVC: x86

Aqui está o a saída em assembly (MSVC 2010):

```

lea     eax, DWORD PTR _x$[ebp]
push   eax
push   OFFSET $SG3833 ; '%d', 00H
call   _scanf
add    esp, 8
cmp    eax, 1
jne    SHORT $LN2@main
mov    ecx, DWORD PTR _x$[ebp]
push   ecx

```

³scanf, wscanf: [MSDN](#)

⁴End of file

```

    push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
    call    _printf
    add     esp, 8
    jmp     SHORT $LN1@main
$LN2@main:
    push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, 00H
    call    _printf
    add     esp, 4
$LN1@main:
    xor     eax, eax

```

A função que chamou (`main()`) precisa do resultado da função chamada (`scanf()`), então a função chamada retorna esse valor no registrador EAX.

Nós verificamos com a ajuda da instrução `CMP EAX, 1` (*CoMParar*). Em outras palavras, comparamos o valor em EAX com 1.

O `jump` condicional `JNE` está logo depois da instrução `CMP`. `JNE` significa *Jump if Not Equal* ou seja, ela desvia se o valor não for igual ao comparado.

Então, se o valor em EAX não é 1, a `CPU` vai passar a execução para o endereço contido no operando de `JNE`, no nosso caso `$LN2@main`. Passando a execução para esse endereço resulta na `CPU` executando `printf()` com o argumento `What you entered? Huh?`. Mas se tudo estiver correto, o `jump` condicional não será efetuado e outra chamada do `printf()` é executada, com dois argumentos: `'You entered %d...'` e o valor de `x`.

Como nesse caso o segundo `printf()` não tem que ser executado, tem um `JMP` precedendo ele (`jump` incondicional). Ele passa a execução para o ponto depois do segundo `printf()` e logo antes de `XOR EAX, EAX`, que implementa `return 0`.

Então, podemos dizer que comparar um valor com outro é geralmente realizado através do par de instruções `CMP/JCC`, onde `cc` é código condicional. `CMP` compara dois valores e altera os registros da `CPU` (flags)⁵. `JCC` checa esses registros e decide passar a execução para o endereço específico contido no operando ou não.

Isso pode parecer meio paradoxal, mas a instrução `CMP` é na verdade `SUB` (subtrair). Todo o conjunto de instruções aritméticas alteram os registros da `CPU`, não só `CMP`. Se compararmos 1 e 1, `1 - 1 = 0` então `ZF` (zero flag) será acionado (significando que o último resultado foi zero). Em nenhuma outra circunstância `ZF` pode ser acionado, exceto quando os operandos forem iguais. `JNE` verifica somente o `ZF` e desvia só não estiver acionado. `JNE` é na verdade um sinônimo para `JNZ` (`jump` se não zero). `JNE` e `JNZ` são traduzidos no mesmo código de operação. Então, a instrução `CMP` pode ser substituída com a instrução `SUB` e quase tudo estará certo, com a diferença de que `SUB` altera o valor do primeiro operando. `CMP` é `SUB` sem salvar o resultado, mas afetando os registros da `CPU`.

⁵TBT: x86 flags, see also: [wikipedia](#).

7.3.2 MSVC: x86 + Hiew

Esse exemplo também pode ser usado como uma maneira simples de exemplificar o patch de arquivos executáveis. Nós podemos tentar rearranjar o executável de forma que o programa sempre imprima a saída, não importando o que inserirmos.

Assumindo que o executável está compilado com a opção /MD⁶ (MSVCR*.DLL), nós vemos a função main no começo da seção .text. Vamos abrir o executável no Hiew e procurar o começo da seção .text (Enter, F8, F6, Enter, Enter).

Nós chegamos a isso:

```

Hiew: ex3.exe
C:\Polygon\ollydbg\ex3.exe  FRO ----- a32 PE .00401000 Hiew 8.02 (c)SEN
.00401000: 55          push     ebp
.00401001: 8BEC       mov     ebp,esp
.00401003: 51          push     ecx
.00401004: 6800304000 push     000403000 ;'Enter X:' --E1
.00401009: FF1594204000 call    printf
.0040100F: 83C404     add     esp,4
.00401012: 8D45FC     lea    eax,[ebp][-4]
.00401015: 50          push     eax
.00401016: 680C304000 push     00040300C --E2
.0040101B: FF158C204000 call    scanf
.00401021: 83C408     add     esp,8
.00401024: 83F801     cmp     eax,1
.00401027: 7514       jnz     .00040103D --E3
.00401029: 8B4DFC     mov     ecx,[ebp][-4]
.0040102C: 51          push     ecx
.0040102D: 6810304000 push     000403010 ;'You entered %d...' --E4
.00401032: FF1594204000 call    printf
.00401038: 83C408     add     esp,8
.0040103B: EB0E       jmps    .00040104B --E5
.0040103D: 6824304000 3push   000403024 ;'What you entered? Huh?' --E6
.00401042: FF1594204000 call    printf
.00401048: 83C404     add     esp,4
.0040104B: 33C0       5xor    eax,eax
.0040104D: 8BE5       mov     esp,ebp
.0040104F: 5D          pop     ebp
.00401050: C3         retn   ; ^^^^
.00401051: B84D5A0000 mov     eax,00005A4D ;' ZM'
1Global 2FileBlk 3CryBlk 4ReLoad 5OrdLdr 6String 7Direct 8Table 91byte 10Leave 11Naked 12AddNam

```

Figura 7.1: Brazilian Portuguese text placeholder

Hiew encontra strings em [ASCIIZ⁷](#) e as exibe, como faz com os nomes de funções importadas.

⁶isso também é chamada “linkagem dinâmica”

⁷ASCII Zero ()


```
x$ = 32
main PROC
$LN5:
    sub     rsp, 56
    lea    rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
    call   printf
    lea    rdx, QWORD PTR x$[rsp]
    lea    rcx, OFFSET FLAT:$SG2926 ; '%d'
    call   scanf
    cmp    eax, 1
    jne    SHORT $LN2@main
    mov    edx, DWORD PTR x$[rsp]
    lea    rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
    call   printf
    jmp    SHORT $LN1@main
$LN2@main:
    lea    rcx, OFFSET FLAT:$SG2929 ; 'What you entered? Huh?'
    call   printf
$LN1@main:
    ; 0
    xor    eax, eax
    add    rsp, 56
    ret    0
main     ENDP
_TEXT   ENDS
END
```

7.4 Ejercicio

- <http://challenges.re/53>