Tracer: users' manual

Dennis Yurichev
<dennis@yurichev.com>

This PDF compilation date: January 28, 2014.
Russian language version of this text (as well as tracer itself) is also accessible at
http://yurichev.com/tracer-en.html

Contents

Preface

Tracer is command-line win32-debugger for performing simple debugging tasks.
Major features:

- Set breakpoint on function execution, track function arguments and result.
- Tracing each instruction of function and dumping register states.
- Set breakpoint on arbitrary point, track CPU registers state and alter them.
- Set breakpoint on memory cell access and track all accesses to it.

Minor features:

- Set breakpoint by address, symbol name or bytemask.
- Unicode string detection in function arguments.
- Both Windows x86 and Windows x64 support.
- Oracle RDBMS .SYM files support.
- Source code included.

Homage

`BPX`, `BPMB`/`BPMW`/`BPMD` options are named after those present in SoftICE, excellent debugger of the past.

Thanks

Alex Ionescu.

Chapter 1

General options

-l:<fname.exe>: load process.

-c:<cmd_line>: define command line for loading process.

For example:

```
tracer.exe -l:bzip2.exe -c:--help
```

If command line contain spaces::

```
tracer.exe -l:rar.exe "-c:a archive.rar *"
```

-a:<fname.exe or PID>: attach to running process by file name or PID number.

Process with that filename should be already loaded. If there're several processes with the same name, tracer will attach to all of them simultaneously.

--loading: dump all module filenames and base addresses while loading (it's DLL files often).

--child: attach to all child processes too.

For example, you could run tracer.exe --child -l:cmd.exe, this will open console cmd.exe window and every process running inside command interpreter will be handled by tracer.

--allsymbols[:<regexp>]: dump all symbols during load or by regular expression:

--allsymbols:somedll.dll!.* can be used for dumping all symbols in some DLL.

--allsymbols:.*printf will print something like this:

```
New symbol. Module=[ntdll.dll], address=[0x77C004BC], name=[_snprintf]
New symbol. Module=[ntdll.dll], address=[0x77B8E61F], name=[_snwprintf]
...
New symbol. Module=[msvcrt.dll], address=[0x75725F37], name=[vswprintf]
New symbol. Module=[msvcrt.dll], address=[0x75726649], name=[vwprintf]
New symbol. Module=[msvcrt.dll], address=[0x756C3D68], name=[wprintf]
```

-s: dump call stack before each breakpoint.

For example:

tracer.exe -l:hello.exe -s bpf=kernel32.dll!WriteFile,args:5

We will see:

```
23B4 (0) KERNEL32.dll!WriteFile (7, "hello to tracer!\r\n", 0x0000000E, 0x0017E3A4, 0) (
    called from 0x7317754E (MSVCR90.dll!_lseeki64+0x56b))
Call stack of thread 0x23B4
return address=731778D8 (MSVCR90.dll!_write+0x9f)
return address=7313FB4A (MSVCR90.dll!_fdopen+0x1c0)
return address=7313F70C (MSVCR90.dll!_flsbuf+0x6e1)
return address=73141E50 (MSVCR90.dll!printf+0x84)
return address=0040100E (hello.exe!BASE+0x100e)
return address=0040116F (hello.exe!BASE+0x116f)
return address=76FCE4A5 (KERNEL32.dll!BaseThreadInitThunk+0xe)
return address=77C9CFED (ntdll.dll!RtlCreateUserProcess+0x8c)
return address=77C9D1FF (ntdll.dll!RtlCreateProcessParameters+0x4e)
23B4 (0) KERNEL32.dll!WriteFile -> 1
```

Stack dump can be very handy, for example, we have a program showing Message Box once and by intercepting `USER32.DLL!MessageBoxA` call we can see a path to this call.

Stack dump feature available for all BPF/BPX/BPM features.

Note: this feature doesn't working in x64 version very well (yet).

If `--dump-fpu` option is set, FPU registers state will be dumped.

If `--dump-xmm` option is set, each XMM registers state will be dumped (if needed) too, unless it is empty.

If `--dump-seh` option is set, all SEH related information will be dumped. For SEH4 information dumping, tracer will use `security_cookie` variable, it will search for it by name in `.MAP` or `.PDB` files.

`-t`: write timestamp at each log line:

`--version`: print current version and date/time of compilation, and also, check for update available for download.

For example:

```
tracer.exe -l:bzip2.exe bpf=cygwin1.dll!fprintf,args:2 -t
```

```
[2013-07-03 07:15:10:056] TID=13056|(0) cygwin1.dll!fprintf (0x611887b0, "%s: For help,
    type: '%s --help'.\n") (called from bzip2.exe!OEP+0x15f1 (0x4025f1))
[2013-07-03 07:15:10:058] TID=13056|(0) cygwin1.dll!fprintf () -> 0x27
```

This feature is useful when one need to log time of some events into journal, like, when exactly some program accessed network.

`--help`: print help.

`-q`: be quiet, no output to console and log file.

`@`: option can be used along with any other options:

```
tracer.exe @filename
```

Each line in the profile represents an option. This can be handy for lengthy and/or often used options, like bytemasks (see below).

`@` option can be used along with any other options:

```
tracer.exe -l:filename.exe @additional_options @even_more_options
```

Chapter 2

How address is defined in tracer

There're 3 ways to define breakpoint address.

- By hexadecimal address: `0x00400000` — that's how address inside of win32-process is to be set. Please note: loading base changing of PE-module is not working out here, so, if you see some address in IDA or any other disassembler, that piece of code may be loaded to another address in memory process (you can use `--loading` options to see, on which base address modules are being loaded).

  So, to set some arbitrary address in specific PE-module, it should be set as: `module.dll!0x400000` — and this address will be corrected automatically if module will be loaded on another base address.

- By symbol.

  For example: `kernel32.dll!writefile`

  Regular expressions can be used here. For example: `.*!printf`: tracer will look for `printf` symbol in each loading module. If the same name occurs in different modules, tracer will use only the first occurence.

  POSIX Extended Regular Expression (ERE) syntax is used here for regular expressions.

  Since this is regular expression, some symbols, like `?`, `.` should be *escaped*. For example, in order to set `?method@class@@QAEHXZ` address, it should be set as `\?method@class@@QAEHXZ`.

  Offset is allowed here. For example: `file.exe!BASE+0x1234` (base is predefined symbol equals to PE file base) or `file.exe!label+0xa`.

Chapter 3

BPF: set breakpoint on function execution

BPF option, in a way, it is a kind of strace[1].

Significant differences with strace are:

- tracer is win32/win64 only.
- Breakpoints not just system calls, but any function.
- Only 4 breakpoints, because of x86 architecture limitation.

BPF option with address without additional options will only track the moment when function was called and what it returns.

For example:

```
tracer.exe -l:bzip2.exe bpf=kernel32.dll!WriteFile
```

```
1188 (0) KERNEL32.dll!WriteFile () (called from 0x610AC912 (cygwin1.dll!sigemptyset+0x1022
    ))
1188 (0) KERNEL32.dll!WriteFile -> 1
```

Note: tracer doesn't know some function is void type, e.g., it doesn't return any value. So it just takes the value at `EAX`/`RAX` register.

Options:

`ARGS:<number>`: define arguments number for the function we would like to intercept.

For example:

```
tracer.exe -l:bzip2.exe -c:--help bpf=kernel32.dll!WriteFile,args:5
```

```
09D0 (0) KERNEL32.dll!WriteFile (0x0000001B, "   If no file names are given, bzip2
    compresses or decompresses", 0x0000003F, "?", 0)
09D0 (0) KERNEL32.dll!WriteFile -> 1
09D0 (0) KERNEL32.dll!WriteFile (0x0000001B, "   from standard input to standard output.
    You can combinesses", 0x0000003B, ";", 0)
09D0 (0) KERNEL32.dll!WriteFile -> 1
09D0 (0) KERNEL32.dll!WriteFile (0x0000001B, "   short flags, so ‘-v -4’ means the same as
     -v4 or -4v, &amp;c.ses", 0x0000003C, "<", 0)
09D0 (0) KERNEL32.dll!WriteFile -> 1
```

What we see here is an attempt to read 5 arguments at each WriteFile function call. If some of these arguments are pointers to some area within process memory, and the data at the pointer can be interpreted as ASCII string, it will be printed instead. This is useful when intercepting string functions like strcmp(), strlen(), strtok(), atoi(), and so on.

[1] http://en.wikipedia.org/wiki/Strace

It is not a problem to make mistake on arguments number (except using `skip_stdcall` option, see below). If defined arguments number greater than real, captured local variables of caller function probably will be printed. Or any other useless junk. If defined arguments number is less than real, then only part of arguments will be visible.

`RT:<number>`: replace the returning value of any function by something else, on fly.

```
tracer.exe -l:filename.exe bpf=function,args:1,rt:0x12345678
```

tracer will put this value to `EAX/RAX` right at the moment when function exited.

`SKIP`: bypass a function. This can be used with `RT` option too.

```
tracer.exe -l:filename.exe bpf=function,args:1,rt:0x12345678,skip
```

This means that the function just gets bypassed and its return value is fixed at 0x12345678.

Note: without "0x" prefix, this value would be interpreted as decimal number.

`SKIP_STDCALL`: the same as <b>SKIP</b> option but rather used for stdcall functions.

The difference between cdecl and stdcall calling conventions is just that cdecl function doesn't align stack pointer at exit (caller should do this). stdcall function aligns stack pointer at exit. cdecl is the most used calling convention. However, stdcall is used in MS Windows. So, if you would like to skip a function in KERNEL32.DLL or USER32.DLL, you should use `skip_stdcall`. Consequently, in this case, tracer must know the exact arguments number, without it the process may crash.[2]

If you'd like to suppress all WriteFile calls, do this:

```
tracer.exe -l:hello.exe bpf=kernel32.dll!WriteFile,args:5,skip_stdcall,rt:1
```

Don't forget to make it return 1, so the caller will not suspect anything! WriteFile arguments number is just 5. Change it to something different, and process crashes.

Note: stdcall calling convention is absent in Windows x64, so this option is absent in win64-version of tracer.

`UNICODE`: treat strings in arguments as unicode (widechar). This could be helpful if you intercept unicode win32 functions with W suffix, for example, MessageBoxW.

Unfortunately, tracer can only automatically detect first half of ASCII table, so multilingual unicode strings will not be detected.

`DUMP_ARGS:<size>`: dump memory on argument (if readable) limited by max size.

If argument contain pointer to valid memory block, it will be printed.

At the function exit, if memory block contents was changed, difference will be printed too.

For example:

```
tracer64.exe -l:test_getlocaltime.exe bpf=.*!getlocaltime,args:1,dump_args:0x30
```

```
TID=6660|(0) KERNEL32.dll!GetLocalTime (0x12ff00) (called from 0x14000100f (getlocaltime.
    exe!BASE+0x100f))
Dump of buffer at argument 1 (starting at 1)
000000000012FF00: 28 FF 12 00 00 00 00 00-00 00 00 00 00 00 00 00 "(..............."
000000000012FF10: 01 00 00 00 00 00 00 00-73 11 00 40 01 00 00 00 "........s..@...."
000000000012FF20: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 "................"
TID=6660|(0) KERNEL32.dll!GetLocalTime -> 0x150
Dump difference of buffer at argument 1 (starting at 1)
0000000000000000: D9 07 0C    06    05   -05    10    24    50 01 "... . . . . $ P."
```

Now we can see how GetLocalTime win32 function fill SYSTEMTIME structure.

`PAUSE:<number>`: Make a pause in milliseconds. 1000 — one second. It is convenient for testing, for creating artifical delays. For example, it is important to know program's behaviour in very slow network environment:

```
tracer.exe -l:test1.exe bpf=WS2_32.dll!WSARecv,pause:1000
```

... or if it will read from some very slow storage:

```
tracer.exe -l:test1.exe bpf=kernel32.dll!ReadFile,pause:1000
```

[2]See also: X86 calling conventions`http://en.wikipedia.org/wiki/X86_calling_conventions`

RT_PROBABILITY:<number>: Used with RT: option in pair, defines probability of RT triggering. For example, if RT:0 and RT_PROBABILITY:30% were set, 0 will be set instead of function's return value in 30% of cases. It's convinient for testing — good written program should handle errors correctly. For example, that's how we can simulate memory allocation errors, 1 malloc() call of 100 will return *NULL*:

```
tracer.exe -l:test1.exe bpf=msvcrt.dll!malloc,rt:0,rt_probability:1%
```

... in 10% of cases, the file will fail to open:

```
tracer.exe -l:test1.exe bpf=kernel32.dll!CreateFile,rt:0,rt_probability:10%
```

Probability may be set in usual manner, as a number in 0 (never) to 1 (always) interval. 10% is 0.1, 3% is 0.03, etc.

About ideas on errors also may be simulated, read here Oracle RDBMS internal self-testing features.

3.1 TRACE option

TRACE: trace each instruction in function and collect all interesting values from registers and memory. After execution, all that information is saved to process.exe.idc, process.exe.txt, process.exe_clear.idc files. .idc-files are IDA scripts, .txt file is grepable by grep, awk and sed.

For example, let's take add_member function from *Using Uninitialized Memory for Fun and Profit*[3] article:

```
int dense[256];
int dense_next=0;
int sparse[256];

void add_member(int i)
{
        dense[dense_next]=i;
        sparse[i]=dense_next;
        dense_next++;

};

int main ()
{
        add_member(123);
        add_member(5);
        add_member(71);
        add_member(99);
}
```

Let's compile it and run tracing on add_member function (determine function address in IDA before):

```
tracer -l:trace_test4.exe bpf=0x00401000,trace:cc
```

We'll get trace_test4.exe.txt file:

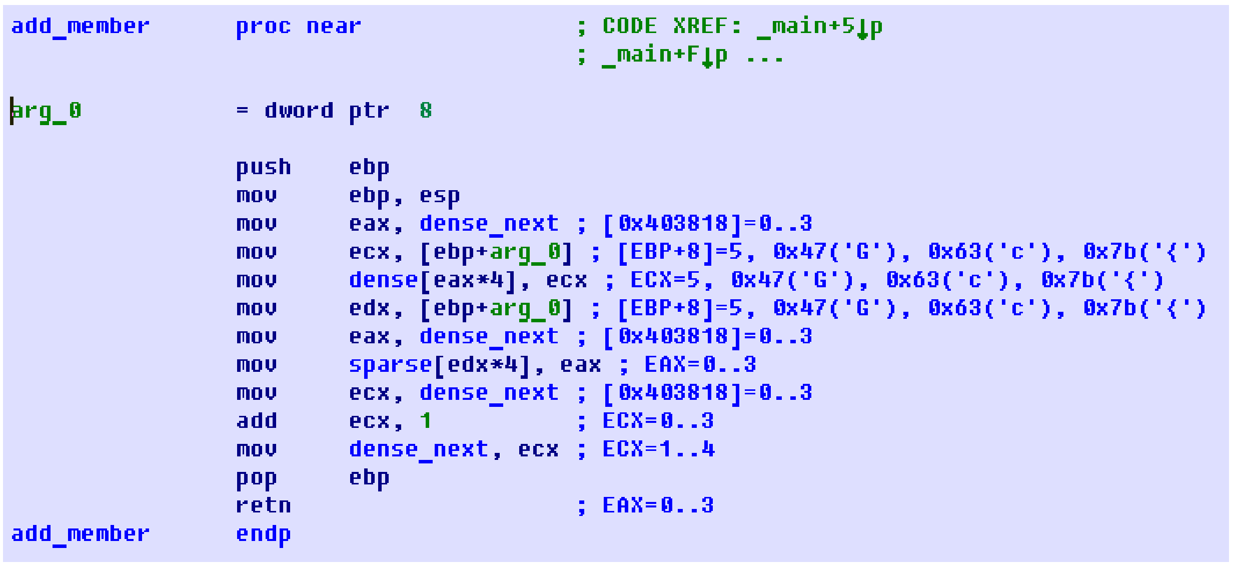
```
0x401000, e=       4
0x401001, e=       4
0x401003, e=       4, [0x403818]=0..3
0x401008, e=       4, [EBP+8]=5, 0x47('G'), 0x63('c'), 0x7b('{')
0x40100b, e=       4, ECX=5, 0x47('G'), 0x63('c'), 0x7b('{')
0x401012, e=       4, [EBP+8]=5, 0x47('G'), 0x63('c'), 0x7b('{')
0x401015, e=       4, [0x403818]=0..3
0x40101a, e=       4, EAX=0..3
```

[3]http://research.swtch.com/2008/03/using-uninitialized-memory-for-fun-and.html

```
0x401021, e=        4, [0x403818]=0..3
0x401027, e=        4, ECX=0..3
0x40102a, e=        4, ECX=1..4
0x401030, e=        4
0x401031, e=        4, EAX=0..3
```

e field is how many times was executed this instruction.

Let's execute trace_test4.exe.idc script in IDA and we'll see:



```
add_member      proc near               ; CODE XREF: _main+5↓p
                                        ; _main+F↓p ...

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, dense_next ; [0x403818]=0..3
                mov     ecx, [ebp+arg_0] ; [EBP+8]=5, 0x47('G'), 0x63('c'), 0x7b('{')
                mov     dense[eax*4], ecx ; ECX=5, 0x47('G'), 0x63('c'), 0x7b('{')
                mov     edx, [ebp+arg_0] ; [EBP+8]=5, 0x47('G'), 0x63('c'), 0x7b('{')
                mov     eax, dense_next ; [0x403818]=0..3
                mov     sparse[edx*4], eax ; EAX=0..3
                mov     ecx, dense_next ; [0x403818]=0..3
                add     ecx, 1          ; ECX=0..3
                mov     dense_next, ecx ; ECX=1..4
                pop     ebp
                retn                    ; EAX=0..3
add_member      endp
```

Figure 3.1: trace_test4.png

Now it is much simpler to understand how this function work during execution.

Executed instructions are highlighed by blue color. Not-executed instructions are leaved white.

If you need to clear all comments and highlight, execute trace_test4.exe_clear.idc script.

All collected information in IDA-script may be reduced to shorten form like *EAX=[64 unique items. min=0xbca6eb7, max=0xfffffed]* (because IDA has comment size limitation). On contrary, everything is saved to text file without shortening, that is why resulting text file may be sometimes pretty big.

One problem of TRACE feature that it is slow, however, functions from system DLLs are skipped (system DLL is that DLL residing in %SystemRoot%) Another problem is that things like exceptions, setjmp/longjmp and other unexpected codeflow alterations are not correctly handled so far.

3.2 Examples

3.2.1 Simple usage

```
tracer.exe -l:bzip2.exe bpf=.*!fprintf,args:3
```

```
TID=5128|(0) cygwin1.dll!fprintf (0x61103150, "%s: I won't write compressed data to a
    terminal.\n", "bzip2") (called from 0x401e03 (bzip2.exe!BASE+0x1e03))
TID=5128|(0) cygwin1.dll!fprintf -> 0x34
TID=5128|(0) cygwin1.dll!fprintf (0x61103150, "%s: For help, type: '%s --help'.\n", "bzip2
    ") (called from 0x401c66 (bzip2.exe!BASE+0x1c66))
TID=5128|(0) cygwin1.dll!fprintf -> 0x27
```

3.2.2 Intercept some Windows registry access functions

```
tracer.exe -l:someprocess.exe bpf=advapi32.dll!RegOpenKeyExA,args:5 bpf=advapi32.dll!
    RegQueryValueExA,args:6 bpf=advapi32.dll!RegSetValueExA,args:6
```

.. or change function suffixes to W and add UNICODE option:

```
tracer64.exe -l:far.exe bpf=advapi32.dll!RegOpenKeyExW,args:5,unicode bpf=advapi32.dll!
    RegQueryValueExW,args:6,unicode bpf=advapi32.dll!RegSetValueExW,args:6,unicode
```

3.2.3 Suppress noisy beeping

```
tracer.exe -l:beeper.exe bpf=kernel32.dll!Beep,args:2,skip_stdcall,rt:1
```

3.2.4 Suppress Message Box

... by making it appear to a caller that the user presses OK every time (IDOK constant is 1):

```
tracer.exe -l:filename.exe bpf=user32.dll!MessageBoxA,args:4,skip_stdcall,rt:1
```

... or CANCEL (IDCANCEL constant is 2):

```
tracer.exe -l:filename.exe bpf=user32.dll!MessageBoxA,args:4,skip_stdcall,rt:2
```

3.2.5 Intercepting rand() call

Another fun is intercepting rand() function in various games. For example, Windows Solitaire card game use it to generate random deal. We can fix rand() return at zero, and Solitaire will do the same deal each time, forever:

In Windows XP x86/x64:

```
tracer.exe/tracer64.exe -l:c:\windows\system32\sol.exe bpf=.*!rand,rt:0
```

In Windows 7 x64:

```
tracer64.exe -l:[full path to]\Solitaire.exe bpf=.*!rand,rt:0
```

3.2.6 FreeCell

When you run Windows (XP SP3) FreeCell and press F2 (New game), you will get a message box "Do you want to resign this game?" We can suppress all that beeping and also make illusion to FreeCell user always press YES:

IDYES constant is 6. FreeCell use MessageBoxW - W mean unicode version of MessageBox.

In Windows XP SP3 x86:

```
tracer.exe -l:c:\windows\system32\freecell.exe bpf=user32.dll!messagebeep,args:1,
    skip_stdcall bpf=user32.dll!messageboxw,args:4,unicode,skip_stdcall,rt:6
```

```
(0) user32.dll!messagebeep (0x20) (called from freecell.exe!BASE+0x1f52 (0x1001f52))
(0) Skipping execution of this function
(0) user32.dll!messagebeep () -> 0x8
(1) user32.dll!messageboxw (0x160152, "Do you want to resign this game?", "FreeCell", 0x24
    ) (called from freecell.exe!BASE+0x1f5f (0x1001f5f))
(1) Skipping execution of this function
(1) user32.dll!messageboxw () -> 0x8
(1) Modifying EAX register to 0x6
```

In Windows XP SP2 x64 Russian:

```
tracer64.exe -l:c:\windows\system32\freecell.exe bpf=user32.dll!messagebeep,args:1,skip
    bpf=user32.dll!messageboxw,args:4,unicode,skip,rt:6
```

```
TID=2836|(0) user32.dll!messagebeep (0x20) (called from freecell.exe!BASE+0x23f9 (0
    x1000023f9))
(0) Skipping execution of this function
TID=2836|(0) user32.dll!messagebeep () -> 0x8
TID=2836|(1) user32.dll!messageboxw (0x5010e, "Do you want to resign this game?", "
    FreeCell", 0x24) (called from freecell.exe!BASE+0x2416 (0x100002416))
(1) Skipping execution of this function
TID=2836|(1) user32.dll!messageboxw () -> 0x8
TID=2836|(1) Modifying RAX register to 0x6
```

3.2.7 Oracle RDBMS Events checking and log writes

In Oracle 10.2.0.1 win64:

```
tracer64.exe -a:oracle.exe bpf=oracle.exe!ksdpec,args:1 bpf=oracle.exe!ss_wrtf,args:3
```

(See also: http://blog.yurichev.com/node/14)

```
TID=3032|(0) oracle.exe!ksdpec (0x2743) (called from 0x9580a9 (oracle.exe!opiodr+0x105))
TID=3032|(0) oracle.exe!ksdpec -> 0xff
TID=3032|(1) oracle.exe!ss_wrtf (0x4a0, "*** 2009-12-04 06:19:01.005\n", 0x1b) (called
    from 0x45318d (oracle.exe!sdpri+0x22d))
TID=3032|(1) oracle.exe!ss_wrtf -> 1
TID=3032|(1) oracle.exe!ss_wrtf (0x4a0, "OPI CALL: type=107 argc= 3 cursor=  0 name=SES
    OPS (80)\n", 0x37) (called from 0x45318d (oracle.exe!sdpri+0x22d))
TID=3032|(1) oracle.exe!ss_wrtf -> 1
TID=3032|(0) oracle.exe!ksdpec (0x2743) (called from 0x9580a9 (oracle.exe!opiodr+0x105))
TID=3032|(0) oracle.exe!ksdpec -> 0xff
TID=3032|(1) oracle.exe!ss_wrtf (0x4a0, "OPI CALL: type=59 argc= 4 cursor=  0 name=
    VERSION2\n", 0x32) (called from 0x45318d (oracle.exe!sdpri+0x22d))
TID=3032|(1) oracle.exe!ss_wrtf -> 1
TID=3032|(0) oracle.exe!ksdpec (0x273e) (called from 0x4a00cc (oracle.exe!kslwte_tm+0x7a8)
    )
TID=3032|(0) oracle.exe!ksdpec -> 0
TID=3032|(0) oracle.exe!ksdpec (0x273e) (called from 0x4a00cc (oracle.exe!kslwte_tm+0x7a8)
    )
TID=3032|(0) oracle.exe!ksdpec -> 0
TID=3032|(0) oracle.exe!ksdpec (0x2743) (called from 0x9580a9 (oracle.exe!opiodr+0x105))
TID=3032|(0) oracle.exe!ksdpec -> 0xff
TID=3032|(1) oracle.exe!ss_wrtf (0x4a0, "OPI CALL: type=104 argc=12 cursor=  0 name=
    Transaction Commit/Rollback\n", 0x46) (called from 0x45318d (oracle.exe!sdpri+0x22d))
TID=3032|(1) oracle.exe!ss_wrtf -> 1
```

3.2.8 Trace memory allocations in Oracle 11.1.0.6.0 win32/win64

```
tracer.exe/tracer64.exe -a:oracle.exe bpf=.*!kghalf,args:6 bpf=.*!kghfrf,args:4
```

```
TID=1600|(0) oracle.exe!kghalf (0x6d35af0, 0xb507ef8, 0x1000, 0, 0, "kzsrcrdi") (called
    from 0x1c7aa83 (oracle.exe!kzctxhugi+0x71))
```

```
TID=1600|(0) oracle.exe!kghalf -> 0xfa3ea58

TID=1600|(0) oracle.exe!kghalf (0x6d35af0, 0xb507ef8, 0x58, 1, 0x6d35530, "UPI heap") (
    called from 0x1e7f8b7 (oracle.exe!__PGOSF266_kwqmahal+0x5b))
TID=1600|(0) oracle.exe!kghalf -> 0xfa4d0d8

TID=1188|(0) oracle.exe!kghalf (0xda39540, 0xda39240, 0x88, 0, "ksirmdt array", 0xda39240)
     (called from 0x6afb5b (oracle.exe!ksz_nfy_ipga+0xf1))
TID=1188|(0) oracle.exe!kghalf -> 0x105d0b10

TID=1188|(0) oracle.exe!kghalf (0xda39540, 0xda39240, 0x48, 1, 0x1204e400, "local") (
    called from 0x3684a64 (oracle.exe!kjztcxini+0x58))
TID=1188|(0) oracle.exe!kghalf -> 0x105d0ab0
```

3.2.9 SQL statements parsing in Oracle RDBMS

In Oracle 11.1.0.6.0 win32/win64:

```
tracer.exe/tracer64.exe -a:oracle.exe bpf=oracle.exe!_?rpisplu,args:8 bpf=oracle.exe!_?
    kprbprs,args:7 bpf=oracle.exe!_?opiprs,args:6 bpf=oraclient11.dll!OCIStmtPrepare,args
    :6</i></p>
```

Note: regular expression `_?function` cover both `function` and `_function`.

```
TID=1140|(2) oracle.exe!opiprs (0x13f029d0, "select 1 from obj$ where name='
    DBA_QUEUE_SCHEDULES'", 0x34, 0x10ae7f50, 0x840082, 0xd9f7a10) (called from 0x6ba3bf (
    oracle.exe!__PGOSF423_kksParseChildCursor+0x2dd))
TID=1140|(2) oracle.exe!opiprs -> 0
TID=1140|(2) oracle.exe!opiprs (0x13f029d0, "select 1 from sys.aq$_subscriber_table where
    rownum < 2 and subscriber_id <> 0 and table_objno <> 0", 0x64, 0x10ad5de8, 0, 0
    x13f007e0) (called from 0x6ba3bf (oracle.exe!__PGOSF423_kksParseChildCursor+0x2dd))
TID=1140|(2) oracle.exe!opiprs -> 0
TID=1140|(0) oracle.exe!rpisplu (3, 0, 0, 0, 0, 0x14430ac0, 0, 0) (called from 0x250b33c (
    oracle.exe!kqdGetCursor+0x106))
TID=1140|(0) oracle.exe!rpisplu -> 0
TID=1288|(2) oracle.exe!opiprs (0x17df8130, "select * from v$version", 0x18, 0x10adee60,
    0, 0) (called from 0x6ba3bf (oracle.exe!__PGOSF423_kksParseChildCursor+0x2dd))
TID=1288|(1) oracle.exe!kprbprs (0xa82bc50, 0, "select timestamp, flags from fixed_obj$
    where obj#=:1", 0x35, 0xffffe3e0, 0x2040800, 1) (called from 0x2ba1b1f (oracle.exe!
    kqldtstr+0x151))
TID=1288|(1) oracle.exe!kprbprs -> 0
TID=1288|(0) oracle.exe!rpisplu (0x1f, 0, 0, 0, 0, 0x2bb5e04, "select  BANNER from
    GV$VERSION where inst_id = USERENV('Instance')", 0xffffc085) (called from 0x2bbcabf (
    oracle.exe!kqldFixedTableLoadCols+0x157))
TID=1288|(1) oracle.exe!kprbprs (0x1090c108, 0, "select timestamp, flags from fixed_obj$
    where obj#=:1", 0x35, 0xffffe3e0, 0x2040800, 1) (called from 0x2ba1b1f (oracle.exe!
    kqldtstr+0x151))
TID=1288|(1) oracle.exe!kprbprs -> 0
TID=1288|(1) oracle.exe!kprbprs (0x10908060, 0, "select timestamp, flags from fixed_obj$
    where obj#=:1", 0x35, 0xffffe3e0, 0x2040800, 1) (called from 0x2ba1b1f (oracle.exe!
    kqldtstr+0x151))
TID=1288|(1) oracle.exe!kprbprs -> 0
TID=1288|(2) oracle.exe!opiprs -> 0
TID=1288|(0) oracle.exe!rpisplu -> 0
```

```
TID=1288|(0) oracle.exe!rpisplu (0x16, 0, 0, 0, 0, 0x10b3ce50, 0, 0) (called from 0
    x250b33c (oracle.exe!kqdGetCursor+0x106))
TID=1288|(0) oracle.exe!rpisplu -> 0
```

3.2.10 Ignore unsigned drivers

```
tracer.exe -l:target.exe bpf=Wintrust.dll!WinVerifyTrust,rt:0
```

3.2.11 Dump function arguments

```
tracer.exe -l:rar.exe "-c:a archive.rar *.exe" bpf=kernel32.dll!writefile,args:5,dump_args
    :0x10
```

RAR writting its signature to the beginning of archive.rar file:

```
TID=7000|(0) KERNEL32.dll!WriteFile (0x118, 0x152410, 7, 0x150fc0, 0) (called from 0
    x403721 (rar.exe!__GetExceptDLLinfo+0x26c8))
Dump of buffer at argument 2 (starting at 1)
00152410: 52 61 72 21 1A 07 00 00-50 30 15 00 5D 83 40 00 "Rar!....P0..].@."
Dump of buffer at argument 4 (starting at 1)
00150FC0: 00 00 00 00 21 7B 40 00-10 24 15 00 18 24 15 00 "....!{@..$...$.."
TID=7000|(0) KERNEL32.dll!WriteFile -> 1
```

3.2.12 Dump function arguments and track difference occured in buffers

```
tracer.exe -l:rar.exe "-c:x archive.rar" bpf=kernel32.dll!readfile,args:4,dump_args:0x10
```

RAR archiver open archive.rar and read signature for the first:

```
TID=6148|(0) KERNEL32.dll!ReadFile (0x120, 0x17b3f8, 7, 0x174c50) (called from 0x403966 (
    rar.exe!__GetExceptDLLinfo+0x290d))
Dump of buffer at argument 2 (starting at 1)
0017B3F8: 00 00 00 00 00 00 00 00-00 00 00 00 48 00 00 00 "............H..."
Dump of buffer at argument 4 (starting at 1)
00174C50: 07 00 00 00 78 4C 17 00-7A 38 40 00 8C 6D 17 00 "....xL..z8@..m.."
TID=6148|(0) KERNEL32.dll!ReadFile -> 1
Dump difference of buffer at argument 2 (starting at 1)
00000000: 52 61 72 21 1A 07      -                        "Rar!..          "
```

3.3 TRACE feature examples

3.3.1 Tracing string functions

Let's take strtok() example:

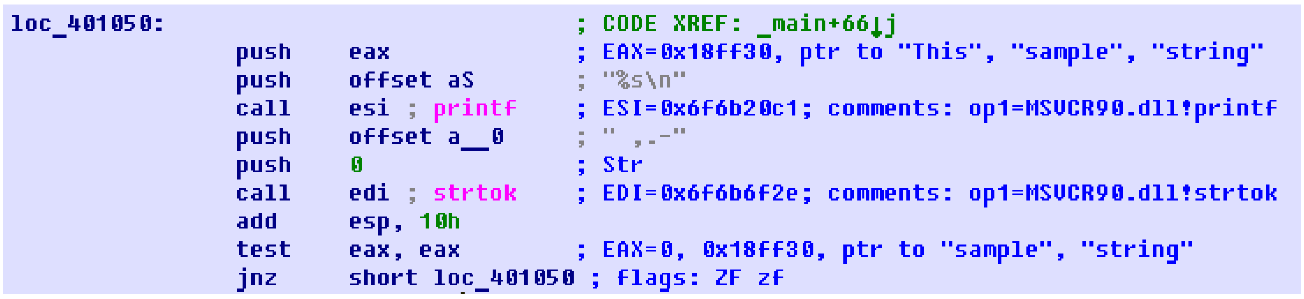
```
// example from http://www.cplusplus.com/reference/clibrary/cstring/strtok/

/* strtok example */
#include <stdio.h>
#include <string.h>

int main ()
```

```
{
  char str[] ="- This, a sample string.";
  char * pch;
  printf ("Splitting string \"%s\" into tokens:\n",str);
  pch = strtok (str," ,.-");
  while (pch != NULL)
  {
    printf ("%s\n",pch);
    pch = strtok (NULL, " ,.-");
  }
  return 0;
}
```

Let's trace main() function:

```
tracer.exe -l:trace_test1.exe bpf=0x00401000,trace:cc
```

After executing resulting .idc script in IDA (only *while* loop body showed here):



```
loc_401050:                               ; CODE XREF: _main+66↓j
                push    eax               ; EAX=0x18ff30, ptr to "This", "sample", "string"
                push    offset aS         ; "%s\n"
                call    esi ; printf      ; ESI=0x6f6b20c1; comments: op1=MSVCR90.dll!printf
                push    offset a__0       ; " ,.-"
                push    0                 ; Str
                call    edi ; strtok      ; EDI=0x6f6b6f2e; comments: op1=MSVCR90.dll!strtok
                add     esp, 10h
                test    eax, eax          ; EAX=0, 0x18ff30, ptr to "sample", "string"
                jnz     short loc_401050 ; flags: ZF zf
```

Figure 3.2: trace_test1.png

Note: "a" is too short string for automatic string detector in tracer, that is why it is absent and its address here instead.

3.3.2 Let's trace quicksort()

Use well-known example:

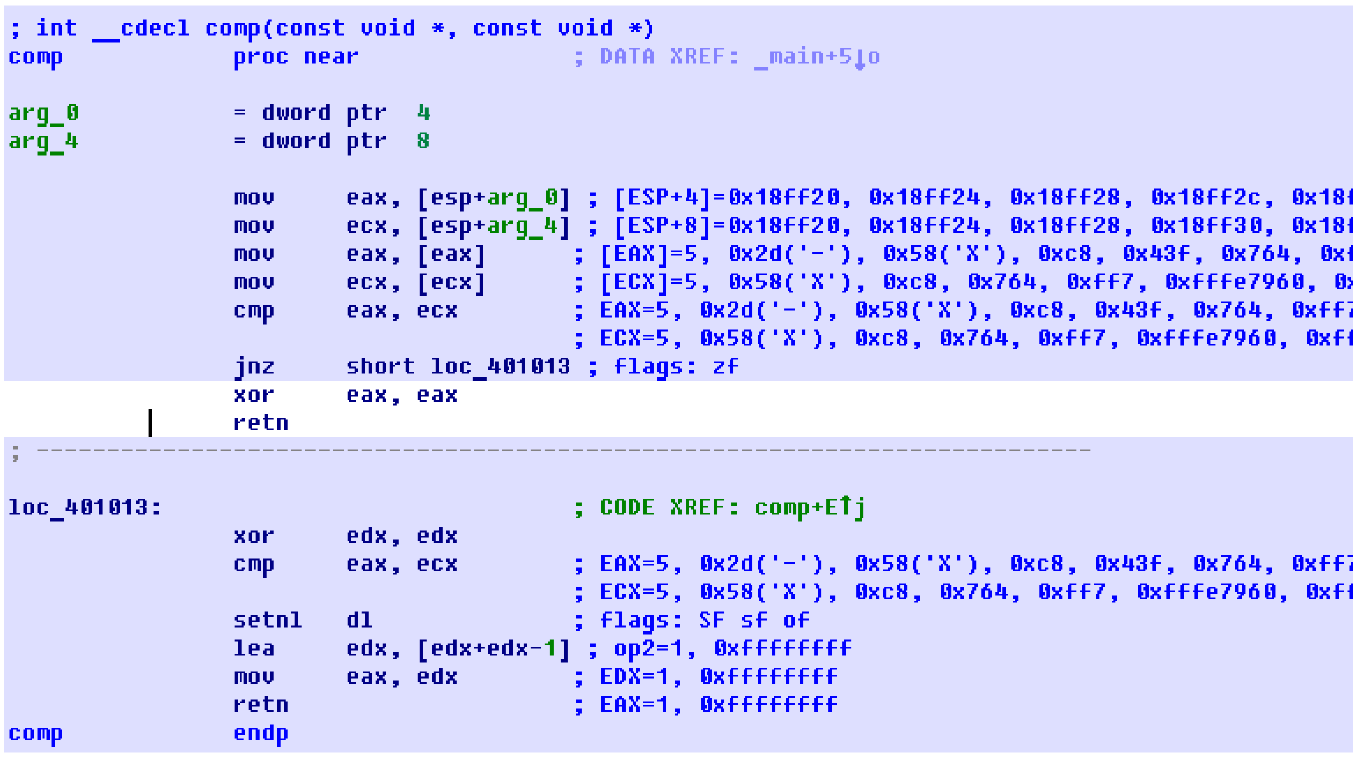
```
//http://cplus.about.com/od/learningc/ss/pointers2_8.htm

/* ex3 Sorting ints with qsort */
//

#include <stdio.h>
#include <stdlib.h>

int comp(const int * a,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;
}
```

Let's trace comp() function:

```
tracer.exe -l:trace_test2.exe bpf=0x00401030,trace:cc
```

We will get after .idc script execution in IDA:



```
; int __cdecl comp(const void *, const void *)
comp            proc near               ; DATA XREF: _main+5↓o

arg_0           = dword ptr  4
arg_4           = dword ptr  8

                mov     eax, [esp+arg_0] ; [ESP+4]=0x18ff20, 0x18ff24, 0x18ff28, 0x18ff2c, 0x18f
                mov     ecx, [esp+arg_4] ; [ESP+8]=0x18ff20, 0x18ff24, 0x18ff28, 0x18ff30, 0x18f
                mov     eax, [eax]      ; [EAX]=5, 0x2d('-'), 0x58('X'), 0xc8, 0x43f, 0x764, 0x
                mov     ecx, [ecx]      ; [ECX]=5, 0x58('X'), 0xc8, 0x764, 0xff7, 0xfffe7960, 0x
                cmp     eax, ecx        ; EAX=5, 0x2d('-'), 0x58('X'), 0xc8, 0x43f, 0x764, 0xff
                                        ; ECX=5, 0x58('X'), 0xc8, 0x764, 0xff7, 0xfffe7960, 0xf
                jnz     short loc_401013 ; flags: zf
                xor     eax, eax
                retn
; ---------------------------------------------------------------------------

loc_401013:                             ; CODE XREF: comp+E↑j
                xor     edx, edx
                cmp     eax, ecx        ; EAX=5, 0x2d('-'), 0x58('X'), 0xc8, 0x43f, 0x764, 0xff
                                        ; ECX=5, 0x58('X'), 0xc8, 0x764, 0xff7, 0xfffe7960, 0xf
                setnl   dl              ; flags: SF sf of
                lea     edx, [edx+edx-1] ; op2=1, 0xffffffff
                mov     eax, edx        ; EDX=1, 0xffffffff
                retn                    ; EAX=1, 0xffffffff
comp            endp
```

Figure 3.3: trace_test2.png

In this example all values are unique, there are no equal ones. Therefore, there are no situation when comp() function returning zero. That is why we see that the comp() part returning zero (xor eax,eax / retn) was not executed.

Chapter 4

BPX: set breakpoint arbitrary point

Content of all CPU registers will be printed.

If at least one FPU register contain something, it will be printed too.

If the floating point number is also NaN (Not-a-Number), FPU register contents will be treated as MMX register and will be dumped too.

`DUMP(ADDRESS|REGISTER[+OFFSET],SIZE)`: dump contents of memory. Define memory address by hexadecimal address or in form `REGISTER+OFFSET`. `SIZE` is memory dump size.

If an asteriks symbol * is set before address or register value, then tracer will read DWORD (or QWORD in x64 version), treat it as address and dump a buffer here. For example: `dump(*ebx,0x100)` — take address on a memory cell EBX register pointing on and dump buffer with size of 0x100 bytes.

`COPY(ADDRESS|REGISTER|SYMBOL[+OFFSET],C-string)`: copy C-string to that address. C-string can be just ASCII-string, but also may contain such sequences like `\xXX`, where `XX` — hexadecimal number. For example: `COPY(EAX,a\x34\x56)` — copy 3 bytes 'a', 0x34, and 0x56 to address from EAX register.

`SET (REGISTER,VALUE)`: set register to value. EIP/RIP, FPU registers ST0..ST7 and flags (PF, SF, AF, ZF, OF, CF, DF) are allowed. Value will be treated as decimal or floating point, unless prefix 0x is present.

Note: tracer never modify FPU tag word register as well as not modify TOP register, so, if some FPU register was marked as "empty" and tracer set some value there, it will remain marked "empty".

Changing EIP/RIP is on other words is code flow altering. This is useful to bypass some code pieces.

4.1 Examples

4.1.1 Task Manager: make illusion we have 32 or 64 CPUs

In Windows XP SP2 x64 Russian:

```
tracer64.exe -l:c:\windows\system32\taskmgr.exe bpx=0x000000010000A8E4,set(rax,64)
```

In Windows XP SP3 x86 English:

```
tracer.exe -l:c:\windows\system32\taskmgr.exe bpx=0x01006647,set(eax,32)
```

4.1.2 Inlined strcmp() intercepting

Let's imagine we have a code we compile in MS VC 2008:

```
printf ("%d\n", strcmp("one", "two"));
```

After compiling we got:

```
<pre>
.text:00401000 BA 50 A1 40 00                   mov     edx, offset aTwo ; "two"
.text:00401005 B9 54 A1 40 00                   mov     ecx, offset aOne ; "one"
.text:0040100A 8D 9B 00 00 00 00                lea     ebx, [ebx+0]
.text:00401010
```

```
.text:00401010                 loc_401010:                             ; CODE XREF:
   _main+2A
.text:00401010 8A 01                           mov     al, [ecx]
.text:00401012 3A 02                           cmp     al, [edx]
.text:00401014 75 29                           jnz     short loc_40103F
.text:00401016 84 C0                           test    al, al
.text:00401018 74 12                           jz      short loc_40102C
.text:0040101A 8A 41 01                        mov     al, [ecx+1]
.text:0040101D 3A 42 01                        cmp     al, [edx+1]
.text:00401020 75 1D                           jnz     short loc_40103F
.text:00401022 83 C1 02                        add     ecx, 2
.text:00401025 83 C2 02                        add     edx, 2
.text:00401028 84 C0                           test    al, al
.text:0040102A 75 E4                           jnz     short loc_401010
.text:0040102C
.text:0040102C                 loc_40102C:                             ; CODE XREF:
   _main+18
.text:0040102C 33 C0                           xor     eax, eax
.text:0040102E 50                              push    eax
.text:0040102F 68 58 A1 40 00                  push    offset byte_40A158 ; char *
.text:00401034 E8 1C 00 00 00                  call    _printf
.text:00401039 83 C4 08                        add     esp, 8
.text:0040103C 33 C0                           xor     eax, eax
.text:0040103E C3                              retn
```

Let's intercept inlined strcmp function and dump what is at ECX and EDX:

```
tracer.exe -l:strcmp.exe bpx=8A013A02752984C074128A41013A4201751D83C10283C20284C075E433C0,
   dump(ecx,0x10),dump(edx,0x10)
```

We got:

```
bytemask_0 is resolved to address 0x401010 (strcmp.exe)
TID=6436|(0) 0x401010 (strcmp.exe!BASE+0x1010)
EAX=0x007722E0 EBX=0x7EFDE000 ECX=0x0040A154 EDX=0x0040A150
ESI=0x00000000 EDI=0x00000000 EBP=0x0018FF88 ESP=0x0018FF44
EIP=0x00401010
FLAGS=PF ZF IF
Dumping memory at ECX
0040A154: 6F 6E 65 00 25 64 0A 00-28 00 6E 00 75 00 6C 00 "one.%d..(.n.u.l."
Dumping memory at EDX
0040A150: 74 77 6F 00 6F 6E 65 00-25 64 0A 00 28 00 6E 00 "two.one.%d..(.n."
```

Note: only first bytemask occurence will be intercepted.

4.1.3 Change flags before conditional dump is taken

```
tracer64.exe -l:flags.exe bpx=0x140001014,set(zf,1)
```

Note: the moment when tracer can change registers state is the moment *before* current instruction is executed. Changing flags *before* TEST or CMP instructions is useless.

4.1.4 Microsoft Excel practical joke

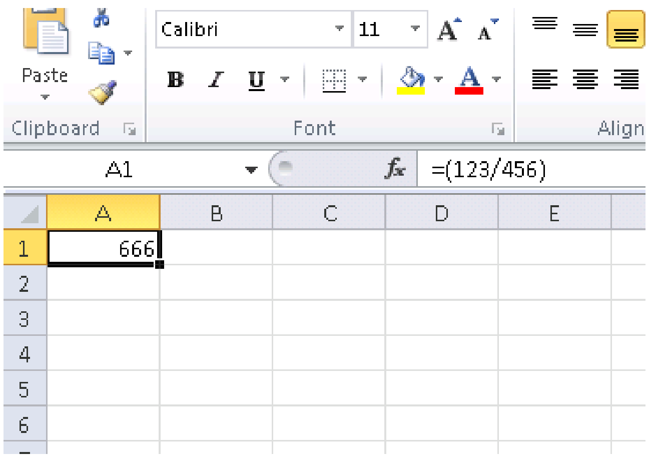
Make result of all divisions 666. Enter "=(123/456)" to check.

Works for Excel.exe version 14.0.4756.1000 (Microsoft Office 2010)

```
tracer.exe -l:excel.exe bpx=excel.exe!base+0x11E91B,set(st0,666)
```

```
tracer64.exe -l:excel.exe bpx=excel.exe!base+0x1B7FCC,set(st0,666)
```

(The address there is the point after FDIV instruction actually do division here)



```
=(123/456)
666
```

Figure 4.1: excel_prank.png

Chapter 5

BPM: set breakpoint on memory access

x86 architecture allows to set breakpoints on a memory value access.

That is, if someone or something modifies some value, tracer will be instantly notified.

It is also should be noted that these breakpoints only practical for global variables, not local ones (stored in stack).

`BPMB=<address>,<option>`: set breakpoint on byte value access. `BPMW=<address>,<option>`: set breakpoint on 16-bit word value access.

`BPMD=<address>,<option>`: set breakpoint on 32-bit dword value access.

`BPMQ=<address>,<option>`: set breakpoint on 62-bit qword value access (available only in tracer64).

`W`: set breakpoint only on memory value write.

`RW`: set breakpoint on both memory value read/write.

Note: because of some unknown reason, Intel achitecture offers only these two opportunities.

5.1 Examples

5.1.1 Tracing value access in Oracle RDBMS

Let's trace read-write access to ktsmgd global variable and see call stack:

```
tracer.exe -a:oracle.exe -s bpmd=oracle.exe!_?ktsmgd_,rw
```

Run in SQL*Plus console (login as SYS before):

```
ALTER SYSTEM SET "_disable_txn_alert"=1;
```

We got:

```
TID=2852|(0) oracle.exe!_ktsmgdcb+0x18: some code reading or writting DWORD variable at
    oracle.exe!_ktsmgd_ (now it contain 0x1)
Call stack of thread TID=2852
return address=0x4682f0 (oracle.exe!_kspptval+0x704)
return address=0x4674b0 (oracle.exe!_kspset0+0x928)
return address=0x8f23c6 (oracle.exe!_kkyasy+0x3cda)
return address=0x92ba1d (oracle.exe!_kksExecuteCommand+0x475)
return address=0x1f75e02 (oracle.exe!_opiexe+0x4bda)
return address=0x1e98390 (oracle.exe!_kpoal8+0x900)
return address=0x9df597 (oracle.exe!_opiodr+0x4cb)
return address=0x6102eb00 (oracommon11.dll!_ttcpip+0xab0)
return address=0x9de77e (oracle.exe!_opitsk+0x4fe)
return address=0x1fdf128 (oracle.exe!_opiino+0x430)
return address=0x9df597 (oracle.exe!_opiodr+0x4cb)
return address=0x450b1c (oracle.exe!_opidrv+0x32c)
return address=0x451352 (oracle.exe!_sou2o+0x32)
return address=0x401197 (oracle.exe!_opimai_real+0x87)
return address=0x401061 (oracle.exe!_opimai+0x61)
return address=0x401c55 (oracle.exe!_OracleThreadStart@4+0x301)
return address=0x77e66063 (KERNEL32.dll!GetModuleFileNameA+0xeb)
```

Visit http://blog.yurichev.com/node/3 for more information about `_disable_txn_alert` parameter and `ktsmgd` value.

5.1.2 Does process checks its own integrity?

Such breakpoints are also useful not only for monitoring variables in memory, but they also can be set on regions of executable code, to get to know, if the process checks integrity of its code, was it modified?

In such cases, often, some function just calculates checksum of the whole executable file, or executable PE-sections, or specific functions. By setting `BPMB` with `R` parameter at the beginning of some functions, it's possible to see, if such checks are happens or not.

Chapter 6

One-time INT3 breakpoint

This breakpoint method allows to set many INT3-type breakpoints by mask. For example, it's possible to set breakpoints to all functions in some DLL:

```
--one-time-INT3-bp:somedll.dll!.*
```

Or, let's set INT3-breakpoints to all functions with `xml` prefix in name:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

On the other side of coin, such breakpoints are triggered only once.

Tracer will show calling of some function, if it happens, but only once. Another drawback — it's not possible to see function's arguments.

Nevertheless, this feature is very useful when you know that some program use some DLL, but don't know which functions. And there are many functions.

For example, let's see, what uptime cygwin-utility uses:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Thus we may see all cygwin1.dll library functions which 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 (0
    x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from uptime.exe!OEP+0xbaa (0
    x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from uptime.exe!OEP+0xcb7 (0
    x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from uptime.exe!OEP+0xcbe (0
    x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from uptime.exe!OEP+0x735 (0x401735)
    )
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from uptime.exe!OEP+0x7b2 (0
    x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from uptime.exe!OEP+0x994 (0x401994)
    )
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from uptime.exe!OEP+0x7ea (0x4017ea
    ))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.exe!OEP+0x809 (0x401809))
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime.exe!OEP+0x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.exe!OEP+0x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.exe!OEP+0x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from uptime.exe!OEP+0x236 (0
    x401236))
```

```
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from uptime.exe!OEP+0x25a (0x40125a)
    )
One-time INT3 breakpoint: cygwin1.dll!setutent (called from uptime.exe!OEP+0x3b1 (0x4013b1
    ))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from uptime.exe!OEP+0x3c5 (0x4013c5
    ))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from uptime.exe!OEP+0x3e6 (0x4013e6
    ))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.exe!OEP+0x4c3 (0x4014c3))
```

Chapter 7

Interacting while running

1) Press ESC or Ctrl-C to detach from the running process.

2) Press SPACE to see current call stacks for each thread of each process.

For example: attach to some running application with opened Message Box, press SPACE and see what probably caused it.

Note: dump call stack feature is not very well working in tracer64.

Chapter 8

Detaching

tracer uses DebugActiveProcessStop() function to detach from the running process. It is present in all modern NT-based operation systems, probably, except Windows NT and Windows 2000. So all tracer can do is just to kill the running process — sorry!

Chapter 9

Some other technical notes

x86 architecture allow to use up to 4 breakpoints simultaneously. So, `BPF`/`BPX`/`BPM` features can be combined in any order up to 4 times.

Stack dumping feature consider stack frames "divided" with EBP base pointer:

See also: Functions and Stack Frames

This means that any function which doesnt use this scheme will be excluded from stack dump — unintentionally.

Note: this feature is not performing very well in tracer64.

All information dumped to stdout is also written to tracer.log file. This file is created at each start.

While loading or attaching, tracer will inspect all modules: main executable and all DLL files loaded after. It will fetch all present symbols, incuding export entries of DLL files. It will also look for FileName.MAP file and try to parse information from it. MAP file has the same format as that produced by IDA disassembler. tracer will also look for FileName.SYM file and try to load symbols from it, treating those as Oracle RDBMS SYM file format: ORACLE_HOME environment value should be set for this. tracer will also look for FileName.PDB file (compile your program in MSVC with `/Zi` option and get debug PDB file for it).

If DLL contain only exports by ordinals, e.g., without names (MFC DLLs, for example), the name of ordinal will be generated in compliance with `ordinal_<number>` format, for example, `ordinal_12`.

Chapter 10

Known issues

10.1 Windows 2000

For running in Windows 2000, Octothorpe library should be compiled with this flag: TARGET_IS_WINDOWS_2000.

Also, dbghelp.dll file from Windows XP should be located in the same folder as tracer.exe.

Chapter 11

Conclusion

This release is not tested properly yet. So please be prepared for any possible crash. I strongly advice to do all experimentation in virtual machine.

If you find any bug, please drop me a line: dennis@yurichev.com. Please attach tracer.log file and screenshot of the last tracer output.

I'll also be thankful for any comments and suggestions related to tracer tool.

If you feel your contribution to source code is worth enough, please send me your patch.

Tracer is also used a lot for illustration purposes in my "Quick introduction to reverse engineering for beginners" book, freely available here.