How debuggers work: Part 2 - Breakpoints

January 27, 2011 at 06:43 Tags Articles , Debuggers , Programming

This is the second part in a series of articles on how debuggers work. Make sure you read the first part before this one.

In this part

I'm going to demonstrate how breakpoints are implemented in a debugger. Breakpoints are one of the two main pillars of debugging - the other being able to inspect values in the debugged process's memory. We've already seen a preview of the other pillar in part 1 of the series, but breakpoints still remain mysterious. By the end of this article, they won't be.

Software interrupts

To implement breakpoints on the x86 architecture, software interrupts (also known as "traps") are used. Before we get deep into the details, I want to explain the concept of interrupts and traps in general.

A CPU has a single stream of execution, working through instructions one by one [1]. To handle asynchronous events like IO and hardware timers, CPUs use interrupts. A hardware interrupt is usually a dedicated electrical signal to which a special "response circuitry" is attached. This circuitry notices an activation of the interrupt and makes the CPU stop its current execution, save its state, and jump to a predefined address where a handler routine for the interrupt is located. When the handler finishes its work, the CPU resumes execution from where it stopped.

Software interrupts are similar in principle but a bit different in practice. CPUs support special instructions that allow the software to simulate an interrupt. When such an instruction is executed, the CPU treats it like an interrupt - stops its normal flow of execution, saves its state and jumps to a handler routine. Such "traps" allow many of the wonders of modern OSes (task scheduling, virtual memory, memory protection, debugging) to be implemented efficiently.

Some programming errors (such as division by 0) are also treated by the CPU as traps, and are frequently referred to as "exceptions". Here the line between hardware and software blurs, since it's hard to say whether such exceptions are really hardware interrupts or software interrupts. But I've digressed too far away from the main topic, so it's time to get back to breakpoints.

int 3 in theory

Having written the previous section, I can now simply say that breakpoints are implemented on the CPU by a special trap called `int 3`. `int` is x86 jargon for "trap instruction" - a call to a predefined interrupt handler. x86 supports the `int` instruction with a 8-bit operand specifying the number of the interrupt that occurred, so in theory 256 traps are supported. The first 32 are reserved by the CPU for itself, and number 3 is the one we're interested in here - it's called "trap to debugger".

Without further ado, I'll quote from the bible itself [2]:

> The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code).

The part in parens is important, but it's still too early to explain it. We'll come back to it later in this article.

int 3 in practice

Yes, knowing the theory behind things is great, OK, but what does this really mean? How do we use `int 3` to implement breakpoints? Or to paraphrase common programming Q&A jargon - *Plz showme the codes!*

In practice, this is really very simple. Once your process executes the `int 3` instruction, the OS stops it [3]. On Linux (which is what we're concerned with in this article) it then sends the process a signal - `SIGTRAP`.

That's all there is to it - honest! Now recall from the first part of the series that a tracing (debugger) process gets notified of all the signals its child (or the process it attaches to for debugging) gets, and you can start getting a feel of where we're going.

That's it, no more computer architecture 101 jabber. It's time for examples and code.

Setting breakpoints manually

I'm now going to show code that sets a breakpoint in a program. The target program I'm going to use for this demonstration is the following:

```
section    .text
    ; The _start symbol must be declared for the linker (ld)
    global _start

_start:

    ; Prepare arguments for the sys_write system call:
    ;   - eax: system call number (sys_write)
    ;   - ebx: file descriptor (stdout)
    ;   - ecx: pointer to string
    ;   - edx: string length
    mov     edx, len1
    mov     ecx, msg1
    mov     ebx, 1
    mov     eax, 4

    ; Execute the sys_write system call
    int     0x80

    ; Now print the other message
    mov     edx, len2
    mov     ecx, msg2
    mov     ebx, 1
    mov     eax, 4
    int     0x80

    ; Execute sys_exit
    mov     eax, 1
    int     0x80

section    .data

msg1    db      'Hello,', 0xa
len1    equ     $ - msg1
msg2    db      'world!', 0xa
len2    equ     $ - msg2
```

I'm using assembly language for now, in order to keep us clear of compilation issues and symbols that come up when we get into C code. What the program listed above does is simply print "Hello," on one line and then "world!" on the next line. It's very similar to the program demonstrated in the previous article.

I want to set a breakpoint after the first printout, but before the second one. Let's say right after the first `int 0x80` [4], on the `mov edx, len2` instruction. First, we need to know what address this instruction maps to. Running `objdump -d`:

```
traced_printer2:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000033  08048080  08048080  00000080  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         0000000e  080490b4  080490b4  000000b4  2**2
                  CONTENTS, ALLOC, LOAD, DATA

Disassembly of section .text:

08048080 <.text>:
 8048080:    ba 07 00 00 00          mov    $0x7,%edx
 8048085:    b9 b4 90 04 08          mov    $0x80490b4,%ecx
 804808a:    bb 01 00 00 00          mov    $0x1,%ebx
 804808f:    b8 04 00 00 00          mov    $0x4,%eax
 8048094:    cd 80                   int    $0x80
 8048096:    ba 07 00 00 00          mov    $0x7,%edx
 804809b:    b9 bb 90 04 08          mov    $0x80490bb,%ecx
 80480a0:    bb 01 00 00 00          mov    $0x1,%ebx
 80480a5:    b8 04 00 00 00          mov    $0x4,%eax
 80480aa:    cd 80                   int    $0x80
 80480ac:    b8 01 00 00 00          mov    $0x1,%eax
 80480b1:    cd 80                   int    $0x80
```

So, the address we're going to set the breakpoint on is 0x8048096. Wait, this is not how real debuggers work, right? Real debuggers set breakpoints on lines of code and on functions, not on some bare memory addresses? Exactly right. But we're still far from there - to set breakpoints like *real* debuggers we still have to cover symbols and debugging information first, and it will take another part or two in the series to reach these topics. For now, we'll have to do with bare memory addresses.

At this point I really want to digress again, so you have two choices. If it's really interesting for you to know *why* the address is 0x8048096 and what does it mean, read the next section. If not, and you just want to get on with the breakpoints, you can safely skip it.

Digression - process addresses and entry point

Frankly, 0x8048096 itself doesn't mean much, it's just a few bytes away from the beginning of the text section of the executable. If you look carefully at the dump listing above, you'll see that the text section starts at 0x08048080. This tells the OS to map the text section starting at this address in the virtual address space given to the process. On Linux these addresses can be absolute (i.e. the executable isn't being relocated when it's loaded into memory), because with the virtual memory system each process gets its own chunk of memory and sees the whole 32-bit address space as its own (called "linear" address).

If we examine the ELF [5] header with `readelf`, we get:

```
$ readelf -h traced_printer2
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048080
  Start of program headers:          52 (bytes into file)
  Start of section headers:          220 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         4
  Section header string table index: 3
```

Note the "entry point address" section of the header, which also points to 0x8048080. So if we interpret the directions encoded in the ELF file for the OS, it says:

1. Map the text section (with given contents) to address 0x8048080
2. Start executing at the entry point - address 0x8048080

But still, why 0x8048080? For historic reasons, it turns out. Some googling led me to a few sources that claim that the first 128MB of each process's address space were reserved for the stack. 128MB happens to be 0x8000000, which is where other sections of the executable may start. 0x8048080, in particular, is the default entry point used by the Linux `ld` linker. This entry point can be modified by passing the `-Ttext` argument to `ld`.

To conclude, there's nothing really special in this address and we can freely change it. As long as the ELF executable is properly structured and the entry point address in the header matches the real beginning of the program's code (text section), we're OK.

Setting breakpoints in the debugger with int 3

To set a breakpoint at some target address in the traced process, the debugger does the following:

1. Remember the data stored at the target address
2. Replace the first byte at the target address with the int 3 instruction

Then, when the debugger asks the OS to run the process (with `PTRACE_CONT` as we saw in the previous article), the process will run and eventually hit upon the `int 3`, where it will stop and the OS will send it a signal. This is where the debugger comes in again, receiving a signal that its child (or traced process) was stopped. It can then:

1. Replace the int 3 instruction at the target address with the original instruction
2. Roll the instruction pointer of the traced process back by one. This is needed because the instruction pointer now points *after* the `int 3`, having already executed it.
3. Allow the user to interact with the process in some way, since the process is still halted at the desired target address. This is the part where your debugger lets you peek at variable values, the call stack and so on.
4. When the user wants to keep running, the debugger will take care of placing the breakpoint back (since it was removed in step 1) at the target address, unless the user asked to cancel the breakpoint.

Let's see how some of these steps are translated into real code. We'll use the debugger "template" presented in part 1 (forking a child process and tracing it). In any case, there's a link to the full source code of this example at the end of the article.

```
/* Obtain and show child's instruction pointer */
ptrace(PTRACE_GETREGS, child_pid, 0, &regs);
procmsg("Child started. EIP = 0x%08x\n", regs.eip);

/* Look at the word at the address we're interested in */
unsigned addr = 0x8048096;
unsigned data = ptrace(PTRACE_PEEKTEXT, child_pid, (void*)addr, 0);
procmsg("Original data at 0x%08x: 0x%08x\n", addr, data);
```

Here the debugger fetches the instruction pointer from the traced process, as well as examines the word currently present at 0x8048096. When run tracing the assembly program listed in the beginning of the article, this prints:

```
[13028] Child started. EIP = 0x08048080
[13028] Original data at 0x08048096: 0x000007ba
```

So far, so good. Next:

```
/* Write the trap instruction 'int 3' into the address */
unsigned data_with_trap = (data & 0xFFFFFF00) | 0xCC;
ptrace(PTRACE_POKETEXT, child_pid, (void*)addr, (void*)data_with_trap);

/* See what's there again... */
unsigned readback_data = ptrace(PTRACE_PEEKTEXT, child_pid, (void*)addr, 0);
procmsg("After trap, data at 0x%08x: 0x%08x\n", addr, readback_data);
```

Note how `int 3` is inserted at the target address. This prints:

```
[13028] After trap, data at 0x08048096: 0x000007cc
```

Again, as expected - `0xba` was replaced with `0xcc`. The debugger now runs the child and waits for it to halt on the breakpoint:

```
/* Let the child run to the breakpoint and wait for it to
** reach it
*/
ptrace(PTRACE_CONT, child_pid, 0, 0);

wait(&wait_status);
if (WIFSTOPPED(wait_status)) {
    procmsg("Child got a signal: %s\n", strsignal(WSTOPSIG(wait_status)));
}
else {
    perror("wait");
    return;
}

/* See where the child is now */
ptrace(PTRACE_GETREGS, child_pid, 0, &regs);
procmsg("Child stopped at EIP = 0x%08x\n", regs.eip);
```

This prints:

```
Hello,
[13028] Child got a signal: Trace/breakpoint trap
[13028] Child stopped at EIP = 0x08048097
```

Note the "Hello," that was printed before the breakpoint - exactly as we planned. Also note where the child stopped - just after the single-byte trap instruction.

Finally, as was explained earlier, to keep the child running we must do some work. We replace the trap with the original instruction and let the process continue running from it.

```
/* Remove the breakpoint by restoring the previous data
** at the target address, and unwind the EIP back by 1 to
** let the CPU execute the original instruction that was
** there.
*/
ptrace(PTRACE_POKETEXT, child_pid, (void*)addr, (void*)data);
regs.eip -= 1;
ptrace(PTRACE_SETREGS, child_pid, 0, &regs);

/* The child can continue running now */
ptrace(PTRACE_CONT, child_pid, 0, 0);
```

This makes the child print "world!" and exit, just as planned.

Note that we don't restore the breakpoint here. That can be done by executing the original instruction in single-step mode, then placing the trap back and only then do `PTRACE_CONT`. The debug library demonstrated later in the article implements this.

More on int 3

Now is a good time to come back and examine `int 3` and that curious note from Intel's manual. Here it is again:

> This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code

`int` instructions on x86 occupy two bytes - `0xcd` followed by the interrupt number [6]. `int 3` could've been encoded as `cd 03`, but there's a special single-byte instruction reserved for it - 0xcc.

Why so? Because this allows us to insert a breakpoint without ever overwriting more than one instruction. And this is important. Consider this sample code:

```
   .. some code ..
   jz   foo
   dec  eax
foo:
   call bar
   .. some code ..
```

Suppose we want to place a breakpoint on `dec eax`. This happens to be a single-byte instruction (with the opcode `0x48`). Had the replacement breakpoint instruction been longer than 1 byte, we'd be forced to overwrite part of the next instruction (`call`), which would garble it and probably produce something completely invalid. But what is the branch `jz foo` was taken? Then, without stopping on `dec eax`, the CPU would go straight to execute the invalid instruction after it.

Having a special 1-byte encoding for `int 3` solves this problem. Since 1 byte is the shortest an instruction can get on x86, we guarantee than only the instruction we want to break on gets changed.

Encapsulating some gory details

Many of the low-level details shown in code samples of the previous section can be easily encapsulated behind a convenient API. I've done some encapsulation into a small utility library called `debuglib` - its code is available for download at the end of the article. Here I just want to demonstrate an example of its usage, but with a twist. We're going to trace a program written in C.

Tracing a C program

So far, for the sake of simplicity, I focused on assembly language targets. It's time to go one level up and see how we can trace a program written in C.

It turns out things aren't very different - it's just a bit harder to find where to place the breakpoints. Consider this simple program:

```c
#include <stdio.h>


void do_stuff()
{
    printf("Hello, ");
}


int main()
{
    for (int i = 0; i < 4; ++i)
        do_stuff();
    printf("world!\n");
    return 0;
}
```

Suppose I want to place a breakpoint at the entrance to `do_stuff`. I'll use the old friend `objdump` to disassemble the executable, but there's a lot in it. In particular, looking at the `text` section is a bit useless since it contains a lot of C runtime initialization code I'm currently not interested in. So let's just look for `do_stuff` in the dump:

```
080483e4 <do_stuff>:
 80483e4:    55                   push   %ebp
 80483e5:    89 e5                mov    %esp,%ebp
 80483e7:    83 ec 18             sub    $0x18,%esp
 80483ea:    c7 04 24 f0 84 04 08    movl   $0x80484f0,(%esp)
 80483f1:    e8 22 ff ff ff       call   8048318 <puts@plt>
 80483f6:    c9                   leave
 80483f7:    c3                   ret
```

Alright, so we'll place the breakpoint at 0x080483e4, which is the first instruction of `do_stuff`. Moreover, since this function is called in a loop, we want to keep stopping at the breakpoint until the loop ends. We're going to use the `debuglib` library to make this simple. Here's the complete debugger function:

```
void run_debugger(pid_t child_pid)
{
    procmsg("debugger started\n");

    /* Wait for child to stop on its first instruction */
    wait(0);
    procmsg("child now at EIP = 0x%08x\n", get_child_eip(child_pid));

    /* Create breakpoint and run to it*/
    debug_breakpoint* bp = create_breakpoint(child_pid, (void*)0x080483e4);
    procmsg("breakpoint created\n");
    ptrace(PTRACE_CONT, child_pid, 0, 0);
    wait(0);

    /* Loop as long as the child didn't exit */
    while (1) {
        /* The child is stopped at a breakpoint here. Resume its
        ** execution until it either exits or hits the
        ** breakpoint again.
        */
        procmsg("child stopped at breakpoint. EIP = 0x%08X\n", get_child_eip(child_pid));
        procmsg("resuming\n");
        int rc = resume_from_breakpoint(child_pid, bp);

        if (rc == 0) {
            procmsg("child exited\n");
            break;
        }
        else if (rc == 1) {
            continue;
        }
        else {
            procmsg("unexpected: %d\n", rc);
            break;
        }
    }


    cleanup_breakpoint(bp);
}
```

Instead of getting our hands dirty modifying EIP and the target process's memory space, we just use `create_breakpoint`, `resume_from_breakpoint` and `cleanup_breakpoint`. Let's see what this prints when tracing the simple C code displayed above:

```
$ bp_use_lib traced_c_loop
[13363] debugger started
[13364] target started. will run 'traced_c_loop'
[13363] child now at EIP = 0x00a37850
[13363] breakpoint created
[13363] child stopped at breakpoint. EIP = 0x080483E5
[13363] resuming
Hello,
[13363] child stopped at breakpoint. EIP = 0x080483E5
[13363] resuming
Hello,
[13363] child stopped at breakpoint. EIP = 0x080483E5
[13363] resuming
Hello,
[13363] child stopped at breakpoint. EIP = 0x080483E5
[13363] resuming
Hello,
world!
[13363] child exited
```

Just as expected!

The code

Here are the complete source code files for this part. In the archive you'll find:

- debuglib.h and debuglib.c - the simple library for encapsulating some of the inner workings of a debugger
- bp_manual.c - the "manual" way of setting breakpoints presented first in this article. Uses the `debuglib` library for some boilerplate code.
- bp_use_lib.c - uses `debuglib` for most of its code, as demonstrated in the second code sample for tracing the loop in a C program.

Conclusion and next steps

We've covered how breakpoints are implemented in debuggers. While implementation details vary between OSes, when you're on x86 it's all basically variations on the same theme - substituting `int 3` for the instruction where we want the process to stop.

That said, I'm sure some readers, just like me, will be less than excited about specifying raw memory addresses to break on. We'd like to say "break on `do_stuff`", or even "break on *this* line in `do_stuff`" and have the debugger do it. In the next article I'm going to show how it's done.

References

I've found the following resources and articles useful in the preparation of this article:

- How debugger works
- Understanding ELF using readelf and objdump
- Implementing breakpoints on x86 Linux
- NASM manual
- SO discussion of the ELF entry point
- This Hacker News discussion of the first part of the series
- GDB Internals

[1] On a high-level view this is true. Down in the gory details, many CPUs today execute multiple instructions in parallel, some of them not in their original order.

[2] The bible in this case being, of course, Intel's Architecture software developer's manual, volume 2A.

[3] How can the OS stop a process just like that? The OS registered its own handler for `int 3` with the CPU, that's how!

[4] Wait, `int` again? Yes! Linux uses `int 0x80` to implement system calls from user processes into the OS kernel. The user places the number of the system call and its arguments into registers and executes `int 0x80`. The CPU then jumps to the appropriate interrupt handler, where the OS registered a procedure that looks at the registers and decides which system call to execute.

[5] ELF (Executable and Linkable Format) is the file format used by Linux for object files, shared libraries and executables.

[6] An observant reader can spot the translation of `int 0x80` into `cd 80` in the dumps listed above.

Comments

Back to top