Modifying a dynamic linked binary *File Descriptor Table* (*FDT*) at runtime catching the `sys_write();` and `sys_writev();` system calls via `Kprobes[4]`.

T. Castillo Girona
<toni.castillo@fa.upc.edu>

September 2, 2009

Abstract

In this article we will discuss how **MODEST** can catch any `sys_write();` and `sys_writev();` system calls using Kprobes. This way, **MODEST** will be able to intercept, capture and redirect any data written by processes linked in a dynamic way.

1 Introduction

Some time ago, `int $0x80` was the only way for a process to request a kernel routine. This matter was extensively described and studied in the previous white-paper concerning our **MODEST** project[3]. Now, it is time to focuss on another way to do that, that is: the `sysenter/sysexit` functions' pair.

Now, any modern Operating System - not only GNU/Linux -, should implement both methods to switch from Ring 3 to Ring 0. In the particular case of GNU/Linux, as seen in our previous study, the static binaries use the old-way `int $0x80`, whereas the dynamic ones use `sysenter/sysexit` functions. Due to this, MODEST was not able to catch the system calls requested by any dynamic binary.

To fix this issue, the latest MODEST implementation module uses Kprobes. Kprobes is fully described in the GNU/Linux Kernel documentation. Kprobes can aid a lot when we are in need of debugging any instruction inside the GNU/Linux Kernel, also the interrupt handlers. This way, all we need to do is to study how Kprobes works, and then adjust it to our problem.

Thus, this article is splitted in two main parts: the first one talks about Kprobes itself; the latest one explains how MODEST was modified so as to use Kprobes to accomplish its goal.

2 Kprobes at a glance

According to the Kernel documentation, there are three different ways to probe a Kernel routine: **Kprobes**, **Jprobes** and **Kretprobes**. The first one can be inserted in any Kernel instruction. The second one can be inserted at the entry point of any Kernel function. The latest one is used when a Kernel function returns.

2.1 Determining where Kprobe should be inserted

Kprobes can be inserted at any Kernel instruction. That means putting a handler anywhere inside the GNU/Linux kernel code. It is fea-

sible, for example, to put this handler at any address inside the Kernel's code segment. As soon as the probed instruction will be reached, this handler will be executed. It is quite obvious that we must know exactly where we have to install this handler routine before doing it.

When it comes to adding a probe point somewhere inside the Kernel code, we can start taking a look at `/proc/kallsyms` [1] or at `/boot/System.map` files. Either way, the address for any symbol will be right there, as shown in Figure 1.

Thus, if we are planning to probe, say, the `sysenter_past_esp` instruction, we have to install our probe handler at `0xc102bbb` address.

2.2 Kernel aids to determine the probe address

We can skip reading those files directly to determine the address of any desired symbol using the Kernel function `kallsyms_lookup_name();`. Bear in mind these addresses will not be the same on different computers, CPU archs or compilers versions.

2.3 The handlers

As described in the GNU/Linux Kernel documentation concerning kprobes, there are three different handlers, each one executed at three different moments: a **pre-handler**, executed just before running the probed instruction; a **post-handler**, executed as soon as the probed instruction finishes; and a **fail-handler**, executed if something goes wrong during the pre or post handlers' execution.

All of them have a parameter called `pt_regs`, a C structure which stores all data stored in the CPU-dependant registers when the probed instruction was hit or when it ends.

[1] This proc special file will be available only in case of enabling the Kernel config option `CONFIG_KALLSYMS`.

Thanks to this data structure, we can retrieve any CPU register value at the time. This data structure is defined in the `include/asm-i386/ptrace.h` header file. For example, during a system call request using the `sysenter/sysexit` functions' pair, as discussed below, the `eax` register stores the system call function number to be executed. So, to obtain this function number, we can read the `eax` field inside the `pt_regs` structure directly, in a C-high style language, skipping inline assembly code, as shown in Figure 2.

3 Using Kprobes inside MODEST

As previously discussed, dynamic binaries, in modern GNU/Linux Kernels, request any system call using the sysenter/sysexit functions' pair on Intel architectures. In fact, any dynamic process request a Kernel routine through the glibc, which, in turn, call the `kernel_vsyscall` function. This function, attached to all of the processes running in the system' address space, could be a call to the old-days `int $0x80` or a call to the new `sysenter/sysexit` mechanism. This way, it is feasible to allow either the old or the new system calls mechanism on the same computer.

3.1 Issuing a system call

As shown in Figure 3, the first three lines of asm code copy the parameters passed to the write(); system call from user stack to the CPU-registers, quite similar when it comes to the `int $0x80` IDT method. Then, the fourth line puts the value `$0x4` in `eax` CPU-register, that is, the `__NR_write` system call.

To issue the call, instead of using the classic int instruction, the glibc executes `call *%gs`[2]`:0x10` instruction. In short,

[2] This register points to the **Thread Control**

```
...
c0102bb4 T sysenter_entry
c0102bbb t sysenter_past_esp
c0102c34 T system_call
...
```

Figure 1: The file /boot/System.map

```
...

#include <linux/syscalls.h>
#include <linux/kprobes.h>
...
int kprobe_prehandler (struct kprobe *p , struct pt_regs *regs){
    if(regs->eax == __NR_write )
    ...
}

...
```

Figure 2: Using pt_regs data structure directly

`@[%gs+0x10]` stores the function described previously, `__kernel_vsyscall()`. So, as soon as this call is performed, the `write` system call will be executed[11].

The offset `$0x10` could be different on other systems, of course.

3.2 Choosing where to put the handler

The sysenter mechanism is implemented in the file `arch/i386/kernel/entry.S`. Taking a look at this asembly source file, we can observe that we could probe exactly this entry point to the Kernel. But we cannot do that just in the sysenter entry point itself but a few lines below, in the label `sysenter_past_esp`. Part of this code is shown in the Figure 5.

In short, the `current` macro will be able to pick the `task_struct *` data structure up for the process issuing the system call right after executing the instruction `movl TSS_sysenter_esp0(%esp),%esp` [5]. So, as soon as the sysenter mechanism reachs the first instruction `sti` placed in `sysenter_past_esp` address, a handler should be executed. In other words, all we need to implement concerning Kprobes will be a pre-handler installed just at this offset. We do not know the exact address of the `sysenter_past_esp` label, of course, but we can get it easily using the Kernel aid `kallsyms_lookup_name`, introduced earlier in this paper.

Therefore, the initialization code for Kprobes is shown in Figure 5. We do not need a post-handler. Maybe it could be a good idea to write a fault handler, to keep an eye over all possibe errors when calling our pre-handler routine. However, the current release of **kmodest.ko** module does not implement a fault handler.

Block (TCB), which, in turn, stores the `AT_SYSINFO` address. This way, any process can find the address of the `__kernel_vsyscall()` function[11].

```
0xb7ed77db <write+11>: mov 0x10(%esp),%edx
0xb7ed77df <write+15>: mov 0xc(%esp),%ecx
0xb7ed77e3 <write+19>: mov 0x8(%esp),%ebx
0xb7ed77e7 <write+23>: mov $0x4,%eax
0xb7ed77ec <write+28>: call *%gs:0x10
```

Figure 3: GDB debugging session showing a call to the write(); system call

```
ENTRY(sysenter_entry)
      CFI_STARTPROC simple
      CFI_DEF_CFA esp, 0
      CFI_REGISTER esp, ebp
      movl TSS_sysenter_esp0(%esp),%esp
sysenter_past_esp:
      /*
      * No need to follow this irqs on/off section:  the syscall
      * disabled irqs and here we enable it straight after entry:
      */
      sti
...
```

Figure 4: GNU/Linux i386 sysenter entry point.

3.3 Catching the `sys_write` and `sys_writev` system calls using the pre-handler

Our pre-handler C code is shown in Figure 7. It is quite easy to understand, merely a few lines quite simmilar to the handler designed to catch the IDT `int $0x80` instruction.

This time, however, the parameters transfered to the system call can be retrieved reading the `pt_regs` data structure, described previously, using C code with no need of implementing low-level asm code. After determining what kind of system call is being requested - that is, `__NR_write` or `__NR_writev` -, the pre-handler must call the operating system scheduler before executing our own implementations for these calls, that is, the `my_sys_write` and `my_sys_writev` routines.

Reading about Kprobes, any code inside our pre-handler function should not sleep. That is because of the fact that the Kprobe code will be always executed with disabled interrupts, and it is not allowed - for security reasons -, to write code which can sleep as long as the interrupts are disabled.

To solve this issue, the pre-handler runs a call to the `schedule()` function.

When our module, **kmodest.ko**, loads, prints out the addresses of the original probed point - in this case, the `sysenter_past_esp` label -, and of the pre-handler - that is, the `kprobe_prehandler()` routine we are talking about -, as shown in Figure 6.

4 Conclussions

Now, **MODEST** project is allowed to redirect any I/O system call through the *"-c"* umod-

```
 ...
kprobe_s.pre_handler = kprobe_prehandler;
kprobe_s.post_handler = NULL;
kprobe_s.fault_handler = NULL;

kprobe_s.addr = (kprobe_opcode_t *)kallsyms_lookup_name("sysenter_past_esp");
...
```

Figure 5: Installing a pre-handler at `sysenter_past_esp address.`

```
 (...):  @ new syscall (int $0x80) at:  0xd0a3c878
(...):  @ sysenter_past_esp at:  0xc0102bbb , catched by 0xd0a3c7d8
```

Figure 6: Printing out the addresses for the original probed point and the pre-handler

est flag and, at the same time, to "peep" the `write()` and `writev()` system calls using the *"-d"* umodest flag. So, **MODEST** can be used either with static or dynamic binaries.

The new approach concerning sysenter mechanism is quite easy to develop or understand. With merely a few code lines, any developer can accomplish a lot of hazardous tasks simply by avoiding low-level asm code in front of these useful and easy-to-use Kernel aids well-known as Kprobes.

It seems quite obvious that the code involving the IDT should be modified so as to avoid using asm code - that is, the `syscalls_entry.S` file -. This way, instead of having two different handlers - one for the `sysenter` and another one for the IDT itself -, the LKM would have only one handler, shared between the two GNU/Linux Kernel mechanisms to enter **Ring 0**. This will be done in **MODEST**'s upcomings revisions.

Now, the main effort will be focussed on skipping the use of a user process - in our particular case it is called **umodest** -, for opening the new file descriptor, that is, the *fd'*, at first instance. Therefore, the LKM will have to do it on its own. Thus, there will be no **SEGFAULTS** related behaviours when it comes to killing the umodest process accidentally.

5 License

```
int kprobe_prehandler (struct kprobe *p , struct pt_regs *regs){

        if(pid_affected!=-1&&(regs->eax==__NR_write||regs->eax==__NR_writev)){

              bytes_to_read = regs->edx;
              userfd = regs->ebx;
              memory = (const char __user *)regs->ecx;

              if(pid_affected==current->pid && userfd==oldfd){
                 switch(regs->eax){
                    case__NR_write:
                        schedule();
                        my_sys_write(krn_fd, memory, bytes_to_read );
                    break;
                    case __NR_writev:
                        schedule();
                        my_sys_writev(krn_fd,(const struct iovec __user*)memory,
                                    bytes_to_read);
                    break;
                 }
              }

        } return 0;

}
```

Figure 7: The kprobes_prehandler() routine.

References

[1] *MODEST project homepage*
http://kmodest.sf.net
http://www.sourceforge.net/projects/kmodest

[2] *Disbaux.es, MODEST page for news concerning this project*
http://disbaux.es/author/tonicas

[3] *Modifying a process File Descriptor Table (FDT) at runtime*
T. Castillo, F. Verdugo & J. Hornos.
Available at *http://kmodest.sf.net*

[4] *Kprobes kernel mainstream documentation.*
`Documentation/kprobes.txt`

[5] *Understanding the Linux Kernel, 3rd edition*
http://oreilly.com/catalog/9780596005658/?CMP=AFCak_book&ATT=Understanding+the+Linux+Kernel

[6] *Essential Linux Device Drivers*
http://www.pearson.ch/Informatik/PrenticeHall/1471/9780132396554/Essential-Linux-Device-Drivers.aspx

[7] *GCC Inline assembly HOW-TO*
http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

[8] *ATT assembly language syntax reference*
http://sig9.com/articles/att-syntax

[9] *Intel X86 Instruction Set Reference A-M*
http://download.intel.com/design/processor/manuals/253666.pdf

[10] *Intel X86 Instruction Set Reference N-Z*
http://download.intel.com/design/processor/manuals/253667.pdf

[11] *Sysenter based System Call mechanism in GNU/Linux 2.6*
http://www.manugarg.com