CPSC 457 – Principles of Operating Systems
Daniel de Castro
Tutorial 20: System call hooking (hijacking)
Expected Time: 30-40 minutes
April 03, 2012

In this exercise, we are going to hook (or "hijack") a system call. In order to do something unharmful, we will only "change" the system call sys_open so it prints a message in /var/log/messages before opening a file. Remember that you are writing a kernel module, so you can basically use the same procedure to change the behavior of your operating system completely.

> IMPORTANT: Boot your VM with its original kernel (to assure no other change will affect this exercise). If you are using Fedora 10, the kernel should be 2.6.27.41-170.2.117.fc10.i686.

1. Create a directory ***hookinglkm*** and access this directory.

2. As we have done before, create the ***Makefile*** to our new module.

```
obj-m += hooking.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

3. To implement our hook, we need to know where the system call table is in memory. We will initially look for it manually. For that, we will look at the ***System.map*** file for the current kernel.

```
$ cat /boot/System.map-`uname -r` | grep sys_call_table
```

   Notice that we are enclosing the ***uname -r*** command with the "left single quote" symbol (which, on an standard US English keyboard, usually shares the key with the tilde, located over the "Tab" key).

   The result should be something like the output below:

```
c06b1db0    R     sys_call_table
```

   The first column indicates the memory address for the system call table. Notice that your value might be different. The second column indicates that it is a read-only memory area. We will come back to that. Memorize the first column. Or, write it down not to forget it.

   Here is a good space for that: _____ (syscall table address)

4. Let us start writing the code for our new module in ***hooking.c***.

```
#include <linux/module.h>     /* Needed by all modules */
#include <linux/init.h>       /* Needed for the macros */

#include <asm/unistd.h>       /* Needed for __NR_close */
#include <linux/syscalls.h>   /* Needed for sys_close */

MODULE_LICENSE("GPL");        /* Don't forget declaring license */

/* Initializer of our LKM */
static int __init hooking_start(void)
{
      /* In the line below, write the address you found */
      unsigned long *table = (unsigned long *) 0xc06b1db0;

      /* Now we check if that address really refers to the
         system call table. We do this by checking if the
         item for an arbitrary syscall (e.g., close) does
         refer to that syscall function address */
      if (table[__NR_close] == (unsigned long) sys_close) {
            printk(KERN_INFO "HOOK: Found it!\n");
      } else {
            /* if it's incorrect, returns -1, so the module
               is not loaded */
            printk(KERN_INFO "HOOK: Nah! Check it again!\n");
            return -1;
      }
      return 0;
}

/* "Destructor" of our LKM */
static void __exit hooking_end(void)
{
      printk(KERN_INFO "HOOK: Goodbye kernel");
}

module_init(hooking_start);
module_exit(hooking_exit);
```

   Notice that we are prepending all the messages with "HOOK". We do that so it gets easier to locate our messages in the file ***/var/log/messages***.

5. Compile our module using ***make*** and try to load it into the kernel using:

```
$ sudo insmod hooking.ko
```

   Now, if the loading is successful, you will find the "Found it" message in the file ***/var/log/messages***.

   Otherwise, you will receive a "Operation not permitted" message. In this case, you should recheck the syscall table address you used and try again, i.e., go back to step 3.

Note: Some developers might prefer to keep an extra ***Terminal*** window open, in which the following command is executed:

```
$ sudo tail -f /var/log/messages
```

By doing this, one can easily verify what messages is being sent to that file.

6. If you could successfully load your kernel: Congratulations, you found the system call table. Now we have to modify it. First, remove our module from the kernel.

```
$ sudo rmmod hooking
```

7. Now, we declare a variable to store the old system call and we also create our own system call. Before the initializer, write the following code:

```
/* variable to store the original call */
asmlinkage int (*o_open) (const char*, int, int);

/* Our new version of sys_open, that prints a message and call
   the original syscall */
asmlinkage int our_sys_open(const char * file, int flags, int mode)
{
      printk(KERN_INFO "HOOK: A file (%s) was opened\n", file);
      return o_open(file, flags, mode);
}
```

8. Now, we actually perform the "hook". In the ***hooking_start*** function, include the following lines before "return 0":

```
o_open = table[__NR_open];       /* Saves the original address in o_open */
table[__NR_open] = our_sys_open; /* Changes the table to point to our
                                    function */
```

And, when removing the module, we should return the table to the normal. So ***hooking_end*** should include the following:

```
table[__NR_open] = o_open;
```

9. As a test, let us compile our module and try to install it.

```
$ make
$ sudo insmod hooking.ko
```

At this point, you should receive several error messages, starting by "OOPS...". In Fedora 10, a box might appear advising that the kernel is compromised. The reason for all of this is that, if you recall, the memory area for ***sys_call_table*** is read-only (see step 3).

10. If you did follow step 9, now you might need to reboot your VM. Remember to choose the original kernel when booting.

11. There are several ways for modifying the memory permissions. Most of them rely on using specific kernel functions, such as ***set_memory_rw*** and ***set_memory_ro*** or ***set_page_rw*** and ***set_page_ro***. While that would be suggested, you might be using different versions of kernel, these functions might not work. In order to complete our implementation, we will use a kernel independent approach, by disabling the memory protection.

    Include the following functions in your program:

```
static void disable_page_protection(void) {

    unsigned long value;
    asm volatile("mov %%cr0,%0" : "=r" (value));
    if (value & 0x00010000) {
            value &= ~0x00010000;
            asm volatile("mov %0,%%cr0": : "r" (value));
    }
}
static void enable_page_protection(void) {

    unsigned long value;
    asm volatile("mov %%cr0,%0" : "=r" (value));
    if (!(value & 0x00010000)) {
            value |= 0x00010000;
            asm volatile("mov %0,%%cr0": : "r" (value));
    }
}
```

12. Now, modify both functions (***hooking_start*** and ***hooking_end***), to disable page protection before modifying the system call table and enabling it back again after the operation. For example, for ***hooking_start***, you should see something like the following:

```
disable_page_protection();
o_open = table[__NR_open];       /* Saves the original address in o_open */
table[__NR_open] = our_sys_open; /* Changes the table to point to our
enable_page_protection;
```

    Don't forget to modify ***hooking_end*** to disable the page protection only when modifying the table, re-enabling it as the table is restored.

13. Finally, compile and install the module. When you check the ***/var/log/messages*** file, you will notice that the system constantly keeps opening different files for the most different reasons. Do not forget removing the module from the kernel (using rmmod), otherwise the log file will quickly increase in size.

14. If you want to practice at home, how about to implement the “Do not trace me” feature (from HW4) into a LKM?

15. Another interesting exercise would be to use the functions ***set_memory_rw***/***set_memory_ro***(or similar) to modify only the memory page that refers to the system call table (and only when necessary).