Homework 4 Wet

Due Date: 21/06/2018 23:30

Teaching assistant in charge:

- Avi Mizrahi

Important: the Q&A for the exercise will take place at a public forum Piazza only. Critical updates about the HW will be published in pinned notes in the piazza forum. These notes are mandatory and it is your responsibility to be updated. A number of guidelines to use the forum:

- Read previous Q&A carefully before asking the question; repeated questions will probably go without answers
- Be polite, remember that course staff does this as a service for the students
- You're not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour
- When posting questions regarding hw4 , put them in the hw4 folder

Only Idan, the TA in charge, can authorize postponements. In case you need a postponement, contact him directly at 234123cs@gmail.com .

Saving Bill from being killed

Introduction

Shelle Driver, a malicious student in the course, heard that the signal *SIGKILL* can't be ignored, and the signaled process will be killed. She worries that one may use this signal to kill a process of Bill, and she wants to prevent this.
Shelle can't modify the kernel, as it requires a reboot that will obviously kill Bill (hence adding/modifying syscalls is not an option), and she doesn't know how to solve this problem.
That's why we are here - we are going to implement a module that intercepts the *kill()* system call and change the way it behaves for a specific program.

In this exercise we will learn how loadable kernel modules (or modules, for short) can hijack system calls and alter their behavior. Malicious modules may use such tricks, for example, to implement a trojan that collects and steals user data.
We remind you that writing a module does not involve any changes to the linux kernel (this is why modules were created...). Still, modules operate in a high privilege level and have access to the kernel address space.
We will need to hack our way through the kernel symbols and carefully modify the kernel data structures.

Detailed Description

Part 1: finding the system call table

We already learned that processes run in two modes: user and kernel. User applications run most of their time under the user mode, so they have access to limited resources. When a process needs to perform a service offered by the kernel, it invokes a *system call*. System calls are software interrupts that the operating system processes in kernel mode.

The Linux kernel maintains a *system call table*, which is simply a set of pointers to functions that implement the system calls. The system call table is stored in the *sys_call_table* variable, which is an array of *void** pointers defined in *arch/i386/kernel/entry.S*. (different system calls have different signatures, so the kernel uses the generic *void** pointers to point to all of them.) The list of system calls implemented by Linux along with their numbers is defined in: *include/asm-i386/unistd.h*.

When our module is loaded, it will hijack the system call by pointing the *sys_call_table* entry to our own-written function. We will also have to save the original function pointer to be able to

restore it later, when our module is unloaded. For example, hijacking the *getpid()* system call will look like:

```
asmlinkage long (*original_sys_getpid)(void);

asmlinkage long our_sys_getpid(void) {
    printk("sys_getpid was called\n");
    /*call original syscall and return its value*/
    return original_sys_getpid();
}

int init_module(void) {
    /*store a reference to the original syscall*/
    original_sys_getpid = sys_call_table[__NR_getpid];
    /*manipulate sys_call_table to point to our fake function*/
    sys_call_table[__NR_getpid] = our_sys_getpid;
    return 0;
}

void cleanup_module(void) {
    /*restore the original syscall*/
    sys_call_table[__NR_getpid] = original_sys_getpid;
}
```

Unfortunately, our module can't just access the *sys_call_table*, since Linux kernel versions 2.5 or greater no longer export the *sys_call_table* structure (The 2.4 kernels shipped with our Red Hat 8.0 is a back-ported version of the 2.6 kernel). Prior to the 2.5 kernels, a module could instantly access the *sys_call_table* structure by declaring it as an extern variable:

```
extern void *sys_call_table[];
```

Although the sys_call_table itself is not exported, a few other system calls such as *sys_read()* and *sys_write()* are still exported and available to modules. We can therefore use a little hack to find the *sys_call_table*, by scanning kernel memory regions and comparing their contents with the addresses of exported system calls. Figure 1 depicts the kernel space memory layout in your RedHat 8.0 machine. You should start scanning at the address of the *system_utsname* structure, which contains a list of system information and is known to be located before the *sys_call_table*. You should then look for a memory location that points to *sys_read()* and deduce that this is the address of *sys_call_table[__NR_read]*.

Note: the *system_utsname* symbol should be imported to your module code with:

```
#include <linux/utsname.h>
```

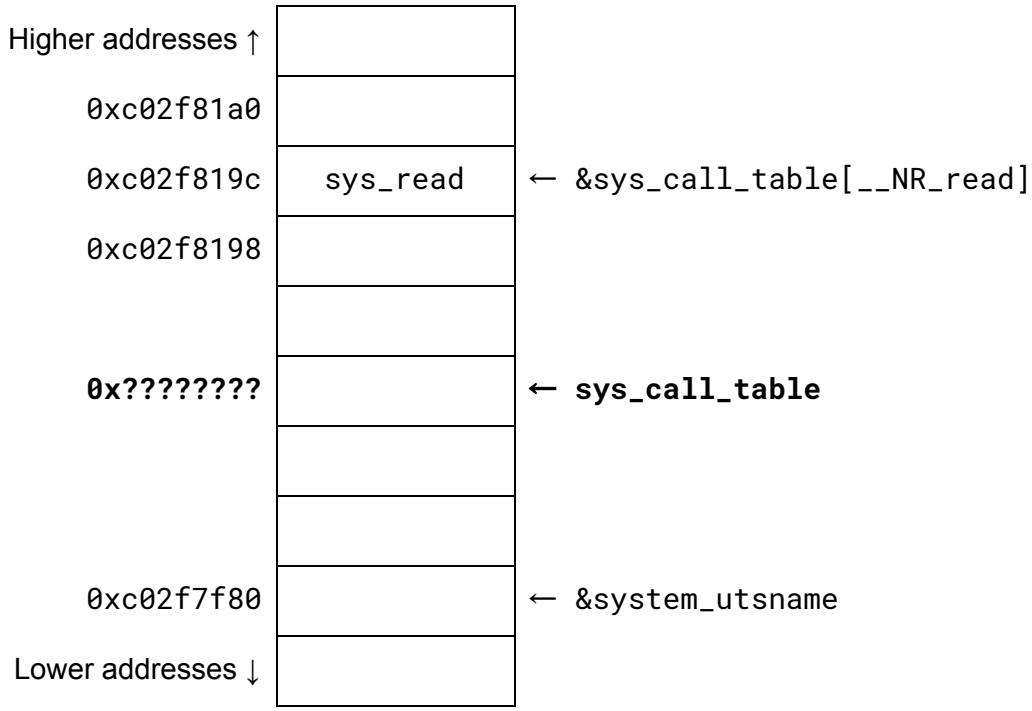

Figure 1: memory layout of the kernel address space

You should fill in the missing code in the *find_sys_call_table()* function located in *intercept.c*. Use a for loop that starts scanning at the location of *system_utsname* and iterates over *scan_range* memory locations. The for loop advances four bytes every time (gcc's default alignment of pointers is four bytes in our architecture) and compares the current content with the address of *sys_read()*. Note: include *<linux/syscall.h>* in order to make *sys_read* symbol visible in your code.

You should also find the number of iterations, *scan_range*, required to discover the *sys_call_table*. A too low number may not find *sys_read* at all; a too high number may find multiple instances of *sys_read* in the kernel memory. It is suggested to pass *scan_range* as a command-line argument and gradually increase it until *sys_read* is found.

Part 2: intercepting *sys_kill()*

Having the address of *sys_call_table*, we can now hijack the *kill()* system call.

You should replace the *sys_kill()* function with your own *our_sys_kill()* and prevent it from killing a process of a program named "Bill". Note that we want to block the signal *SIGKILL*, and allow other signals to be sent. Because Shelle might want to save programs other than "Bill" in the future, the module will depend on a command-line string argument called *program_name*. The *program_name* parameter should have no default value, so you may initialize it to NULL. Your module should do nothing when *program_name* is not set (i.e., when *program_name==NULL*).

The new *our_sys_kill()* function should return:

- *-EPERM*, if:
    a. the signalled process is an instance of the program *program_name*, and
    b. the sent signal is *SIGKILL*
- else - the value returned by the original *sys_kill()*

Important Notes and Tips

- Start working on a clean VM image (can be downloaded from the course website).
- You can validate that you found the correct *sys_call_table* address by looking at */boot/System.map* (use "*grep sys_call_table /boot/System.map*" in your terminal).
- For your convenience, we provide a makefile that builds the module. We will use the same makefile when checking the assignment.
- Before intercepting *sys_kill()*, you may find it easier to try intercepting *sys_getpid()* as explained before.
- Don't forget that the *kill()* syscall sends a general signal, not necessarily *SIGKILL*.
- You can find the *sys_kill()* signature in *kernel/signal.c*.
- Assume *program_name* is a char array of size < 16.
- Use the field ***comm*** in the process descriptor to get the program name.
- You don't need to use synchronization mechanisms in this exercise.
- Don't forget to restore the original *sys_kill()* function when your module is unloaded.
- Why using a "for" loop, rather than a "while" loop, to find the *sys_read()* symbol? Real-life hacking is more complicated than what we just described. For example, Shelle probably wouldn't know the kernel version on which Bill runs, so she would try several techniques like the method that you will use now and hope that one of them will succeed. If she would start scanning from the *system_utsname* symbol infinitely on a different kernel version, she might reach an invalid memory address, page fault, and die.

Submission

You should create a zip file (use zip only, not gzip, tar, rar, 7z or anything else) containing the following files:

a. intercept.c (its template is provided in the course website).
   If you missed a file and because of this, the exercise is not working, you will get 0 and resubmission will cost 10 points. In case you missed an important file (such as the file with all your logic) we may not accept it at all. In order to prevent it you should open the tar on your host machine and see that the files are structured as they supposed to be in the source directory. It is highly recommended to create another clean copy of the guest machine and open the tar there and see it behave as you expected.

b. A file named **submitters.txt** which includes the ID, name and email of the participating students. The following format should be used:

```
Linus Torvalds linus@gmail.com 234567890
Shelle Driver elle.driver@bill.com 345678901
```

Important Note: Make the outlined zip structure exactly. In particular, the zip should contain only the above files, without directories.

You can create the zip by running (inside VMware):

```
zip final.zip intercept.c submitters.txt
```

The zip should look as follows:

```
zipfile -+
         |
         +- intercept.c
         |
         +- submitters.txt
```

Important Note: when you submit, **retain your confirmation code and a copy of the file(s)**, in case of technical failure. Your confirmation code is **the only valid proof** that you submitted your assignment when you did.

Have a Successful Journey,
The course staff