Homework 4 Wet

Due Date: 15/01/2018 23:30

Teaching assistants in charge:
- Sami Zreik (samizr@cs.technion.ac.il)
- Omer Glick (omer.glick@cs.technion.ac.il)

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 the TA in charge can authorize postponements. In case you need a postponement, please fill out the following form: https://goo.gl/forms/nMCWQvYeKEKm4pq82
Introduction

You have worked very hard this semester on your OS homework, so it’s time to kick back and enjoy a relatively simpler assignment.

Shellby Driver, a malicious student taking the Operating Systems course with you this semester, had learned in the 7th tutorial that the signal SIGKILL can’t be ignored. This means that if a processes is signaled with SIGKILL, it will be killed no matter what. She worries that one may use this signal to kill the notoriously known process of Bill, and she wants to prevent this.

Shellby can’t modify the kernel, as it requires a reboot that will obviously kill Bill (hence adding/modifying syscalls is not an option). However, Shellby did not pay enough attention in class, and she does not know how to solve the problem. Therefore Shellby requires your help!

You are going to hack into the kernel and implement a module that intercepts the kill() system call. This module will change the way kill() behaves for a specific program. This is meant to teach you how modules can be used to 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.

Reminder - Writing a module does not involve any changes to the linux kernel (this is why modules were created...), so while implementing this homework you will probably not need to restart the kernel as often as you did implementing the first and second homeworks. Moreover, since 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.
Homework Description

Part 1: finding the system call table

Background and review:

We 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 requires more than those limited resources, it invokes a system call, which offers a variety of services to the user. 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 above described 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. (Note: different system calls have different signatures, this is why the kernel uses the generic void* pointers, to be able 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.

In this part, we would want to write a module that finds the system call table. We will afterwards use it to hijack the ‘kill’ system call by pointing the ‘kill’ system call entry in the sys_call_table to our own, modified kill function. Unfortunately, our module can’t just access the sys_call_table directly. That is because our RedHat 8.0 distribution of the Linux kernel, which is a combination of the 2.4 and 2.6 kernels, does not export the sys_call_table structure. Previous kernel versions allowed users to instantly access the sys_call_table structure by declaring it as an extern variable:

extern void *sys_call_table[];

However, there is still a way to gain access to the table. Even though the sys_call_table itself is not exported, a few other system calls such as sys_read() and sys_write() are exported and available to modules. We can therefore use a little hack to find the sys_call_table. We can scan kernel memory regions and compare 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 (refer to Figure 1 below for more), 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:

```c
#include <linux/utsname.h>
```
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 Shellby 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:
  - The signalled process is an instance of the program program_name.
  - The sent signal is SIGKILL.
- else: the value returned by the original sys_kill()

How should you work?

1. Feel free to start by reviewing the modules and IPC (signals) tutorials. Look into some of the files mentioned above to get a better idea of what you need to do.

2. Load a clean VM image into your VMWare (a new image can be downloaded via the course website).

3. Implement the find_sys_call_table function (provided in intercept.c) to find the system call table. This is how you should implement it:

   - 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().
   - You need to include <linux/sysscall.h> in order to make sys_read symbol visible in your code. Think about how you store it in a variable. For example, if you were to store the address of sys_getpid, you would store it in a variable like this:

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

4. Test your function. You can validate that you have found the correct sys_call_table address by looking at /boot/System.map (use “grep sys_call_table /boot/System.map” in your terminal to extract the address). You can partially implement init_module and cleanup_module to test that easily.

5. Implement the new sys_kill and test it. Edit init_module and cleanup_module accordingly. REMEMBER that you need to restore the pointer to the original sys_kill function once the module cleans up.

Additional Notes and Tips

- For your convenience, we provide a makefile that builds the module and a skeleton for intercept.c. We will use the same makefile when checking the assignment, so you do not need to submit it.
- 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.
- 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, Shellby 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 do not submit this file, naturally the exercise won’t work, and you will get 0 and resubmission will cost 10 points. It is also recommended to create another clean copy of the guest machine and run the module there one last time to see if it behaves 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
   Ken Thompson ken@belllabs.com 345678901
   ```

**Important Note:** Make the outlined zip structure exactly. In particular, the zip should contain only 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