TECHNION
Israel Institute
of Technology

Computer
Science
Department

Operating Systems – 234123

Homework Exercise 3 – Wet

Teaching Assistant in charge:

Sari Khalil and Daniel Bogachenko

Assignment Subjects & Relevant Course material

Modules, System Calls and a bit of Virtual Memory

Recitations 2,5-6, Lectures 2,5-6

Introduction

Shellby Driver, a student taking the Operation Systems course with you this semester, had learned in the 3rd tutorial that the signal SIGKILL can't be ignored. This means that if a process 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 modifying syscalls in a static matter and recompiling is not an option). However, there is another way to save Bill! but Shellby did not pay enough attention in class, and she does not know how to solve the problem. Therefore Shelby 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 behaviour. Malicious modules may use such tricks, for example, to implement a trojan that collects and steal user data (think about a module that alters read/write system call to send the data to someone else and then call the original implementation. That's how hackers can steal your data without you ever knowing it!). 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 structure.

who wouldn't want to dance like Uma Thurman!

Warm-up (writing a system call wrapper!)

As a warm-up, let's start by writing a new wrapper for our syscall. open a new header file called `kill_wrapper.h` and implement the wrapper. The wrapper will have to do the following:

1. print the following message

> "welcome to our kill wrapper!"

2. Move the user parameters to the registers
3. Place the syscall number in rax
4. Execute syscall instruction
5. Handle the syscall's return value

Steps 2-5 should behave as the original kill() system call wrapper. Remember how a system call wrapper should look like, which registers does it put the parameters in, and in what order. If you need more information, you may check out:

- the example given in the 1st Complementary Tutorial, or
- this semester's ATAM tutorials.

We want that calling `kill()` will invoke your wrapper and not the original libc wrapper. Your user code should therefore include `kill_wrapper.h` **and not** include `signal.h` (which is the libc header declaring `kill()`). Go ahead and test your wrapper to see that it calls and handles `kill()` correctly before moving to the next part.

Note:

notice that it is not a good practice to print stuff in your function wrappers, we simply added this requirement so you could assert that your kill calls are going through your wrapper!

Writing a Malicious Module! (to save Bill)

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 a service offered by the kernel, it invokes a *system call*. Furthermore, system calls are the only entry points into the kernel, so they are considered the interface between a process and the operating system. All programs needing resources must use system calls.

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/x86/entry/syscall_64.c` (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 on the x86 architecture, along with their numbers is defined in: `arch/x86/entry/syscalls/syscall_64.tbl` (note that this file isn't written in any programming language, it's just a formatted file)

When our module is loaded (insmod), it will hijack the system call by pointing the *system call table* entry occupied by the original `sys_kill` pointer to our self-written function. We will also have to save the original function pointer to be able to restore it later, when our module is unloaded (rmmod).

Unfortunately, our module can't access the *system call table* directly, since Linux kernel new versions no longer export the *system call table* structure with a header. Therefore, your first mission is to find the address of the *system call table*, by using an internal private kernel function *(see bellow)*.

Having the address of the *system call table*, we would want to change the entry of the sys_kill() to point to our own-written function, but there is one more problem!
The kernel protects itself by turning off the R/W flag in (virtual) pages that contain static read-only structures, like our beloved system call table, therefore we will not be able to update it.
But please do not forget that we are in Kernel Mode which means that we can change the default permissions. Hence, your second mission will be to modify the R/W permission by implementing `allow_rw` function. With that being said we shouldn't forget to disallow the R/W permissions when removing the module (i.e when calling rmmod).

Shelly might want to save programs other than "Bill" in the future, therefore the module will depend on a command-line string argument called `program_name`.
for example, a possible loading command for our module could be:

```
insmod intercept.ko program_name="some_other_program"
```

The `program_name` parameter is an optional parameter where it's default value is "Bill".

Note: Before you start working on the VM please go to the terminal and insert the following command:
`sudo apt install -y linux-headers-4.15.0-60-generic`
This package provides kernel header files for version 4.15.0 on 64 bit x86 SMP.

Now you are set and ready to override the `kill()` system call. Remember the goal of the new implementation is to prevent it from killing a process of a program named "Bill" (or named by the passed `program_name` parameter). Note that we want to block the signal SIGKILL and allow other signals to be sent to "Bill".

Suggested Solution:

Bellow you can find our suggested solution, you will be provided with a skeleton (`intercept.c`) to help you implement it, with that being said, feel free to solve it however you want, you are not obligated to follow the solution or use the provided skeleton. You will also be provided with a `makefile` to help you compile your `intercept.c` file into a module.
`intercept.c` will contain :

int init_module(void):
As you have learned in the tutorial, this function is called when loading the module (i.e insmod <module_name>), to initialize our module we need to do the following:

1. find the address of the system call table (Hint: use `kallsyms_lookup_name` function).
2. call `plug_our_sys_kill` (see below).

void allow_rw(unsigned long addr):
As previously said, some kernel addresses (virtual pages) are read only. For example, the system call table resides in read only virtual pages. This prevents us from changing its content, to do so we would need to alter those permissions.
This function takes an address as an argument and set the R/W flag in the corresponding PTE on. You may use:
`pte_t* lookup_address(unsigned long address, unsigned int *level).`
Google it to find out what it does.

void disallow_rw(unsigned long addr):
We wouldn't want to keep the R/W flag on for the future, therefore this function turns it off as it was before. It does so by finding the corresponding PTE to 'addr' and sets its R/W flag to be turned off.

asmlinkage long our_sys_kill(int pid, int sig):
this is our new implementation of the kill system call and it should do the following :
Return -EPERM, if:

a. The signaled process is an instance of the program_name.
b. And, the sent signal is SIGKILL

Else – the value returned by the original `sys_kill()`.

void plug_our_sys_kill(void):
This function updates the entry of the kill system call in the system call table to point to `our_sys_kill()`, but before updating the entry we should change the R/W permissions. Do not forget to change them back after updating the table.

void unplug_our_syskill(void):
This function updates the entry of the kill system call in the system call table to point back to its original kill system call. but before updating the entry we should change the R/W permissions. Do not forget to change them back after updating the table.

void cleanup_module(void) :
This function is called when removing the module from the kernel (i.e rmmod <module_name>), so it will have to undo the changes it made (restore the system call table to the original state).

Informative Notes and Technical Details

- You must implement this exercise in C. Java/C++/Assembly are prohibited.
- 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 and you must not change it.
- Don't forget that the kill() syscall send a general signal, not necessarily SIGKILL.
- You can find the sys_kill() signatures 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.

Questions & Answers

- The Q&A for the exercise will take place at a public forum Piazza **only**. Please **DO NOT** send questions to the private email addresses of the TAs.
- 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 **hw3**, put them in the **hw3** folder

Late Days

- Please **DO NOT** send postponement requests to the TA responsible for this assignment. Only the **TA in charge** can authorize postponements. In case you need a postponement, please fill out the attached form

Submission

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

1) 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.

2) kill_wrapper.h which contains the wrapper that you wrote.

3) 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 the X files, without directories.

You can create the zip by running:

```
zip XXX_YYY.zip <source_files> submitters.txt
```

The zip should look as follows:

```
zipfile -+
     |
     +- intercept.c
     |
     +- kill_wrapper.h
     |
     +- 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