Homework 4 Wet

Due Date: 14/1/18 23:30

Teaching assistant in charge: **Yehonatan Buchnik**

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 Arie, the TA in charge, can authorize postponements. In case you need a postponement, contact him directly.

Introduction

In this exercise, we will develop a linux kernel module that provides new logging services. To this end, we will learn some new techniques including: inline assembly, hooking, and drivers. The developed functionality will be similar to that we developed in HW1, but instead of changing the kernel itself we will use a loadable kernel module to hook the `system_call` routine and manage the data obtained from it.
We remind you that writing a module does not involve any changes to the linux kernel!
Still, as modules operate in the highest privilege level and have access to the kernel address space, developing a kernel module will make the mission easier and cleaner than what we had in HW1.
The exercise is assembled from 3 parts; this separation should ease your work and help you. Still, the grade is for the whole exercise as a complete one.

To implement the new functionality, we will follow the next steps:

1. We will find the address of the IDT table.
2. Once we have this address, we will assign our implementation of system_call to the 128 entry of the IDT.
3. In the hooked system_call routine we will implement the logging service.
4. We will also have to develop our customized read function for the logged data.

Detailed Description

Part 1: Finding the IDT address

As you have learned, the kernel holds an IDT table, which describes, for every interrupt and exception, the handler routine address and other required data (DPL, for example). To log the system calls data, we will change the corresponding IDT entry to point on our routine.
How can we find the IDT address? Luckily for us, we have a special assembly instruction to load the **idtr** register to a given memory location.

1.1 The idtr Register

The **idtr** is a special 48-bit register that holds the address of the IDT table.
It contains two fields: **base** and **limit**, such that **base** holds the base address of the IDT and **limit** defines the length of the IDT in bytes.

Name	Bit	Description
Limit	0..15	Defines the length of the IDT in bytes - 1 (minimum value is 100h, a value of 1000h means 200h interrupts).
Base	16..47	This 32 bits are the linear address where the IDT starts (INT 0)

The **idtr** can be loaded/stored using the LIDT, SIDT assembly instructions.
(see wikipedia for more)

1.2 Implementation

Go to ***syscalls_logger.c*** and complete the following:

1. complete the struct `struct _desc` such that it will describe exactly the form of the **idtr** register.
2. Use the `store_idt` macro (which accepts an address of type `struct _desc` and stores in it the **idtr** register using the **sidt** assembly), To find the IDT address.
   You can call the macro in the `init_module()` function and use printk to see the result of your work while loading the module.

Part 2: Hooking the IDT

(see wikipedia for more)

2.2 IDT Gate

As you have seen in tutorial r9_interrupts, each IDT entry is called **gate**. Each gate is a 64-bit size and holds the following data:

Name	**Bits**	**Full Name**	**Description**
Offset	48..63	Offset 16..31	Higher part of the offset.
P	47	Present	Set to **0** for unused interrupts.
DPL	45,46	Descriptor Privilege Level	Gate call protection. Specifies which privilege Level the calling Descriptor minimum should have. So hardware and CPU interrupts can be protected from being called out of userspace.
S	44	Storage Segment	Set to **0** for interrupt and trap gates (see below).
Type	40..43	Gate Type 0..3	Possible IDT gate types :

0b0101	0x5	5	80386 32 bit task gate
0b0110	0x6	6	80286 16-bit interrupt gate
0b0111	0x7	7	80286 16-bit trap gate
0b1110	0xE	14	80386 32-bit interrupt gate
0b1111	0xF	15	80386 32-bit trap gate

0	32..39	Unused 0..7	Have to be **0**.
Selector	16..31	Selector 0..15	Selector of the interrupt function (to make sense - the kernel's selector). The selector's descriptor's DPL field has to be **0** so the **iret** instruction won't throw a #GP exeption when executed.
Offset	0..15	Offset 0..15	Lower part of the interrupt function's offset address (also known as pointer).

To hook a gate we will modify the offset field and to leave all other fields unchanged. (Note that the address of the routine, the offset field, is split into high and low parts.)

2.1.1 Implementation

Implement the C structure `struct idtGate` of ***syscalls_logger.c*** such that it will represent the 64-bit in the easiest and clearest way to perform the hooking.

2.2 Creating an assembly routine

We want to hook entry #128 of the IDT and forward system calls to our own `patched_system_call`. At this stage, `patched_system_call` will only call the original `system_call`, such that nothing will change, only to see if we are doing well.
Unfortunately, we can't simply assign the address of `patched_system_call` to IDT entry #128 and call the original *system_call* handler from within `patched_system_call`:
we want the module to be dependent as little as we can in the specific kernel implementation and therefore we won't include the relevant header files, because of that the module is unaware to the *system_call* routine.
But a bigger problem is that while compiling the module the compiler adds the header of the function which include the saving of the **ebp** register and assign **esp** to it, which is devastating to the *system_call* routine (again, why?)
A possible solution to the problem is to declare the function with the ***naked*** attribute which says the compiler not to add the header to the function. Unfortunately, in our "novel" system this feature is unsupported by the compiler.
The solution is to declare a C function with the same name as the assembly routine, that way we trick the compiler such that referring to the C function will cause the program to refer the routine instead.

2.2.1 Implementation

Go to ***syscalls_logger.c*** and implement the `patched_system_call` routine such that it will only jmp to the original *system_call* routine.
Note that you can't call the *system_call* statically as its location may change between different runs of the OS, therefore you have to call it dynamically. (Think how you can obtain the address dynamically.)
Now you can test the correctness of your hooking logic: perform the hooking in `init_module` and make sure your system still operates correctly after it.
Don't forget to restore the original state in `cleanup_module.`

Part 3: Writing the logger device

Your mission is to write a module that creates a logger device, which is responsible to log system call events. When a process opens the device it will log the invocation of the system call that have made by the process.
The logging should be in the following format: `{syscall_num, jiffies, current->time_slice}`. Notice that multiple processes may open the device simultaneously.

Your driver should support the following functions:

- `int open(const char *pathname, int flags)`
  The function register the calling process to the logging service. **After** returning from the function, every system call invocation that will made by the process will be logged in the mentioned format.
  Parameters:
  - pathname - the path to the file which represents the device.
  - flags - permissions.

  Return value:
  On success return the fd of the opened device.
  On failure return -1.

- `int close(int fd)`
  The function closes the logging service for the calling process and frees every data related to it.
  Parametres:
  - fd - the file descriptor to be closed.

  Return value:
  On success return 0.
  On failure return -1.

- `ssize_t read(int fd, void *buf, size_t count)`
  The function writes *count* record in the mentioned format into buf.

If count is greater than the actual records number then write only the current amount.
Note that reading from the device does not erase the readed records.
Parameters:
- fd - the file descriptor of the required device.
- buf - a buffer to write the data into it.
- count - the number of **records** to be written.

Return value:
On success return the number of records which were written.
On failure return -1.

For example, assuming a process with PID=1000 that opened the device, immediately after that it invoked the *getpid* system call when jiffies was 2000 and the process's time_slice is 10.
Then invoking the read function with *count=1* should returns 1 and stores {20, 2000, 10} in buf.

3.2 Implementation Notes

- You can assume that the maximum log size for a given process is 1K of records.
- You can assume that no process calls to more than 1024 system calls while the driver is open.
- If the log is full log no new data should be added.

Important Notes and Tips

- Note that as others file objects, the file object is shared through families relations and through threads.
  It's the user responsibility to coordinate those scenarios. (You don't need to take care of it)
- Start working on a clean VM image (can be downloaded from the course website).
- For your convenience, we provide a makefile that builds the module. We will use the same makefile when checking the assignment.
- Don't forget to restore the original gate function when your module is unloaded.

Submission

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

a. syscalls_logger.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
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 (inside VMware):

```
zip final.zip syscalls_logger.c submitters.txt
```

The zip should look as follows:

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

Have a Successful Journey,
The course staff