Assignment 3

Due Mar 30 by 11:59pm **Points** 18 **Submitting** a file upload **File Types** c and h

Tips, Clarifications

- You may (and probably should) modify code outside of the "CODE HERE" comments. In particular, you may want to modify sleepy_dev in sleepy.h, and you may need to move around the mutex acquire/release in sleepy_read/sleepy_write.
- If at all possible, try to work with a local VM image rather than running on CADE. You can copy the needed VDI from CADE this way: "scp myusername@lab1-1.eng.utah.edu:/home/cs5460/ubuntu-14.04-for-OS.vdi ." or through any scp client. The setup steps given for assignment 2 should work locally as well as on CADE. I recommend using our VM image, since Linux has added protections against things like shady in newer releases.
- VirtualBox's snapshotting functionality can be a helpful debugging tool in this assignment. To use it, open the "Snapshotting" panel under "Machine Tools". There you can create a snapshot of your VM just before you try loading your module into the kernel. If you load your module and it hoses things, you can resume from the snapshot to recover right back to where you were without having to wait to reboot the VM.
- The Blocking I/O section from **Chapter 6** (http://static.lwn.net/images/pdf/LDD3/ch06.pdf) of the Linux Device Driver book and the first few pages of **Chapter 7** (http://static.lwn.net/images/pdf/LDD3/ch07.pdf) on jiffies are mainly what you need to know for this assignment.
- Several other chapters of that book are interesting and provide full detail on the rest of what's going on in the skeleton code. You shouldn't have to read it, but if you want to know more, or have questions on the overall flow/details its all covered in that book. In particular,
  - **Chapter 1** (http://static.lwn.net/images/pdf/LDD3/ch01.pdf) : gives an intro to kernel modules
  - **Chapter 2** (http://static.lwn.net/images/pdf/LDD3/ch02.pdf) : describes their magical Makefiles, roughly how they are compiled, how they are loaded into the kernel, what context they operate in, and how init/uninit works for them.
  - **Chapter 3** (http://static.lwn.net/images/pdf/LDD3/ch03.pdf) : describes character devices, major/minor device numbers, how those interact with the filesystem, how char devs are created, and how char device operations (open/read/write) are bound to device instances.
  - Take a look at these if you want a deeper understanding of Linux, its modules, and how to extend its functionality.
- Recall that the "student" user from the Ubuntu install on the VDI file has a password of "student", which you'll need for "sudo".

Introduction

For this assignment, you will hack a couple of Linux kernel modules. Refer to assignment 2 for instructions about how to compile these modules, load and unload them, etc. You may also refer to any of the several books about writing Linux device drivers and to any of the many web pages about this same topic. In short,

use whatever resources you like **but you must not copy code from your classmates, or even look at their code**. You are required to do your own work.

The files that you need to get started are in the **assignment3** directory in the Files part of the course Canvas page.

A Sleepy Character Driver

The provided files sleepy.c and sleepy.h provide a skeleton character device driver much like the one you were given earlier. One important difference is that this device driver provides 10 instances of the device; they will show up as /dev/sleepy0 through /dev/sleepy9. To be clear: the loadable kernel module is loaded one time but it provides multiple devices. When the Linux kernel invokes functions in this driver, a number will be provided to indicate which device is being communicated with. It is up to you to keep these devices separated from each other in the code that you write.

Just to keep things simple, the sleepy driver will take its commands through the read/write interface. The specification for the functionality you are to implement is:

- If a process writes a 4-byte integer to a sleepy device, it is put to sleep for the number of seconds stored in the integer (if the value is negative, no sleep is performed). [**NOTE:** This is a 4-byte machine integer, it is *not* formatted as ASCII. So you should not try to parse such an integer in your driver (e.g. using sscanf or atoi). Rather, you should read 4 bytes from user space (using copy_from_user() and then treat them as a 4-byte integer. Of course a variable of "int" type in C (on this platform at least) is just 4 bytes]
- If a process writes any other number of bytes, -EINVAL is returned.
- If a process reads from a sleepy device, all processes waiting are immediately awakened.
- When a write to the sleepy driver returns because the sleep ended normally (that is, because the specified number of seconds has elapsed) the return value is zero.
- When a write to the sleepy driver returns because the sleep ended abnormally (that is, because some other process read from the sleepy device) then the number of seconds remaining in that process's sleep is returned.
- Your driver must not use busy-waiting. Sleeping processes are to be placed onto a wait queue.

Remember that each sleepy device is independent of the others. So, for example, a read from /dev/sleepy3 must not wake up processes sleeping on /dev/sleepy7.

The functions that you will use to implement the sleepy driver are described in **chapter 6 (http://lwn.net/images/pdf/LDD3/ch06.pdf)** (Blocking I/O section) and **chapter 7 (http://lwn.net/images/pdf/LDD3/ch07.pdf)** (first three pages about jiffies) of Linux Device Drivers. You should use the interruptible versions of the sleep and wakeup calls.

Important Instructions for Grading

We've added a bit of code to help us test the sleepy module. Check out the assignment3 directory and pick up test_sleepy.c. This is a start of the code we'll use to test yours. You'll need to add a few specific printk calls for us to test your code to sleepy.c.

sleepy_write should emit:
int minor;
minor = (int)iminor(filp->f_path.dentry->d_inode);
printk("SLEEPY_WRITE DEVICE (%d): remaining = %zd \n", minor, remaining_seconds);

and sleepy_read should emit:
int minor;
minor = (int)iminor(filp->f_path.dentry->d_inode);
printk("SLEEPY_READ DEVICE (%d): Process is waking everyone up. \n", minor);

The test_sleepy reads and write from these devices. If you run it and you've added the printks correctly, then the tail our your dmesg output should match or be quite close to ours:
[14757.149176] SLEEPY_READ DEVICE (9): Process is waking everyone up.
[14764.149474] SLEEPY_READ DEVICE (0): Process is waking everyone up.
[14764.151027] SLEEPY_WRITE DEVICE (0): remaining = 1
[14764.151038] SLEEPY_WRITE DEVICE (0): remaining = 1
[14764.151057] SLEEPY_WRITE DEVICE (0): remaining = 1
[14764.151105] SLEEPY_WRITE DEVICE (0): remaining = 1
[14764.151128] SLEEPY_WRITE DEVICE (0): remaining = 1
[14764.151647] SLEEPY_WRITE DEVICE (0): remaining = 1
[14764.151792] SLEEPY_WRITE DEVICE (0): remaining = 1
[14764.151915] SLEEPY_WRITE DEVICE (0): remaining = 1
[14764.152030] SLEEPY_WRITE DEVICE (0): remaining = 2
[14764.152145] SLEEPY_WRITE DEVICE (0): remaining = 2

Apart from this you are required to make sanity checks that are mentioned in the assignment.

A Shady Character Driver

The provided files shady.c and shady.h provide another skeleton character device driver. This one is going to be a little bit evil. Its purpose is to eavesdrop on processes belonging to another user. You are being given this assignment not because I want you to become proficient at writing **kernel-mode rootkits (http://en.wikipedia.org/wiki/Rootkit)** but rather because doing this will let you write some interesting code and also because it illustrates just how much trust we must have in anyone who writes code that we load into our OS kernels!

The first job of the shady module is to hack the system call table so that any call to the open() system call is diverted to the shady module. This used to be really easy, but the Linux kernel developers have wised up and made it so that the location of the system call table is no longer provided to kernel modules. Additionally, they have used the virtual memory system to mark the system call table as read-only. You will have to bypass both of these protections.

You must find the location of the system call table on your machine. This is listed on a file in the /boot directory called System.map-xxxx where the xxxx is a kernel version. Make sure to use the system map for the kernel you are actually running! The uname -a command will tell you this. Now that you know the

address where the system call table lives, hard-code it in a global variable called system_call_table_address in shady.c. You have to do this so we can find it and change it to something different, if necessary, while grading your code.

Now, modify shady's __init function so that it turns off write protection on the system call table. You may use this handy function which goes ahead and directly modifies the appropriate page table entry:

```
void set_addr_rw (unsigned long addr) {
  unsigned int level;
  pte_t *pte = lookup_address(addr, &level);
  if (pte->pte &~ _PAGE_RW) pte->pte |= _PAGE_RW;
}
```

Now you are ready to intercept a system call. First, add this to your shady module:

```
asmlinkage int (*old_open) (const char*, int, int);

asmlinkage int my_open (const char* file, int flags, int mode)
{
   /* YOUR CODE HERE */
}
```

(Note: you can and probably must modify code outside of where the comments indicate your code should start/stop.) This code makes a location for storing the old value of the system call address and it also shows how to declare your new system call. Your next task is to save the current value of the "open" system call into old_open and replace it with the address of your my_open function. Once you do this, your code will be called any time any process on this VM opens a file! At this point, if you have a bug, your system will die right away, so tread carefully and save your work often. It is important that your my_open code calls the old_openfunction so that files can still be opened, otherwise the system will not work at all. Put a printk into your my_open code to make sure it is getting called. Make sure to restore the old system call when your module is unloaded. Do not worry about setting the page protections back how there were. You'll find useful information in the files unistd_64.h in:

/usr/include/x86_64-linux-gnu/asm/unistd_64.h

You now have one final job. We don't want to spy on everyone, but rather on a specific user. Create a new user on your system called "mark" and look in /etc/passwd to figure out what mark's userid is. Modify your system call interceptor so that it does nothing (other than call the actual code to open a file) unless it is mark who is opening the file. If mark opens a file, use printk to print a message such as:

```
mark is about to open '/etc/ld.so.cache'
```

You must put mark's UID into a global variable in shady.c called marks_uid so that we can change this UID for grading purposes. You can find useful functions for finding the current UID in this file:

/usr/src/linux-headers-3.13.0-45-generic/include/linux/cred.h

EXTRA CREDIT: Hide your shady device driver so that lsmod and /proc/modules do not list it when it is loaded.

Handin

Submit four files in Canvas: sleepy.c sleepy.h shady.c shady.h