**COMP 4108 Assignment 3**

Shane Bishop

Student #: 101030053

# Part A

**Question 1**

Question 1 has no marks associated with it.

**Question 2**

Question 2 has no marks associated with it.

**Question 3**

From <https://en.wikipedia.org/wiki/System.map#Filesystem_location> I learned that the System.map file can likely be found at /boot/System.map-$(uname -r). I then used fgrep to look for any lines in the file with the substring sys\_call\_table:



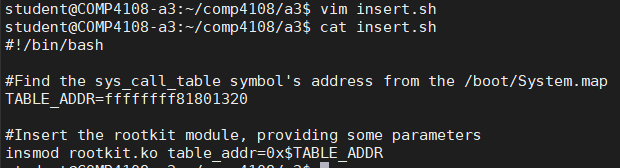
I already knew my system was x64, but I confirmed this with arch:



I knew my system architecture was not x32 or ia32, so by process of elimination, the address applicable to my architecture was ffffffff81801320.

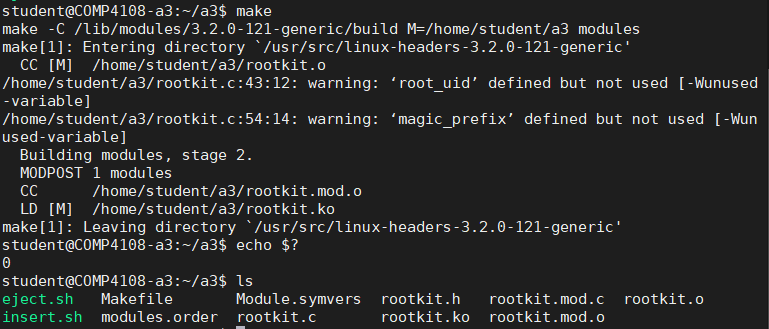
**Question 4**

I edited insert.sh with the address I found in question 5, as pictured in the screenshot below.



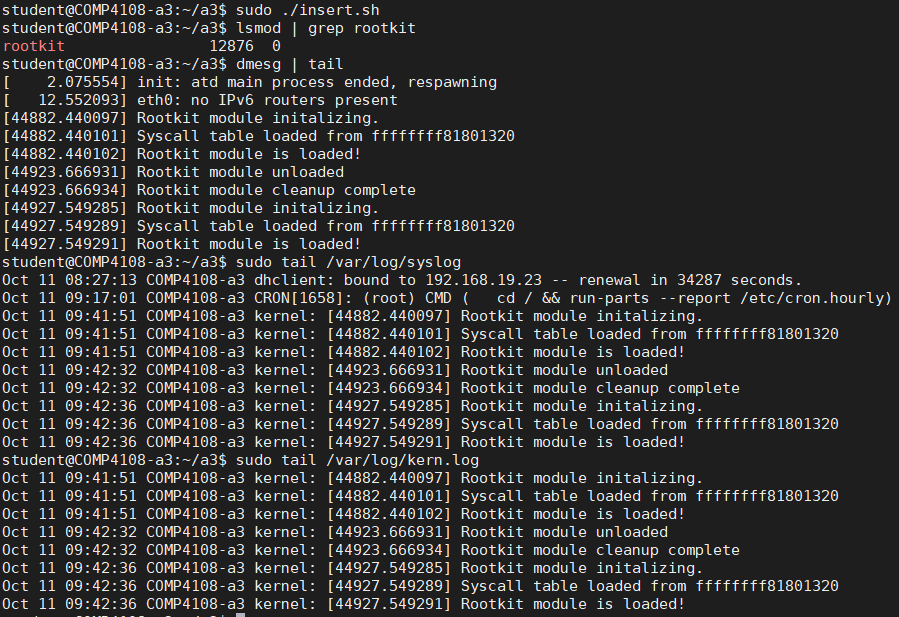
**Question 5**

I confirmed I could compile the rootkit framework, as shown below.



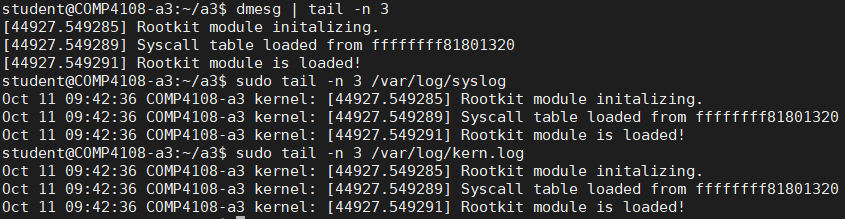
In confirmed the compilation was successful by both checking the exit code of make was 0, and checking (with ls) that new .o and .ko files were generated in my directory (I know from previous kernel hacking experience that I should expect these files to be built).

**Question 6**



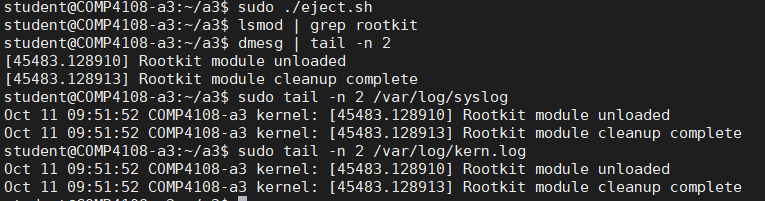
In the screenshot above, I first inserted the LKM with sudo ./insert.sh. I grep’d the output of lsmod to confirm the rootkit was loaded successfully (the man page for lsmod says it prints status info on all currently loaded kernel modules).

Kernel log messages can be accessed in three different ways: with the dmesg program, by reading /var/log/syslog, or by reading /var/log/kern.log. In the screenshot above, I tail each of them. From the three sets of output it can be seen that I inserted the module, ejected it, and reinserted it. The only lines of real interest are the last three lines of each, as those three lines are from when the module was inserted last:



The tail command, by default, prints the last 10 lines of a file. The -n option of tail can be used to specific a number of lines to print, rather than the default of 10 lines, as illustrated here – I use -n 3 to only print the last three lines of each source of kernel/system logs.

**Question 7**



I ejected the kernel module with sudo ./eject.sh. I then checked the rootkit was not loaded using lsmod and grep (grep has no output when there are no matches, so the empty output in the screenshot above means that the rootkit module is no longer loaded). Since I knew from my output for question 6 that the kernel module only prints two lines of output on removal, I used tail -n 2 to only get the last two lines of dmesg, /var/log/syslog, and /var/log/kern.log, as those are the only lines relevant to the removal of the kernel module.

**Question 8**

I went through the rootkit.c file and looked at all of the comments that said “NEEDED FOR PART A”. In each case, I inserted

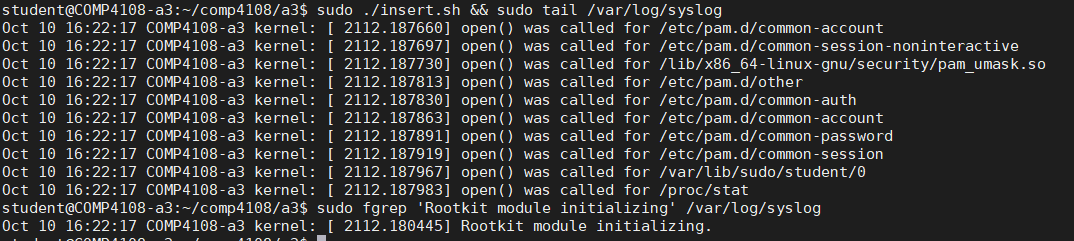
set\_addr\_rw((unsigned long) sys\_call\_table);

before the bit of code, and inserted

set\_addr\_ro((unsigned long) sys\_call\_table);

after the bit of code. As asked to do in the guiding comments, I make the sys\_call\_table address R/W (permitting both read and write), and then reset the permissions back to read-only after the operation. I didn’t need to do any more than that, since both set\_addr\_rw() and set\_addr\_ro() functions were already written in rootkit.c.

I then ran sudo ./insert.sh && sudo tail /var/log/syslog. The first part of the command was to insert the kernel module, and the second part was to see the output of the syslog to confirm that the open() hook was working correctly. Since there are over 100 entries of logs for open(), I fgrep’d /var/log/syslog to confirm that the rootkit module was loaded prior to the open() logs.



# Part B

Before giving detail on Question 1 and 2, I will give a high-level description of the problem and my solution.

The problem is to use my kernel module to allow my user (i.e., the student user) to be able to run any program with an effective UID of 0 (i.e., run any program as root). This can be achieved since kernel code has even more power and control than the root user, since it has direct access to all of the kernel’s data structures.

In order to achieve this, we can intercept all execve() system calls. Before passing the intercepted system call parameters, we can check the current effective UID matches the root\_uid parameter (this is a parameter passed to the kernel module when it is inserted with insmod). If the effective UID matches the root\_uid parameter, we can change the effective UID for the process to 0, and then run the execve() system call. This will execute the program with 0 is the effective UID.

**Question 1**

To do this question, I first looked at the execve(2) man page. Looking at the new\_open() function’s signature, I realized the new\_execve() function would be modelled of off that. Using the signature of execve() from the man page, I wrote a function prototype in rootkit.h, and then created an empty function in rootkit.c. I then copied the body of new\_open() into the empty body of new\_execve().

I knew from looking at /usr/src/linux-source-3.2.0/include/asm-generic/unistd.h that I could reference execve with \_\_NR\_execve, so I did that with the line

execve\_hook = find\_syscall\_hook(\_\_NR\_execve);

I could print the filename being execve’d simply with

printk(KERN\_INFO "Executing %s\n", filename);

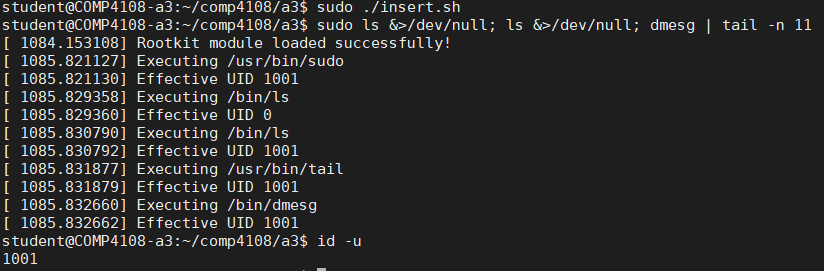
since filename was a parameter to new\_execve(). I then looked at /usr/src/linux-source-3.2.0/include/linux/cred.h and found that I could get the effective UID with the current\_euid() macro, so I added the line

printk(KERN\_INFO "Effective UID %d\n", current\_euid());

to my new\_exeve() function.

All of these modifications to rootkit.c can be found in the code directory in code/rootkit.c.

I then ran the following:



First, I inserted the kernel module (sudo ./insert.sh). Then I ran ls, both as root and as student (redirecting the output to /dev/null). I then used dmesg to look at the logs. Finally, I ran id -u to print the student user’s UID.

From the logs we can see the module was loaded successfully, and then sudo, ls, tail, and dmesg were all executed. We can see that every execve was run with the student user’s eUID (we know this from the output of id -u being 1001 and the eUIDs in the logs also being 1001) except for ls, which ran as root (eUID is 0). This is as expected:

* sudo ran with eUID 1001 because sudo needs to check if the user belongs to the sudo group (and get a password) before running the program it is given to run
* The first ls ran with eUID 0 because it was a child process of sudo (and sudo defaults to running as root if no other user is specified)
* The second ls, the tail, and the dmesg were all run by the student user

**Question 2**

I began by taking a look at [cred.h](https://elixir.bootlin.com/linux/v3.2/source/include/linux/cred.h) and [cred.c](https://elixir.bootlin.com/linux/v3.2/source/kernel/cred.c) in the Linux source code. I read the documentation for prepare\_creds(), and saw that the return value of prepare\_creds() needs to be passed to either commit\_creds() or abort\_creds(). prepare\_creds() takes no parameters and returns a struct cred\*, which is the parameter type both commit\_creds() and abort\_creds() accept. So I knew I would call prepare\_creds(), and then pass its return value to commit\_creds() if the return values is not NULL, else I would pass the return value to abort\_creds().

I then took a look at the commands in rootkit.c for the root\_uid paramater. I added the following two lines after the root\_uid declaration:

module\_param(root\_uid, int, 0);

MODULE\_PARM\_DESC(root\_uid, "UID to map to root");

I then modified insert.sh. I added the line

USERNAME=student

and then modified the last line to be

insmod rootkit.ko table\_addr=0x$TABLE\_ADDR root\_uid="$(id --user "$USERNAME")"

I did things this way so that the user to be given root privileges could be easily modifed in the future, if desired. (This allows for easier reusability of the rootkit.)

I then modified my new\_execve() function in rootkit.c. I added the following lines:

if (current\_euid() == root\_uid) {

struct cred \*new\_cred = prepare\_creds();

if (new\_cred != NULL) {

//Modify new\_cred to have an UID and eUID of 0

new\_cred->uid = 0;

new\_cred->euid = 0;

//Commit new\_cred

commit\_creds(new\_cred);

} else {

//prepare\_creds() returned NULL, so abort

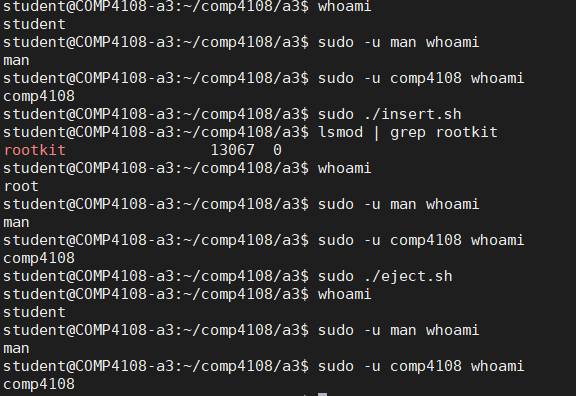
abort\_creds(new\_cred);

}

}

All of these code modifications can be found in the code directory in the respective files.

Finally, I ran the following:



I first run whoami as student, man, and comp4108, and see that for each user whoami displays student, man, and comp4108 respectively, as is normal.

Then I insert the module (sudo ./insert.sh) and then confirm the module is loaded (lsmod | grep rootkit).

Then I rerun the same three whoami commands I ran before inserting the module. whoami prints root for the student user (as the bash shell continues to run as the student user). This shows the module is correctly giving the student user root privileges. We see whoami continues to print man for the man user and comp4108 for the comp4108 user. This shows that the module is *only* giving root privileges to the student user, as desired.

Finally, I remove the module and rerun the three whoami commands again, to demonstrate that everything returns to normal once the module is removed.

# Part C

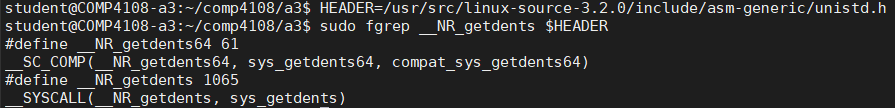
Before giving detail on Question 1 and 2, I will give a high-level description of the problem and my solution.

The problem is we want to find a way to hide files in the filesystem from any programs that use the legacy getdents() system call to look at dir-entries (in this case our target is specifically the ls program, but this is applicable to all programs that use getdents()).

The solution is to use a kernel module to hook all getdents() system calls. The system call will be intercepted by the kernel module’s code. The system call can be run as normal, but the data can be manipulated prior to the user space program seeing the data. In this way, it will make it invisible to the user space caller that one or more dir-entries were hidden.

**Question 1**

First I made the assumption that, following the same pattern as for open() and execve(), that the macro for getdents() would be \_\_NR\_getdents. I confirmed this was true:

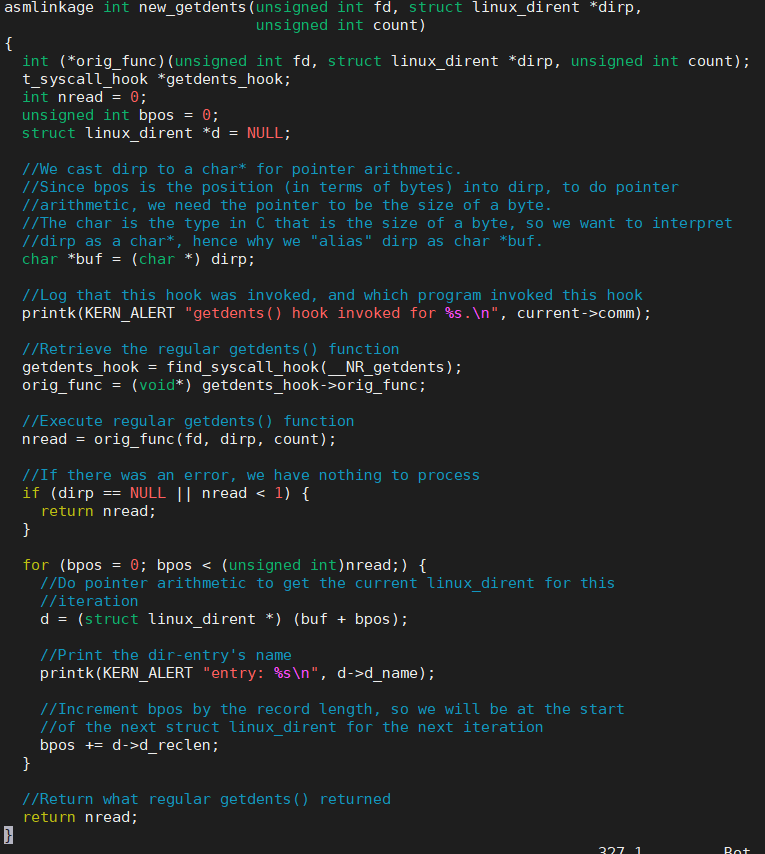


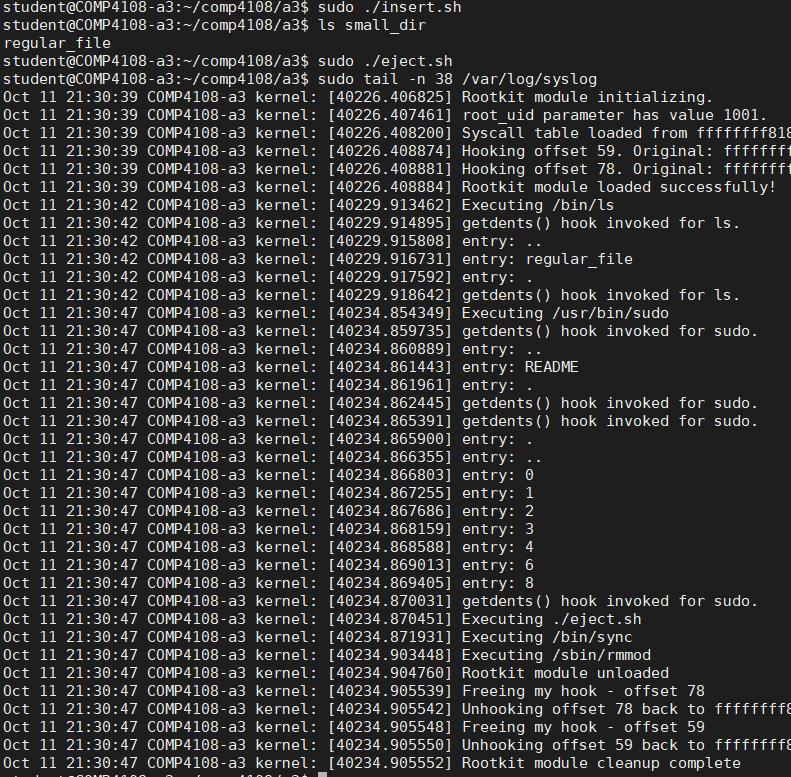
The first two lines of the output of fgrep are for getdents64(), which we aren’t interested in. The third line shows the define, and the fourth line “declares” the syscall (at least that’s my understanding of the fourth line). So my assumption was correct.

To see my C code for the version of new\_getdents() for this question, see code/partb\_question1/rootkit.c. I have this separate from code/rootkit.c, as the final version (for question 2) is significantly different from my original code for this question.

I then wrote my new\_getdents() syscall hook function based on the [getdents man page](https://man7.org/linux/man-pages/man2/getdents.2.html). I had a few false starts before I realized I needed to cast the dirp argument to getdents() as a char\* for pointer arithmetic. This is because the d\_reclen member of the struct linux\_dirent has the size of the record in bytes. To get the correct offset from dirp to the start of the each record, I needed to do pointer arithmetic on a char\* so that the pointer arithmetic would properly offset the correct number of bytes.

Here is a screenshot of my code for this question, including comments (this is also available in code/partb\_question1/rootkit.c):



I then tested my code to confirm everything worked: 

First, I inserted my module. Then I ran ls, so the getdents() would execute. Then I ejected the module. Finally, I tail’d /var/log/syslog to get the output for the module (I ran this a couple times to find the number of lines I needed to get with tail).

Let’s analyze the output in /var/log/syslog in the screenshot above.

* First, the module initializes.
* After that, we see ls ran, and that the three entries were “..”, “regular\_file”, and “.”.
* getdents() was invoked by ls again (this is because there may have been more dir-entries that hadn’t fit in ls’s buffer for the first call)
* getdents() is invoked by sudo four times. I’m not really sure why sudo was invoking getdents(), but it was. This would have happened with my sudo ./eject.sh command. (sudo only “effectively” invokes getdents() twice, but it does the same thing as ls, where it calls it again to get anything that didn’t fit in the buffer the first time, hence four calls total.)
* Then the module unloaded, as indicated by the last 6 lines of the screenshot above

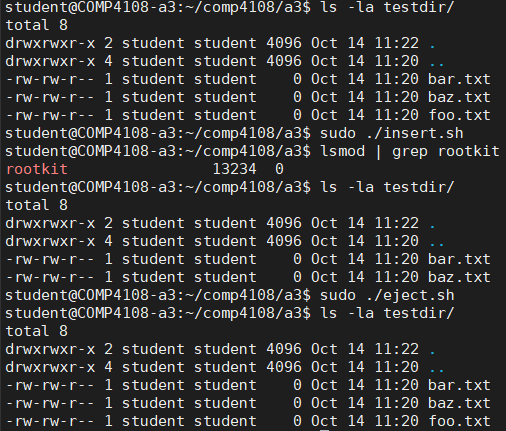
**Question 2**

The C code for the final version of my new\_getdents() function is rather complex, so I have provided detailed comments in code/rootkit.c. The code will not be explained here, as it is explained with the comments in the source code.

Other than my changes to new\_getdents(), I also updated insert.sh so that the magic\_prefix parameter is specified as "foo", and I updated the magic\_prefix variable in rootkit.c to take the parameter specified.

The goal for this question was to make all dir-entries invisible to ls (or any other program that uses the legacy getdents() system call) by hooking all getdents() system calls using the kernel module. This was done by calling the regular getdents() function from within the kernel (since it’s a kernel module), and then editing the dirp pointer to exclude all dir-entries a name prefixed by magic\_prefix. Finally, this dirp pointer would be available back in user space to the calling process, which would be unaware that the dirp pointer had ever been modified.

Here is a screenshot demonstrating that my kernel module successfully hides files with a specified prefix:



First I run ls -la testdir to show all directory entries normally visible with ls (-a includes “normal” hidden dir-entries that start with “.”) I then insert my kernel module with sudo ./insert.sh. I confirm the module is now inserted with lsmod | grep rootkit. I then run ls -la testdir again, and now I see foo.txt is not visible with ls, since I used “foo” as the magic prefix. Then when I remove my module and run ls again, foo.txt is visible once more.