Project #5: Rootkit

ECE 650 – Spring 2025

Due on 4/29 11:59 pm

General Instructions

1. You will work individually on this assignment.

2. The code for this assignment should be developed and tested using a Linux Virtual machine based on **Ubuntu 20.04** that you may create at the following link:
   https://vcm.duke.edu/
Select following image: Ubuntu 20.04 VM
Kernel version: Linux 5.4.0-104-generic (You can see your kernel version using "uname -r")
You can use "apt-cache search linux-image" to search the images and use "apt-get upgrade [image name]" to update into a specific kernel version

Please make sure you are using correct environment. Otherwise, your code is not likely to work when in our testing environments.

3. You must follow these assignment guidelines carefully, and turn in everything in the described formats. Otherwise, you will lose grades.

4. This assignment involves writing and testing a kernel module. If you have bugs in your module, **it is entirely possible to break your system**. Make sure to use the **snapshots feature** on the Duke VM to recover to a usable state. It is preferable to work on a system that does not impact any other work.

5. To be noted, there will be no pretest for this project.

Overview

In this assignment, you will implement a portion of Rootkit functionality to gain:

1. Hands-on practice with kernel programming
2. A detailed understanding of the operation of system calls within the kernel
3. Practice with fork/exec to launch child processes
4. An understanding of the types of malicious activities that attackers may attempt against a system (particularly against privileged systems programs).

Our assumption will be that via a successful exploit of a vulnerability, you have gained the ability to execute privileged code in the system. Your "attack" code will be represented by a small program that you will write, which will (among a few other things, described below) load a kernel module that will conceal the presence of your attack program as well as some of its malicious activities. The specific functionality required of the attack program and kernel module (as well as helpful hints about implementing this functionality) are described next.

Tips on Working with the Virtual Machine

When you create your virtual machine and log-in for the first time, you will notice there may be few programs installed (e.g. no gcc, emacs, vim, etc.). You can download your choice of software easily using the command: sudo apt-get install <package name>. For example:

```
        sudo apt install build-essential emacs
```

Detailed Submission Instructions

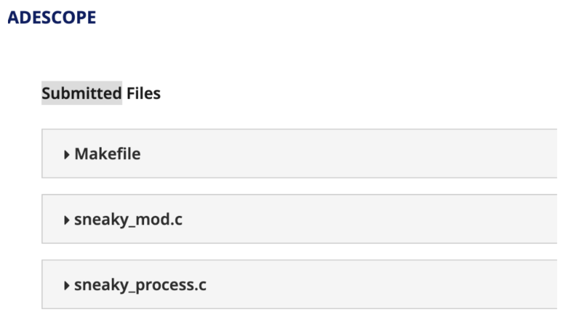
Your submission will include 3 (and only 3) files:

1. **sneaky_mod.c** – The source code for your sneaky module with functionality as described below.
2. **sneaky_process.c** – The source code for your sneaky (attack) program with functionality as described below.
3. **Makefile** – A makefile that will compile "sneaky_process.c" into "sneaky_process", and will compile "sneaky_mod.c" into "sneaky_mod.ko". In most cases, this will simply be the example Makefile provided with the skeleton module example code.

You will submit a single zip file named **"proj5_netid.zip"** to gradescope, e.g.:

```
        zip proj5_netid.zip sneaky_mod.c sneaky_process.c Makefile
```

Submission format:


ADESCOPE

Submitted Files

- Makefile
- sneaky_mod.c
- sneaky_process.c

After you submit project 4, you will need to give a demo to the TA. The tentative demo date is set for April 30. However, if you complete the assignment before April 15 and demo it to the TA during office hours, you will receive 10 points of extra credit.

Attack Program

Your attack program (named sneaky_process.c) will give you practice with executing system calls by calling relevant APIs (for process creation, file I/O, and receiving keyboard input from standard input) from a user program. Your program should operate in the following steps if you run `sudo ./sneaky_process`:

1. Your program should print its own process ID to the screen, with exactly following message (the print command in your code may vary, but the printed text should match):
`printf("sneaky_process pid = %d\n", getpid());`
2. Your program will perform 1 malicious act. It will copy the /etc/passwd file (used for user authentication) to a new file: /tmp/passwd. Then it will open the /etc/passwd file and print a new line to the end of the file that contains a username and password that may allow a desired user to authenticate to the system. Note that this won't actually allow you to authenticate to the system as the 'sneakyuser', but this step illustrates a type of subversive behavior that attackers may utilize. This line added to the password file should be exactly the following:
`sneakyuser:abc123:2000:2000:sneakyuser:/root:bash`
3. Your program will load the sneaky module (sneaky_mod.ko) using the "insmod" command. Note that when loading the module, your sneaky program will also pass its process ID into the module. You may reference the following page for an understanding of how to pass arguments to a kernel module upon loading it:
http://www.tldp.org/LDP/lkmpg/2.6/html/x323.html
4. Your program will then enter a loop, reading a character at a time from the keyboard input until it receives the character 'q' (for quit, you don't need to hit carriage return or other keys after 'q'). Then the program will exit this waiting loop. Note this step is here so that you will have a chance to interact with the system while: 1) your sneaky process is running, and 2) the sneaky kernel module is loaded. **This is the point when the malicious behavior will be tested in another terminal on the same system while the process is running.**
5. Your program will unload the sneaky kernel module using the "rmmod" command.
6. Your program will restore the /etc/passwd file (and remove the addition of "sneakyuser" authentication information) by copying /tmp/passwd to /etc/passwd.

Recall that a process can execute a new program by: 1) using fork() to create a child process and 2) the child process can use some flavor of the exec*() system call to execute a new program. You will want your parent attack process to wait on the new child process (e.g. using the waitpid(…) call) after each fork() of a child. Remember **system()** is a wrapper around fork() and execve().

Sneaky Kernel Module (a Linux Kernel Module – LKM)

Your sneaky kernel module will implement the following subversive actions:

1. It will hide the "sneaky_process" executable file from both the 'ls' and 'find' UNIX commands. For example, if your executable file named "sneaky_process" is located in /home/userid/hw5:
   a. "`ls /home/userid/hw5`" should show all files in that directory except for "sneaky_process".
   b. "`cd /home/userid/hw5; ls`" should show all files in that directory except for "sneaky_process"
   c. "`find /home/userid -name sneaky_process`" should not return any results
2. In a UNIX environment, every executing process will have a directory under /proc that is named with its process ID (e.g /proc/1480). This directory contains many details about the process. Your sneaky kernel module will hide the /proc/<sneaky_process_id> directory (note hiding a directory with a particular name is equivalent to hiding a file!). For example, if your sneaky_process is assigned process ID of 500, then:
   a. "`ls /proc`" should not show a sub-directory with the name "500"
   b. "`ps -a -u <your_user_id>`" should not show an entry for process 500 named "sneaky_process" (since the 'ps' command looks at the /proc directory to examine all executing processes).
3. It will hide the modifications to the /etc/passwd file that the sneaky_process made. It will do this by opening the saved "/tmp/passwd" when a request to open the "/etc/passwd" is seen. For example:
   a. "`cat /etc/passwd`" should return contents of the original password file without the modifications the sneaky process made to /etc/passwd.
4. It will hide the fact that the sneaky_module itself is an installed kernel module. The list of active kernel modules is stored in the /proc/modules file. Thus, when the contents of that file are read, the sneaky_module will remove the contents of the line for "sneaky_mod" from the buffer of read data being returned. For example:
   a. "`lsmod`" should return a listing of all modules except for the "sneaky_mod"

Your overall submission will be tested by compiling your kernel module and sneaky process, running the sneaky process, and then executing commands as described above(hint: in another terminal) to make sure your module is performing the intended subversive actions.

Helpful Hints and Tips for Implementing sneaky_mod.c

- This assignment should not require a tremendous amount of code. For example, in my sample solution, the sneaky_mod.c file has approximately 200 lines, the sneaky_process.c is much shorter.

- You can inspect the system calls that are made by a command using the “strace” UNIX command, e.g. “strace ls”.
- You may find this link helpful if you are confused about “pt_regs” struct.
  https://elixir.bootlin.com/linux/v5.4/source/arch/x86/include/uapi/asm/ptrace.h#L44

- You need to get the parameter out of the “pt_regs” struct. The parameter passing convention for a system call is: param#0 to param#5 are respectively passed into the **DI**, **SI**, **DX**, **R10**, **R8** and **R9** registers. You can find the definition for pt_regs here: https://elixir.bootlin.com/linux/v5.4/source/arch/x86/include/asm/ptrace.h#L56

- For these subversive actions in the sneaky kernel module, you will need to hijack (and possibly modify the contents being returned by) system calls.

- For #1 and #2, I would highly recommend reading the ‘man getdents64’ page (including code sample). It will fill in an array of “struct linux_dirent64” objects, one for each file or directory found within a directory.

- For #2, you can know the “sneaky_process” pid by using the module_param(…) technique described here: http://www.tldp.org/LDP/lkmpg/2.6/html/x323.html

- For #3, you should check out the “openat” system call (as in the skeleton kermel module posted). Note that if, say, you wanted to pass a new string filename to the open system call function, that string has to be in “user space” and not something defined in your kernel space module. You can use the “copy_to_user(…)” function to achieve that:
  `copy_to_user(void __user *to, const void *from, unsigned long nbytes)`
  Hint, for the user buffer, could you use the character buffer passed into the openat(…) call?

- For #4, you may want to check out the “read” system call.

- You may also need to understand enable_page_rw & disable_page_rw and sys_call_table used in the program.

- You may find functions such as memmove useful in “deleting” information from a data structure, i.e. copy all the contents after the entry you want to delete to the position of the start of the data to delete.

- If there are pieces of the skeleton module code that you are interested to understand more deeply, please ask on ED! We’d be glad to give detailed descriptions.

- If you would like to allocate and free memory in kernel space please checkout `kvzalloc` and `kvfree`.

- If you would like to copy something from user space to kernel space, please checkout `copy_from_user`.

- Have fun with this assignment! Try out other sneaky actions, if you'd like, once you get the hang of it.

Extra Credit

If you submit the assignment before **April 15** and **demo it to the TA** during office hours, you will receive **10 points** of extra credit.

FAQ

1. Q: my sneaky_process's functionality is correct, but every time when I type "q" and exit the process, the vcm will get stuck and then require reloading the window (I use vscode). In my previous version of sneaky_process, during testing in another terminal, it gets stuck and then requires reloading the window after a period of time after running the process. Does it run out of CPU or memory?
   A: No, there are some errors inside your code. For example, if you modify the user-mode data structure inside kernel mode program, it may cause system error when process exit, although it finishes running. And you will also lose points because of system error.

2. Q: My sneaky_mod changes the read function, and the result from lsmod while the module is loaded is identical to the result from lsmod before it is loaded. However, I noticed if I run the cat command on a file including the word "sneaky_mod" that line is also removed. Is this behavior acceptable for this project?
   A: No, if additional info is removed by the program, you will lose some points. Because it is easy to detect system is attacked.

3. Q: I want to ask how we can test our project5. Just run the sneaky_process. Then run these instructions in Sneaky Kernel Module (a Linux Kernel Module – LKM). Such as ls , find and so on. And check whether these actions show like the pdf said?
   A: Yes. Use `gcc` to compile your `sneaky_process`.OR modify Makefile to add `sneaky_process` into `all`

4. Q: Before running sudo, "ls" shows "sneaky_process", when running sudo, "ls" doesn't show "sneaky_process", after running, "ls" still cannot show "sneaky_process". But after restarting the virtual machine, "ls" can show sneaky_process again. How can I solve this problem?
   A: Please make sure your sneaky_process removes the sneaky_mod when it finishes running. It seems to me that you did not 'clean things up' when exiting the sneaky_process so the malicious behavior is still in effect. Then the restart removed it for you so everything seems normal again.