Assignment #5: Rootkit

ECE 650 – Spring 2020

See course site for due date

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 that you may create at the following location:
   https://vm-manage.oit.duke.edu/vm_manage
   Select following image: Ubuntu 16.04
   Other environments, unfortunately due to complexity, will not be supported.

3. You must follow this assignment spec carefully, and turn in everything that is asked (and in the proper formats, as described). Due to the large class size, this is required to make grading more efficient.

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

   Tl; Dr: You may brick your system! Proceed with caution. Use isolated VM and/or VM snapshots.

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
```

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:

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). 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.

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 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_process.c file has approximately 120 lines of code, and the sneaky_mod.c file has approximately 200 lines.

- You can inspect the system calls that are made by a command using the "strace" UNIX command, e.g. "strace ls".

- 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, read up on the "getdents" system call (get directory entries): `int getdents(unsigned int fd, struct linux_dirent *dirp, unsigned int count)`. I would highly recommend reading the 'man getdents' page (including code sample). It will fill in an array of "struct linux_dirent" objects, one for each file or directory found within a directory. You should also place the following struct definition at the top of your sneaky_mod.c code to make sure that the "struct linux_dirent" is interpreted correctly:

```
struct linux_dirent {
  u64            d_ino;
  s64            d_off;
  unsigned short d_reclen;
  char           d_name[BUFFLEN];
};
```

    - 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 "open" 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 open(…) call?

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

- You may also need to modify the function pointers to pages_rw, pages_ro and sys_call_table used in the program. This information can be extracted from your system using the following command (already mentioned in the sneaky_mod.c file comment):

```
sudo cat /boot/System.map-*-*-generic | grep -e set_pages_rw -e
set_pages_ro -e sys_call_table
```

Sample output:

```
> sudo cat /boot/System.map-*-*-generic | grep -e set_pages_rw
-e set_pages_ro -e sys_call_table
ffffffff8107ba00 T set_pages_ro
ffffffff8107ba70 T set_pages_rw
ffffffff81e00240 R sys_call_table
ffffffff81e01600 R ia32_sys_call_table
ffffffff8107ba00 T set_pages_ro
ffffffff8107ba70 T set_pages_rw
ffffffff81e00240 R sys_call_table
ffffffff81e01600 R ia32_sys_call_table
```

The first three from above (ignoring ia32) can be used to set the pointers in the program.

- You may find functions such as memcpy 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 and return the updated size (excluding the deleted entry).
- If there are pieces of the skeleton module code that you are interested to understand more deeply, please ask on piazza! We’d be glad to give detailed descriptions.
- Have fun with this assignment! Try out other sneaky actions, if you’d like, once you get the hang of it.