CSC 347

A2: Report

Michael De Lisio, Arshia, Gharai

28th October, 2019

Part A:

Part a:

Commands used [Transcript]:

- strace ./vuln_slow 3 "message" => This command was used to trace through the code to find out the output file of program which came out to be "/home/inoroot/.debug_log"
    - Output = The whole strace output its too long to include it in here :)

- ./vuln_slow 60 "message sent" => Then we run the program with the arguments 60 and “message sent” respectively. This gives us just the right amount of time to be able to change the output file of the program using the next two commands.
    - Output =

```
Sleeping for 180 seconds.
Writing debug message.
Goodbye!
```

- rm /home/inoroot/.debug_log => This command is used while the program is running so we delete the original output file.
    - Output = not output just removes the debug_log file

- ln -s /A2/Racing/Slow/root_file /home/inoroot/.debug_log => After deleting the original output file, we replace it with a symbolic link to root_file which is out desired destination.
    - Output = Again, not output, just links the two files

Contents of the root_file after the execution:

```
#
# This file is owned by root!
# You can test your ability to exploit a race condition by attempting to
# insert lines into this file. Note: You should *NOT* chown/chmod this
# file. You *MUST* insert lines into it using the setuid program 'vuln'
# included in the same directory
#
#
# Your Lines Should Follow Here:
message sent
```

Explanation of how the attack works:

So basically first since we have the control to set the delay of the program to any delay that we want, we make sure the delay is enough for our purpose. Then we run the program and when the program is delaying, it has already opened the original output file .debug_log. we now have to remove the original output file .debug_log using the ‘rm’ command and next, use the ‘ln’ command to link (symbolic link) the root_file to.debug_log file before the delay ends in a different terminal window. The reason that this method works is that since we link or change the output file of the program using a symbolic link after the original output file was opened by the program, the program won’t be able to check the permission of the root_file even though the user does not have permission to access the root_file file which needs root access. In regards to the **access()** system call, it is used by programs to find out whether or not the user has access to a file without the additional privileges of the setuid.

Part b:

Commands used [Transcript]:

Terminal 1:

```
>>> inoroot@csc347-a2:/A2/Racing/Fast$ ./vuln.sh
Writing debug message.
Goodbye!
Writing debug message.
Goodbye!
Invoking user does not have access to log file. DENIED.
Invoking user does not have access to log file. DENIED.
Invoking user does not have access to log file. DENIED.
Invoking user does not have access to log file. DENIED.
Writing debug message.
Goodbye!
Invoking user does not have access to log file. DENIED.
Invoking user does not have access to log file. DENIED.
^C
```

Terminal 2:

```
>>> inoroot@csc347-a2:/A2/Racing/Fast$ ./exploit.sh
ln: failed to create symbolic link `/home/inoroot/.debug_log': File exists
ln: failed to create symbolic link `/home/inoroot/.debug_log': File exists
ln: failed to create symbolic link `/home/inoroot/.debug_log': File exists
ln: failed to create symbolic link `/home/inoroot/.debug_log': File exists
^C
```

Checking to see if we have root privileges:

```
>>> inoroot@csc347-a2:~$ sudo whoami
>>> [sudo] password for inoroot:
Root
>>> inoroot@csc347-a2:/A2/Racing/Fast$ sudo -i
>>> root@csc347-a2:~# ls
rhosts
```

Explanation of how the attack works:

So in this attack, we have two different programs running at the same time to try to get root privileges. vuln.sh first removes the old output file and then runs vuln_fast with a high nice value. Running a program with a high nice value means the program gets low priority in the CPU so our other program exploit.sh which does the actual symbolic linking gets to finish first while vuln_fast is still running. When all the programs have stopped running, we have successfully changed the output file of vuln_fast to /etc/sudoers using a symbolic link.'

Part B

To find the system call table address we needed to read the System.map file in the boot directory. To read the System.map file you must have root privileges we had to use sudo, specifically: `sudo cat /boot/System.map-$(uname -r)` t, where uname -r gets the current release version of the OS in use. We used grep to find the syscall table symbol in the file and then used the cut demand with delimiter equal to ‘ ‘ to split the resulting output into an array. Here is the final command used, which only uses privilege escalation when necessary: `cut -d ' ' -f1 <<< $(sudo cat /boot/System.map-$(uname -r) | grep " sys_call_table").`

To load the kernel module we needed to first run make, then use sudo when running the bash insert.sh since you need root privileges to load and unload any kernel module.

To hook the open syscall all we needed to do was make the syscall table writable, by modifying the permission bits of the page storing the syscall function pointers. We called set_addr_rw first to make the table writable and then set_addr_ro after the hook was completed. Also, we called hook_syscall to swap the function pointer for the open syscall, specifically the line: `hook_syscall(new_hook(__NR_open, (void*) &new_open)).` This all done inside the init_module function.

Part C

The most challenging part of part C was figuring out the correct procedure for modifying the task_structs cred attribute. The cred structure houses the important data involving a task's (processes) privileges and identification. The mod_euid function's purpose is to modify the cred->euid attribute that is read only. Therefore, the copy and replace technique was used. To allocate a new copy of the current cred struct that was writable was done using the prepare_cred function. The prepare_cred functions acquires a cred struct mutex as well as returning a pointer to the cred struct. Once we'ved obtained a copy of the current cred struct we can then modify it and call commit_creds which replaces the old cred struct with the new struct we allocated and also releases the lock held by the prepare_cred function.

To hook the system call umask, we followed the same procedure as we did in Part B; first making the syscall table writable and then calling `hook_syscall(new_hook(__NR_umask, (void*) &new_umask)).` New umask logged a KERN_INFO message to the syslog every time the umask syscall was invoked. To generate the message we used `printk(KERN_INFO "umask(%d) invoked by PID %d (EUID %d)", mode, current->pid, current_cred()->euid),` using the global task_struct current variable to get the pid and the current_cred() wrapper function to acquire a reference to the cred struct for the effective uid.

For privilege escalation we provided the module with a parameter called root_uid, which was effectively the parameter that new_umask used to check whether the user invoking umask with argument 31337 had a specific effective uid. We set the root_uid equal to the student users euid with the bash command: id -u student (and not echo $EUID - since, the insert.sh file was run with sudo privileges and this would have given us the root euid instead of the student euid) in the insert.sh file.

Example Without Privilege Escalation:

>>> echo “EUID = $(id -u student)

1001

>>> umask 0 &

[1] 501

>>> dmesg | tail -n 1

[4626.957470] umask(0) invoked by PID 501 (EUID 1001)

Example With Privilege Escalation:

>>> echo “Student EUID = $(id -u student)”

Student EUID = 1001

>>> echo “Root EUID = $(sudo echo $EUID)”

Root EUID = 0

>>> umask 31337 &

[1] 488

>>> dmesg | tail -n 2

[4626.957470] umask(31337) invoked by PID 488 (EUID 1001)

[4626.957471] PID 488 elevated to EUID 0

When running the ihazroot binary, with the kernel module installed, the user is given access to a root sub shell. Ihazroot calls syscall(__NR_umask, 31337) (where __NR_umask is 95 on the lab machines) which elevates the privileges for the process with ihazroot’s pid and then prompts the user to write to STDIN just like a regular bash shell would (the prompt is “Sub shell: ”) and then processes the string and passes that string to the C library function system(). To exit the sub shell session the user types “exit”. All user input is appended and prepended with braces before being passed to the library function system so that the commands are run in a bash sub shell.

Example Running Ihazroot.c:
>>> id -u
1001
>>> ./ihazroot
Sub shell: id -u
0
Sub shell: exit
Ending sub session . . .

Running id -u shows the change in euid just by running the ihazroot binary.
Note: this example can be found also in a2/part_c/demo_transcript.txt

Part D

Question 1:

Using strace on ls, we can observe that ls uses the system call **getdents()**. Therefore the System call that we have to hook is **getdents()**.
The bash commands yields:

>>> strace -o ./file ls -a
. ..
>>> cat ./file | grep “write(1” -C 6 // shows system calls invoked just before
gentdents(3, /* 2 entries */, 32768) = 456 // ls -a writes to fd = 1 (STDOUT)
getdents(3, /* 0 entries */, 32768) = 0
close(3) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f267f8c8000
write(1, "347REPO file grep group_0191 "..., 45) = 45
close(1)
...

After reading man 2 getdents it is evident that the getdents syscall returns a pointer to a linux dirent struct (basically a linked list of directory entries). To hide certain files from a users vision we could modify the return value of getdents by removing any files with a certain naming convention from being added to linux struct dirent. Since the kernel module is read only, we have to change it so we are able to write to it. To do that, we can use set_addr_rw which allows us to make the system call table writable. Now that the system call table is writable, we can hook the system call getdents() using: `hook_syscall(new_hook(__NR_getdents, (void*) &new_getdents)),` which would represent the hook for getdents.

The only thing left to do is to change the return value of the getdents() system call which is invoked by ls and other functions that access the filesystem. Observing the linux dirent definition in man 2 getdents page, it becomes evident that d_off attribute will be of use. We could use the unsigned long d_off value to increment the pointer returned by getdents to traverse the linked dirent struct while checking the d_name attribute (file name) to see if we want to remove the entry from the linked structure. Then return the modified struct as the original getdents system call would.

Question 2:

If the system administrator suspected some unauthorized activity, what they might do to find hidden files or rootkits on the system is to use RKHunter or chkrootkit. These are specific rootkit detection tools in linux which help find hidden files or rootkits.
Also, if a file has been hidden by a simple mv command for example `mv text.txt .text.txt`, you can view these hidden files using the command `ls -a` or `ls -al` .