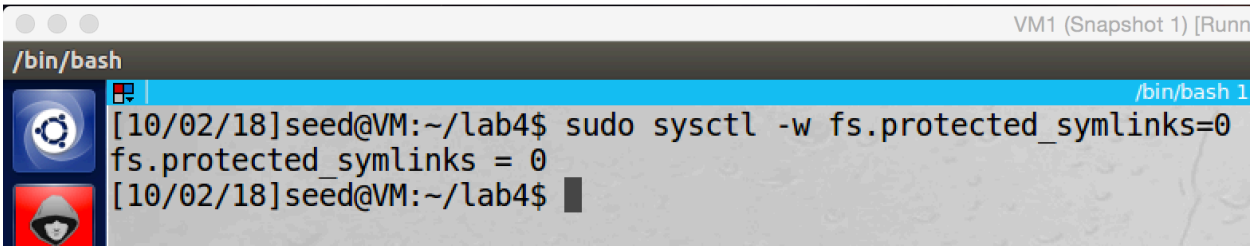
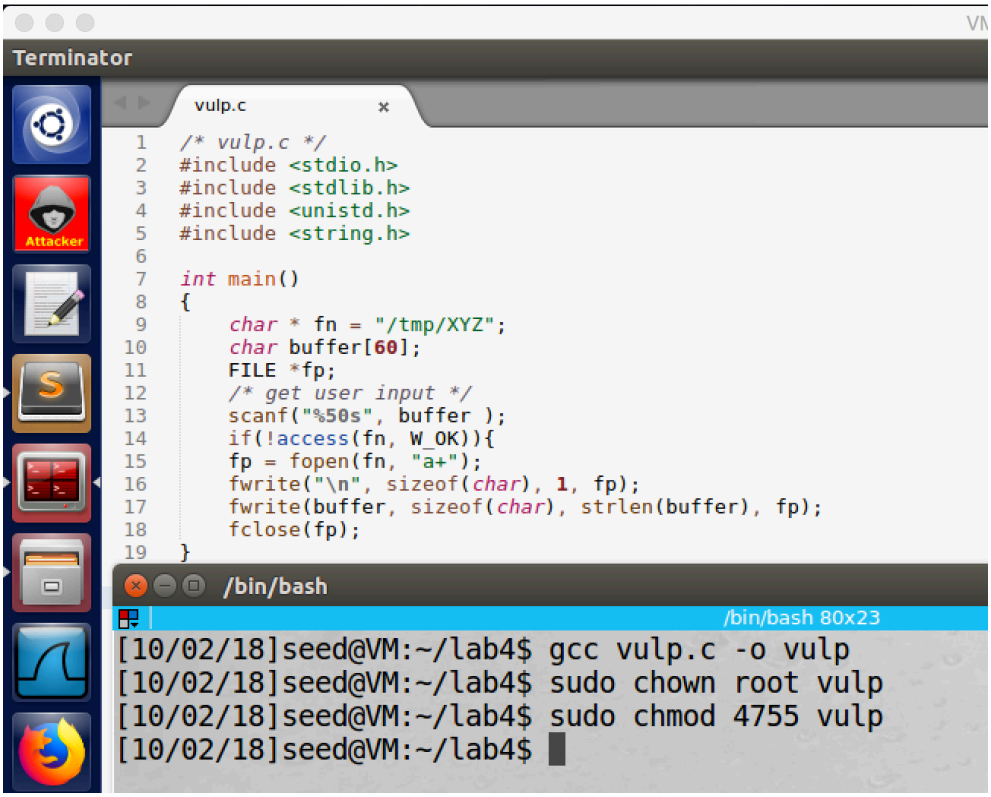
CSE643 Lab4
Yishi Lu
10/04/2018

Initial Setup

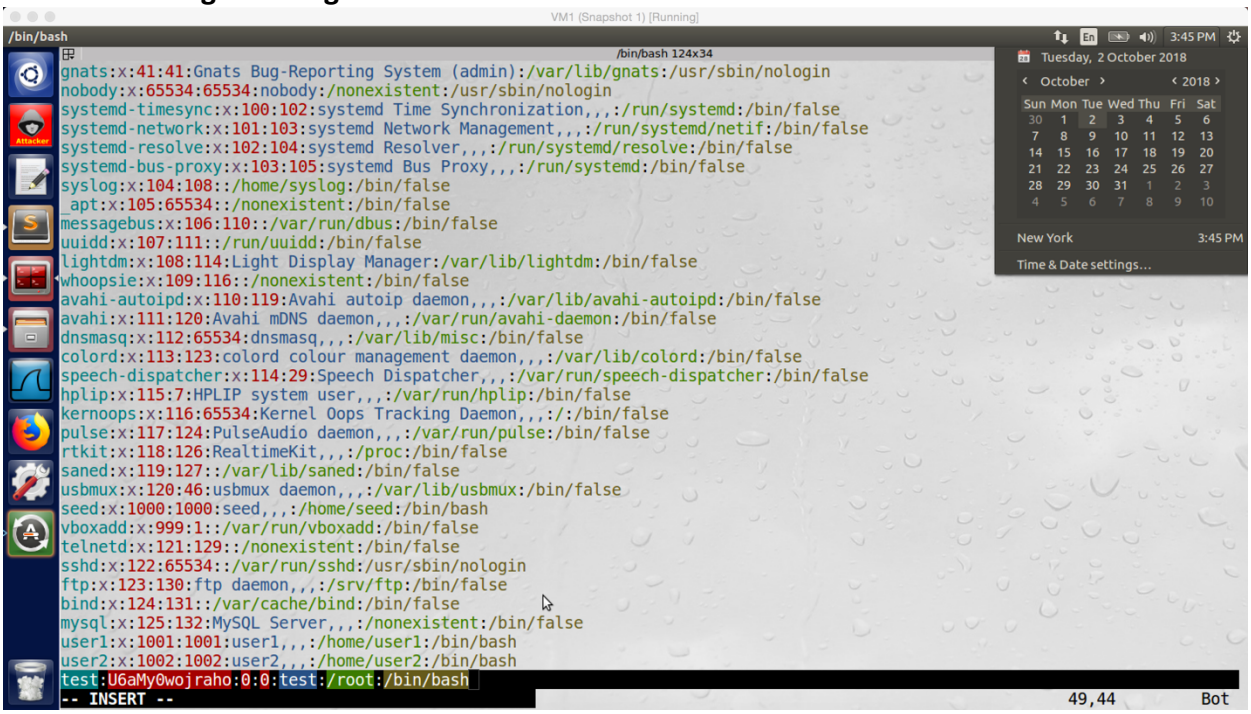


we close sticky symbolic links by setting its value to 0.

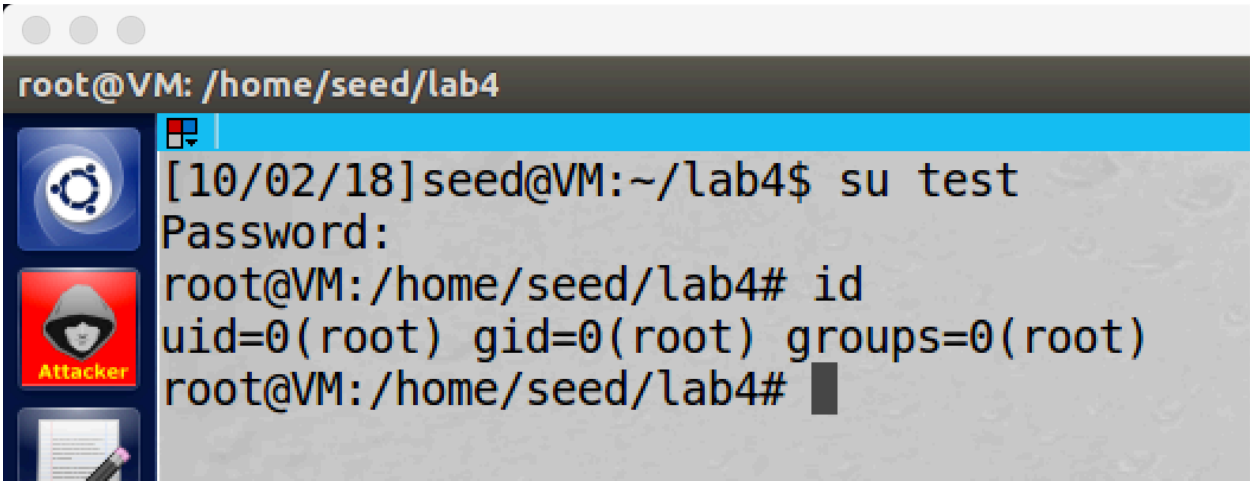


we get the vulnerable program from lab description, and then we compile it and make it Set-UID root program.

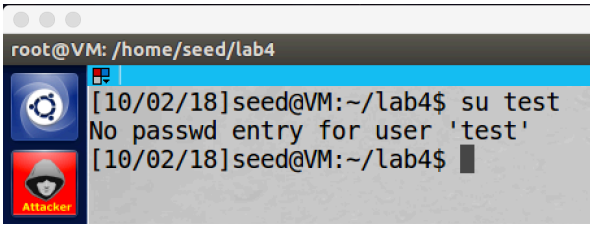
Task1: Choosing Our Target



screenshot1, we added the test account with the magic password into the /etc/passwd file



screenshot2, we switch to the test account, and we do not need to enter any character, we just press on enter, and we login the test account. Then we check the user id of the test account, it is a root account.

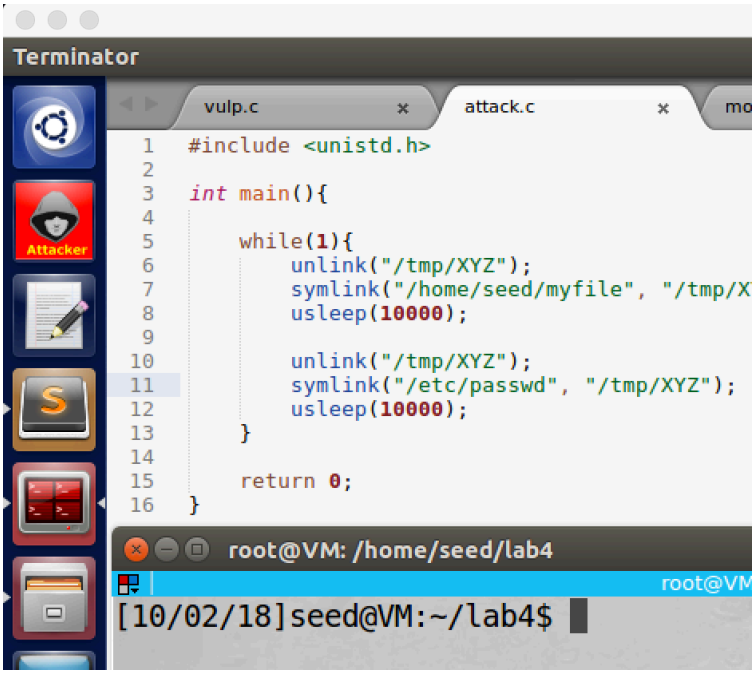


screenshot3, after we finish the test, we delete the test account

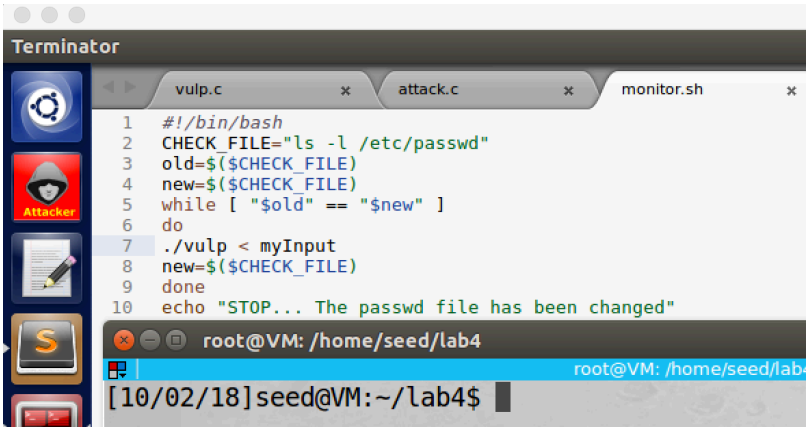
Observation and Explanation:

In this task, we test the magic password. We first open the /etc/passwd file, and we add an entry in the end, this entry defines a user account. The account has user name test, password slot is set to be the magic password, and the privilege slot is set to be 0 which means this is a root account (screenshot1). Then we switch to test account, and we just press enter, we login to the account. We also check the privilege of the account, it's a root account (screenshot2).

Task2: Launch the Race Condition Attack

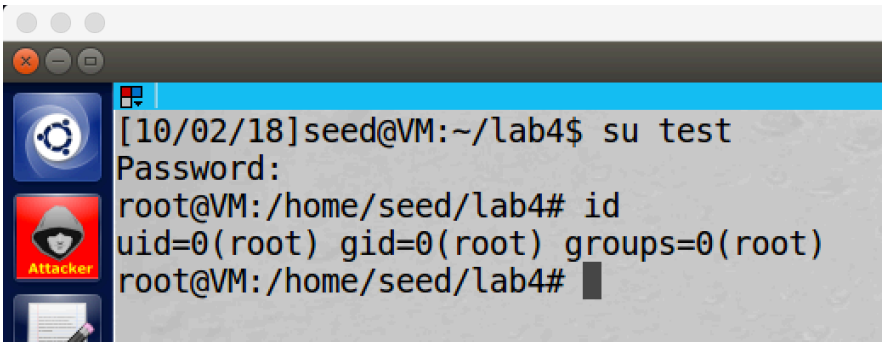


screenshot1, we create an attack program. When the program run, it keeps to change symbolic link of /temp/XYZ to myfile (created by us) and to our target file /etc/passwd.



screenshot2, we run a script to keep run the vulp program. Once our attack, the script will stop. And myInput is a file which contains the string “test:U6aMy0wojraho:0:0:test:/root:/bin/bash”

screenshot3, we run the attack program in another terminal. And then we run the script monitor.sh, after we run the program a while, it stops which means the /etc/passwd is modified.



screesnhot4, we switch to the test account, and it is a root account. So our attack succeeds.

Observation and Explanation:

In this task, we perform the race condition attack. Race condition is very common in operating system or software. It basically means concurrent execution of two processes or threads access a shared resource may have different results because the sequence or timing of the processes or threads. There are several types of race condition, here we focus on TOCTTOU, which occurs when checking for a condition before using a recourse.

In the vulnerable program, there is a function access(), this function checks the real user ID of the process whether it has write permission to write on the target file. In our attack, we want to open and write to /etc/passwd, but this file is created by root. So when access() called, it will prevent us to open and write to the file (our real user ID is not root, so we cannot pass the verification of access()). Therefore, we can use the vulnerability of race condition of the program. Before we perform attack, we need another program. In my case, it called attack.c (screenshot1)

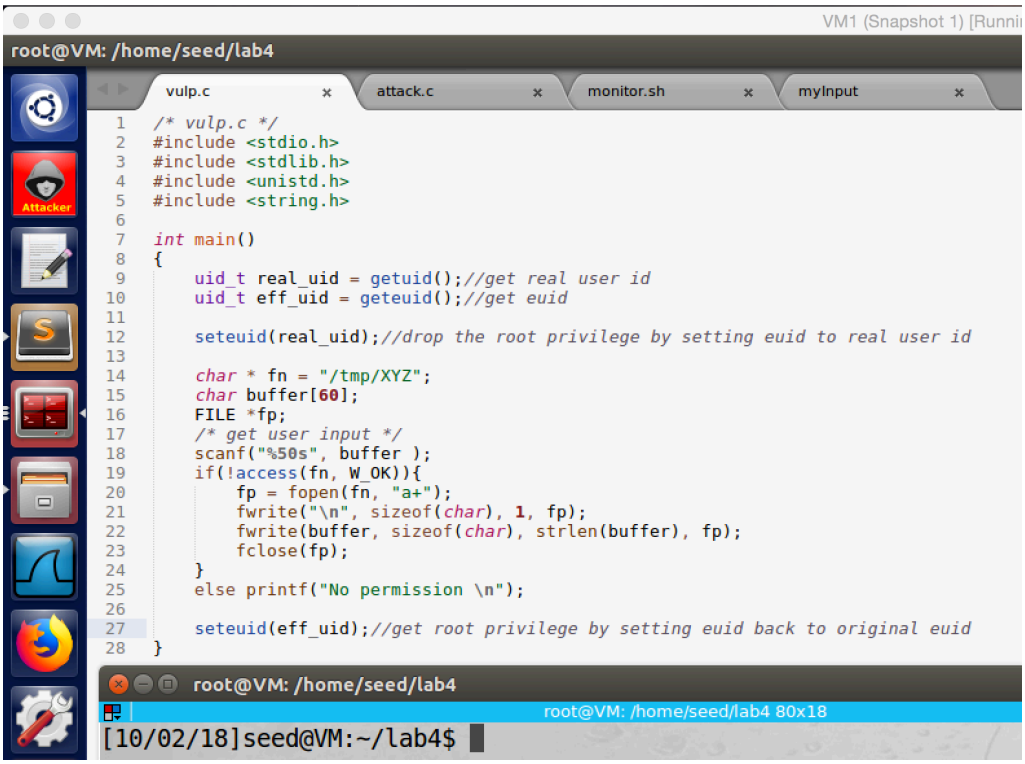
```
#include <unistd.h>

int main(){

        while(1){
                unlink("/tmp/XYZ");
                symlink("/home/seed/myfile", "/tmp/XYZ");
                usleep(10000);

                unlink("/tmp/XYZ");
                symlink("/etc/passwd", "/tmp/XYZ");
                usleep(10000);
        }

        return 0;
}
```

As the above program shows, it keeps to change the symbolic link of /tmp/XYZ. We first point the symlink to myfile, and then unlink it. Then we point /temp/XYZ to /etc/passwd, and then unlink it as well. It keeps doing such thing forever. “myfile” is created by us, so when access() is called to check, if the target file is “myfile”, then the verification will pass. So our goal is that we first point the symbolic link of /tmp/XYZ to myfile, so we can pass the verification of access(). Because we are not interesting on open and modify “myfile”, we want to open and write on

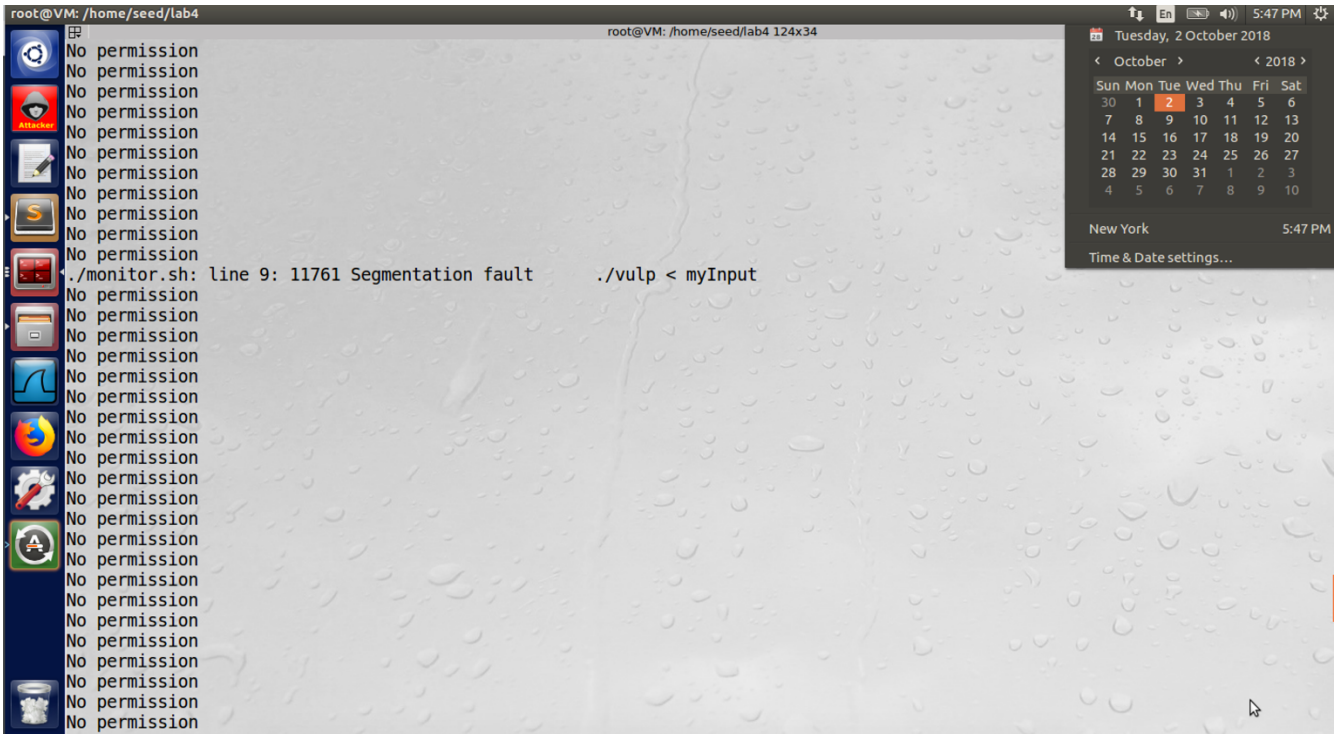
/etc/passwd. So after we pass the verification of access(), we change the symbolic link of /tmp/XYZ to /etc/passwd. Then the open() will open the file /etc/passwd (the open() only check the euid. And the vulnerable program is a Set-UID root program, so there is no any problem). Because we cannot control the timing, we need to run both program in many times; if we are lucky, we will succeed at last. To automatically run the vulnerable program, we create a script, which will run the program until /etc/passwd is modified (screenshot2).

Now we start the attack, we first run the attack program in another terminal. Then we run the script, after a while, the script stops which means the /etc/passwd file is modified (screenshot3). Afterwards, we try to switch to the test account, and we are successful. Then we run command “id”; as the screenshot4 shows, we get a root shell. Our attack succeeds.

Task3: Countermeasure: Applying the Principle of Least Privilege



screenshot1, we modified the vulnerable program. Before opening the file, we drop the root privilege. After finishing the opening file, we restore the root privilege.



screenshot2, after we compile and set vulp to Set-UID root program, we perform the attack again. At this time, even our successfully pass access() and open /etc/passwd, our attack still fails. because we do not have root privilege. Open() will check the euid, /etc/passwd requires root privilege, so we cannot open it. Our attack fails.

Observation and Explanation:

Principle of Least Privilege states that a program should not use more privilege than what is needed by the task. In vulnerable program, to open a file in /tmp does not need to use root privilege, so it is the problem stated by the Principle of Least Privilege. To fix this problem, we change the vulnerable program, we add following lines (marked as red):

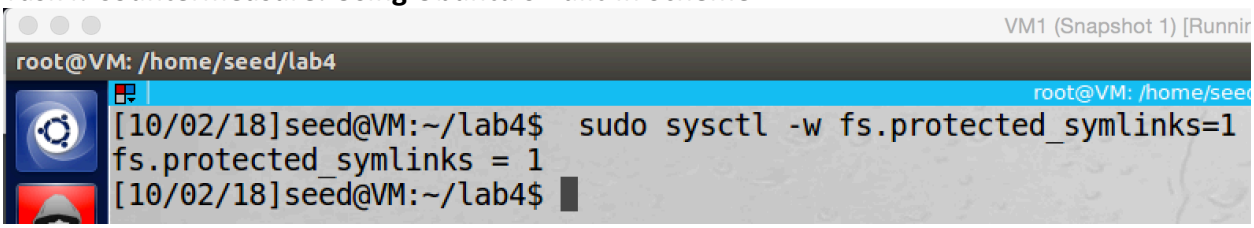
```
/* vulp.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main()
{
        uid_t real_uid = getuid();//get real user id
        uid_t eff_uid = geteuid();//get euid

        seteuid(real_uid);//drop the root privilege by setting euid to real user id

        char * fn = "/tmp/XYZ";
        char buffer[60];
        FILE *fp;
        /* get user input */
        scanf("%50s", buffer );
        if(!access(fn, W_OK)){
```

```
                fp = fopen(fn, "a+");
                fwrite("\n", sizeof(char), 1, fp);
                fwrite(buffer, sizeof(char), strlen(buffer), fp);
                fclose(fp);
        }
        else printf("No permission \n");

        seteuid(eff_uid);//get root privilege by setting euid back to original euid
}
```

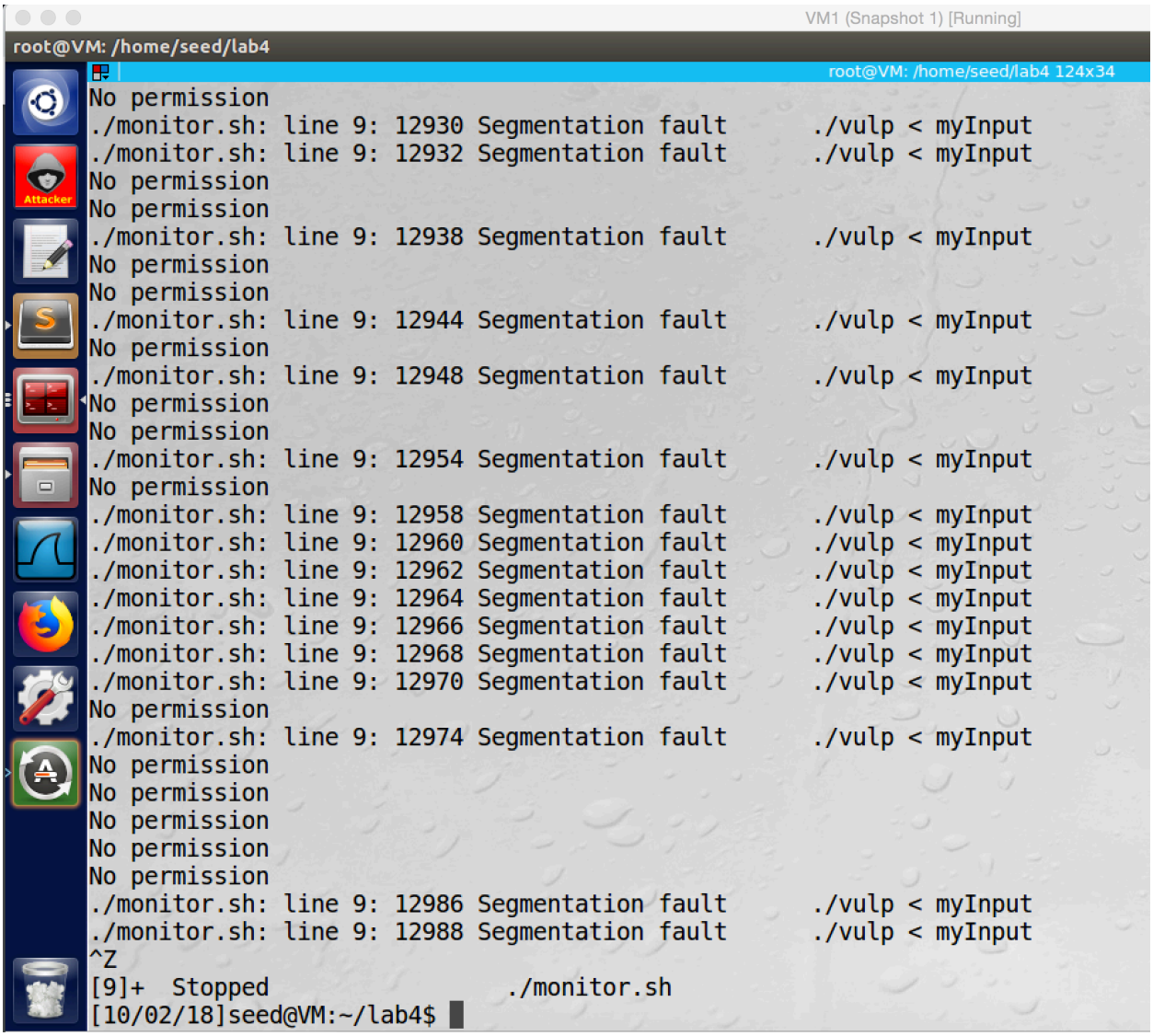
we first store the real user id and effective user id in variables real_uid and eff_uid. Then we call seteuid(real_uid), this function sets the effective user id of current process to its real user id. So it removes the root privilege of the process. After we finished opening a file, we restore the root privilege by seteuid(eff_uid) which sets the effective user id to original euid.

Then we perform the attack again. As the screenshot2 shows, even we pass the access(), and call open() to open /etc/passwd; our attack still fails because we do not have privilege to open and modify this file (open() checks the euid, and our root privilege is dropped by seteuid(real_uid)).

Task4: Countermeasure: Using Ubuntu's Built-in Scheme



screenshot1, we enable the sticky symbolic protection



screenshot2, we perform the attack again. When the vulnerable program runs, it crashes in a lot of times.

Observation and Explanation:

In this task, we perform the attack again; the difference is, we enable the sticky symbolic protection (screenshot1). After we run the script, the vulnerable crashed a lot of times (screesnhot2).

- How does this protection scheme works?

  Sticky symbolic protection is a build-in protection mechanism of Ubuntu. When it turns on, even attacker win race condition, he cannot cause any damage. This protection only applies to world-writable sticky directories, such as /tmp. When the protection is enabled, symbolic links inside a sticky world-writable directory can only be followed when the owner of the symbolic link matches the follower or the directory owner. In our case, the owner of symbolic link is us (a normal user account), but the

owner of directory /tmp and the follower both are root; so they are not match. As a result, the fopen() function will deny to open the target file for us. Therefore, our attack fails.

- What are the limitations of this scheme?

  It only protects on world-writable sticky directories, such as /tmp. It cannot eliminate race condition vulnerability of a program; so if attacker does not use symbolic link as attack tool, then once he exploits and wins the race condition in a vulnerable program, he still can cause damage.

Appendix:

Script used in task2:

```
#!/bin/bash
CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]
do
./vulp < myInput
new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

Content of myInput:

```
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```