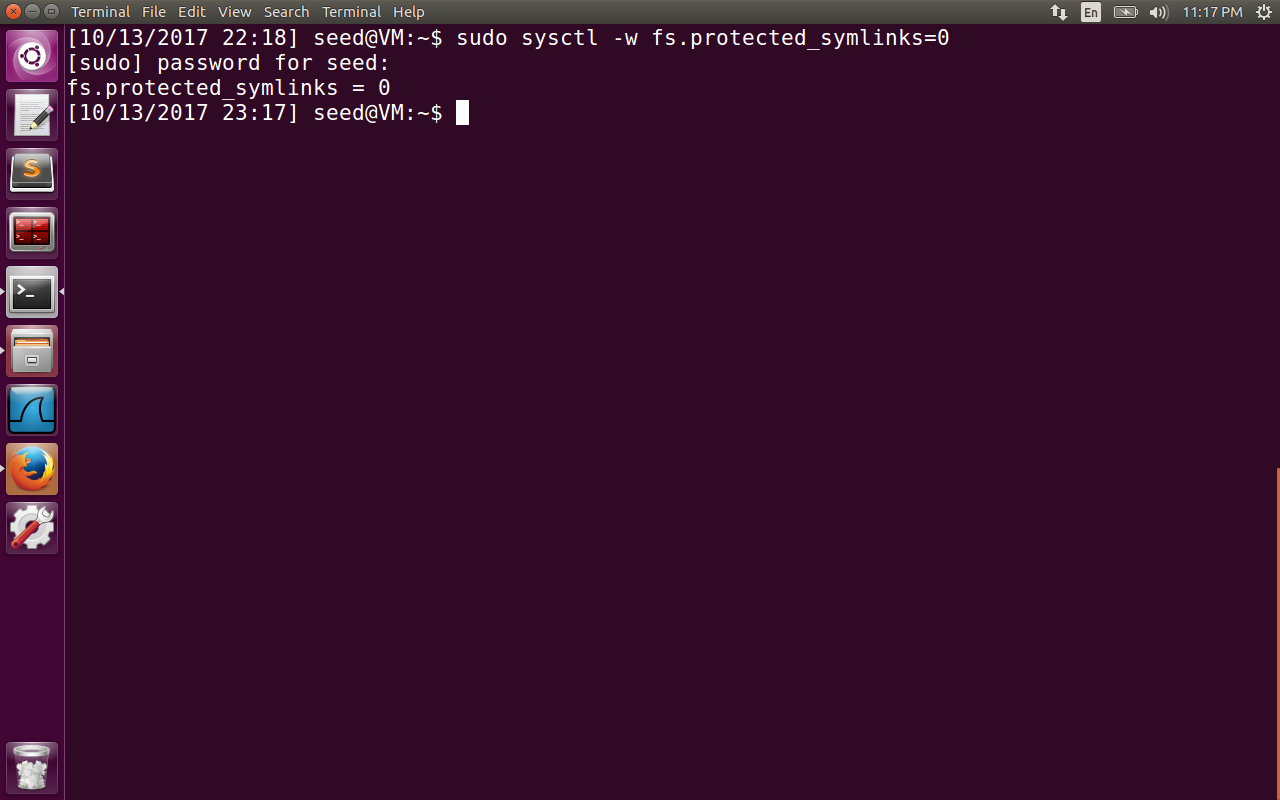
RACE CONDITION & DIRTY COW VULNERABILITY LAB

SUID: 646254141

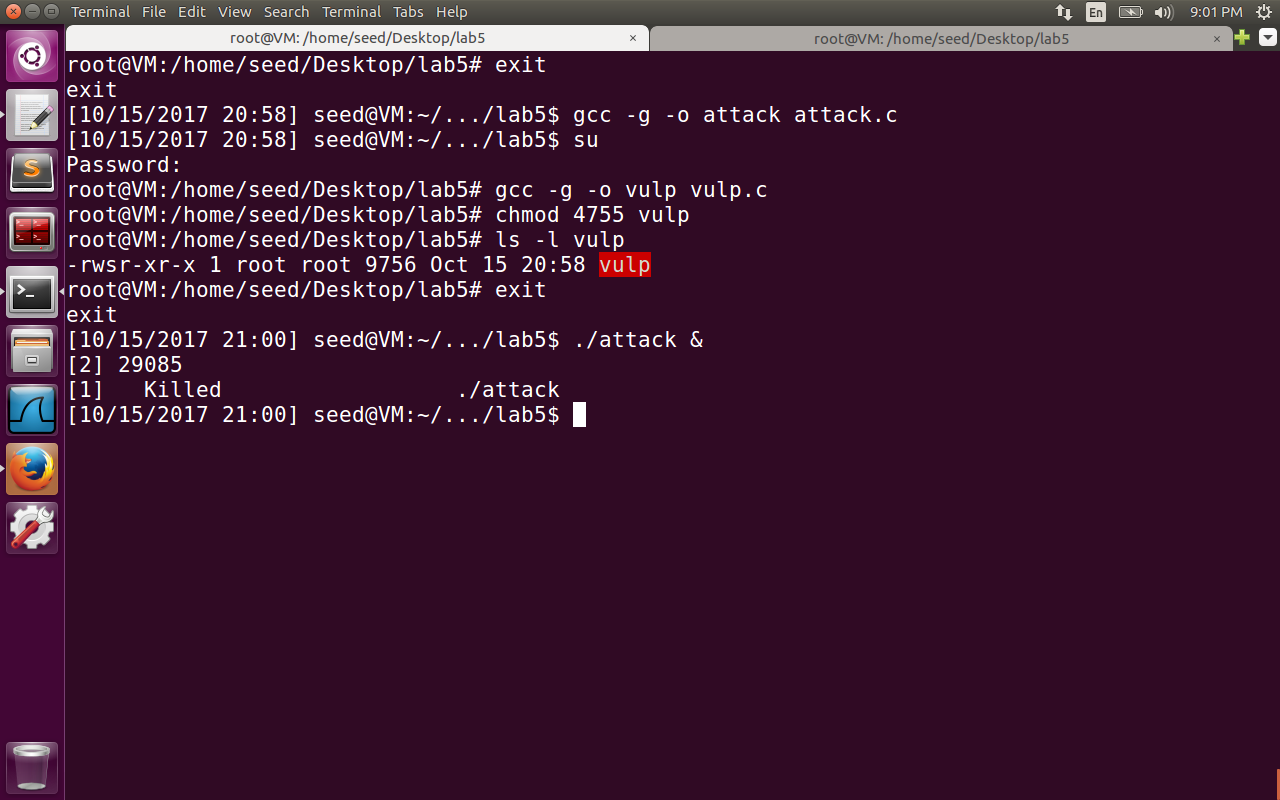
NAME : JASHWANTH REDDY GANGULA

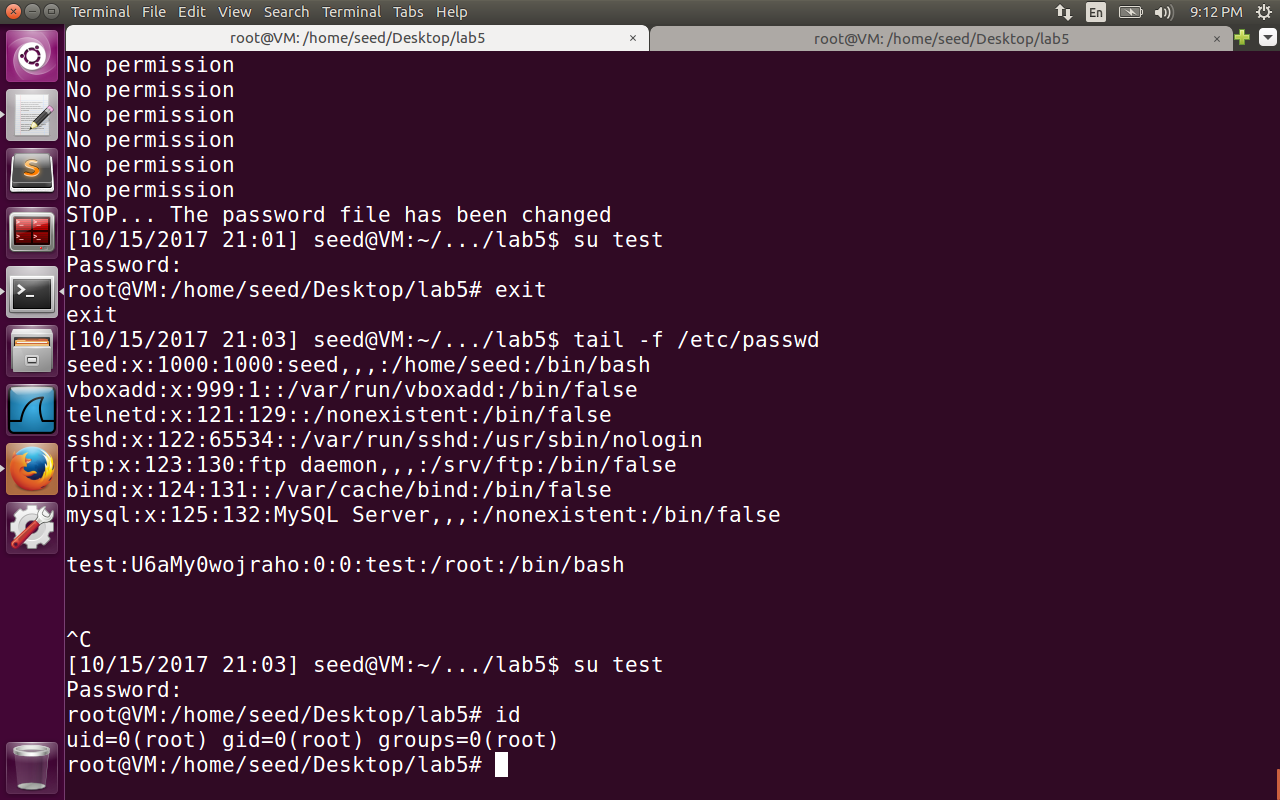
2.1 Initial Setup

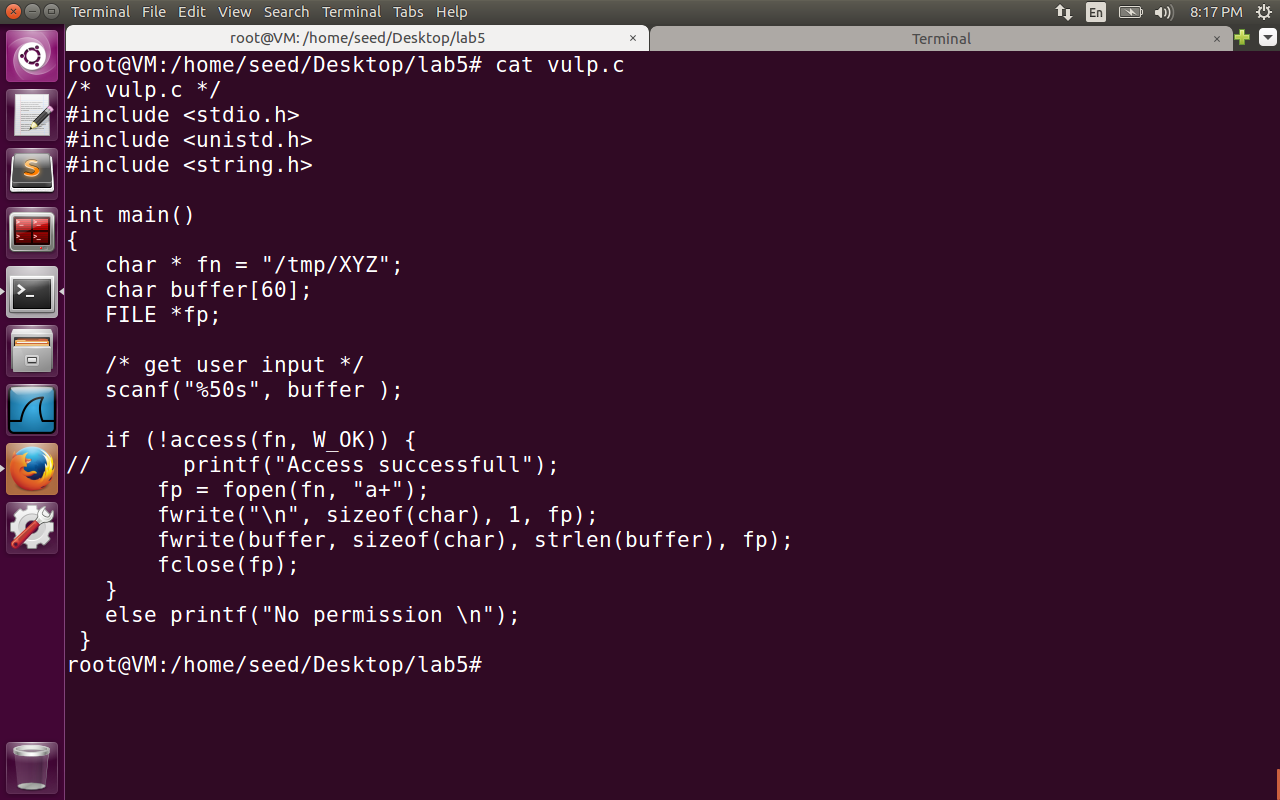


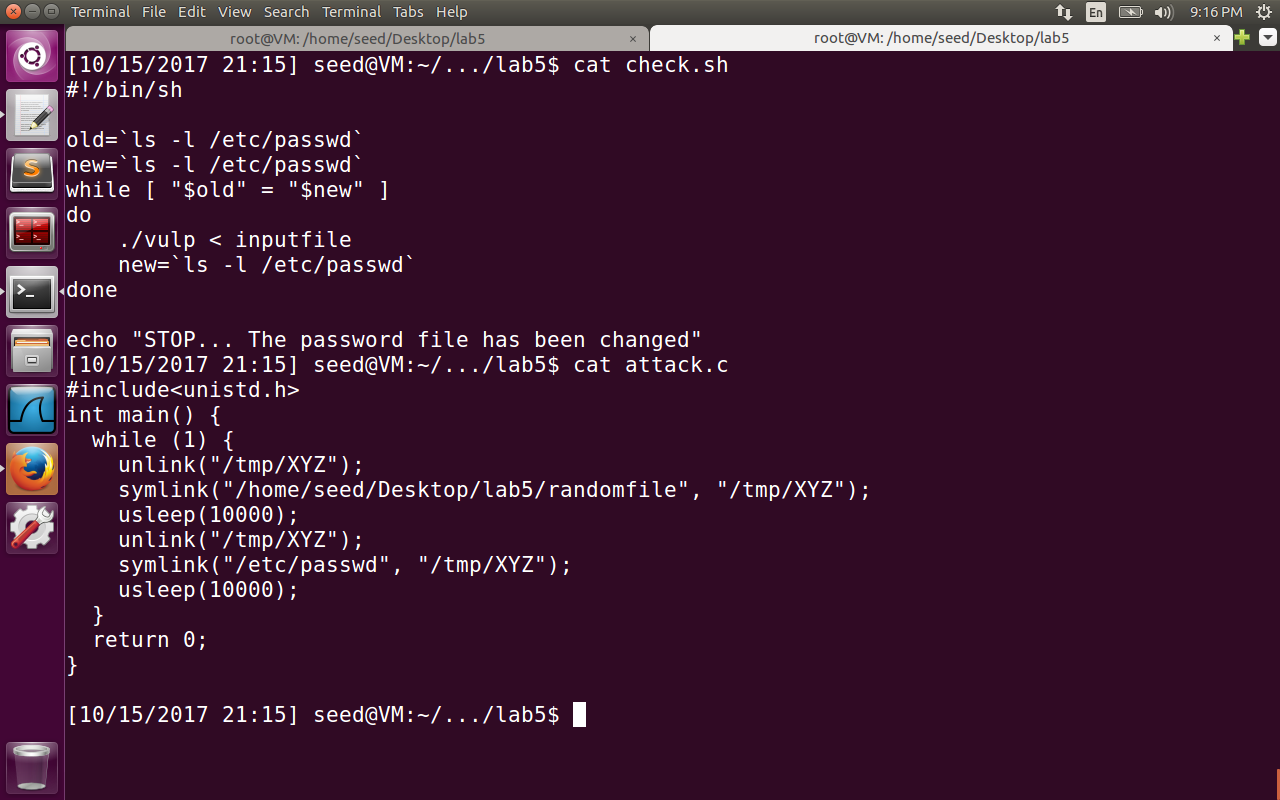
As per the given instructions on Ubuntu VM 16.04 comes with built in protection for race condition attacks. According to the documentation, “symbolic links in world-writable sticky directories (e.g. /tmp) cannot be followed if the follower and directory owner do not match the symbolic link owner.” In this lab, we need to disable this protection.

**2.3 Task 1: Exploit the Race Condition Vulnerability**









1. The above screenshots demonstrate the attack being successful.

2. In the first image we compile the vulp.c program and make it a set-uid program as shown. We also compile the attack.c program which continuously unlinks the “/tmp/XYZ” symbolic link to a random file in the current folder(“/home/seed/Desktop/lab5/randomfile) and to (“/etc/passwd”).

3. The attack program is killed before the start of attack if any using kill -9 pid (process id). The attack process is run in the background.

4. The check.sh is made an executable by using the command chmod a+x check.sh . The check.sh checks for any modification to the /etc/passwd file by continuously monitoring it using the ‘ls –l /etc/passwd’. When the timestamp of file is modified the while loop breaks and prints saying that password file has been modified.

5. The vulnerable program in the script takes input from the input file which contains the following string. “test:U6aMy0wojraho:0:0:test:/root:/bin/bash”  
The string is added to the end of the file /etc/passwd as shown in the figure . The above string adds a user ‘test’ and makes uid as zero with ‘\n’ as password.

6. The test user then has root privileges meaning he can do anything the root can do. The tail –f /etc/passwd displays the last lines of the file which contains the above string added to it. We are also able to login as test user with (enter as password), id = 0(root account privileges).

Even though the vulp has root privileges, it is run by seed user. So the access() call restricts the

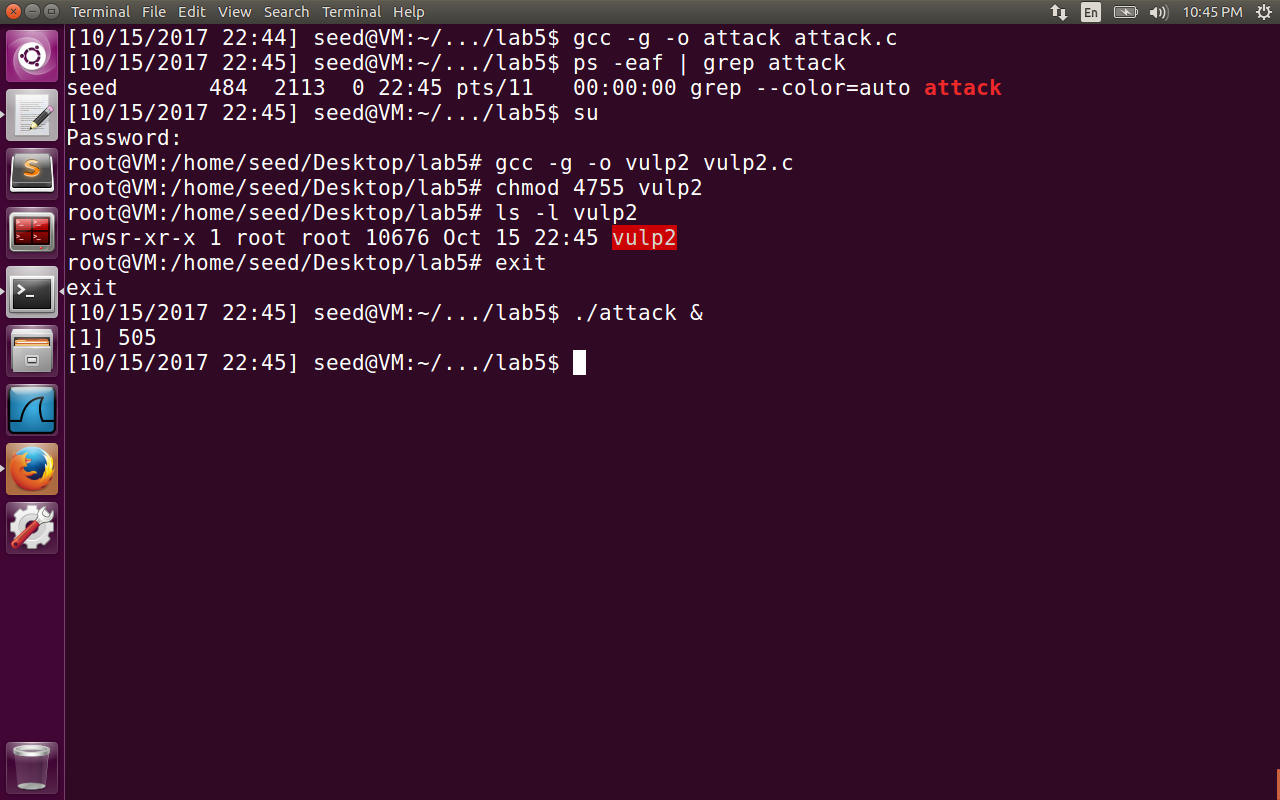
Program from opening root privilege file /etc/passwd. However when the symbolic link points to randomfile which is created by seed user, access() will be successful. The fopen() is always successful if the file exists because it checks for effective user-id unlike access() function which checks for real user-id. Before the fopen() call is made to open the file, if the attack file modifies the symbolic link to point to password file, our race condition attack is successful. The script continuously runs just to hit this condition by trying to bypass the access() check and taking a chance.

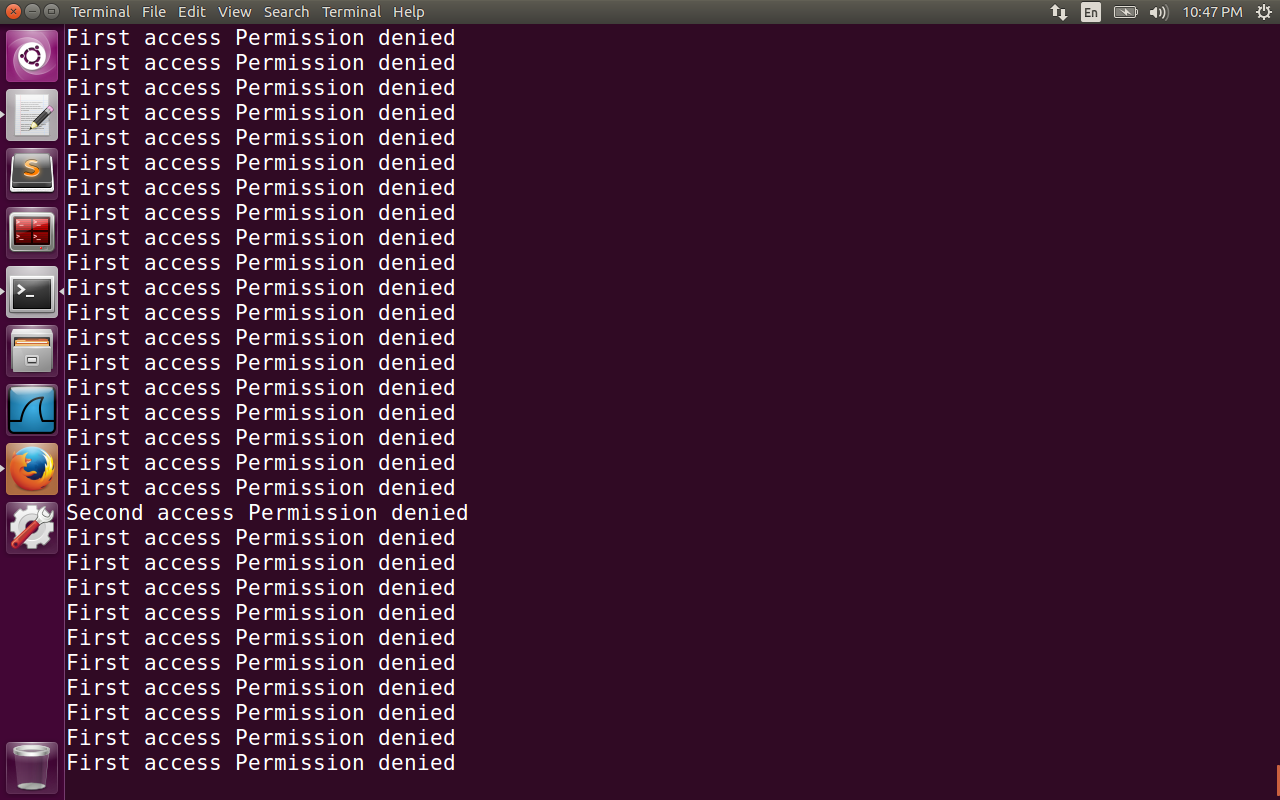
**2.4 Task 2: Protection Mechanism A: Repeating**

In this task2, instead of eliminating the race condition, we try to add many race conditions so that probability of attack reduces by p^(number of race conditions introduced in the program), where p is the probability of successful race condition attack with one race condition in the code.



The above vulp2.c program tries to access the file multiple times. If the first access is successful, we open the required file(“/etc/passwd”) using fd1. The same code is repeated for 2 more times. The probability that our race condition is hit is reduced drastically. If we happen to open random file instead of “/etc/passwd. The stat structure contains a field called st\_ino where this indicates the inode number of the file/directory which is unique. The check stat1.st\_ino == stat2.st\_ino && stat2.st\_ino == stat3.st\_ino will not be successful as the inode information will be different, if the symbolic link changes between the file open mechanisms. This indicates race condition has occurred and attack not being successful. To enter the successful attack loop, the inode number of file descriptors fd1, fd2, fd3 has to be same. Also the access() function has to be successful i.e., when the access() function is running , the symbolic link has to point to some random file which has access permissions for seed user. For all the conditions to be met with the attack program trying to change the symbolic link periodically, the probability of attack reduces a lot compared to first task.

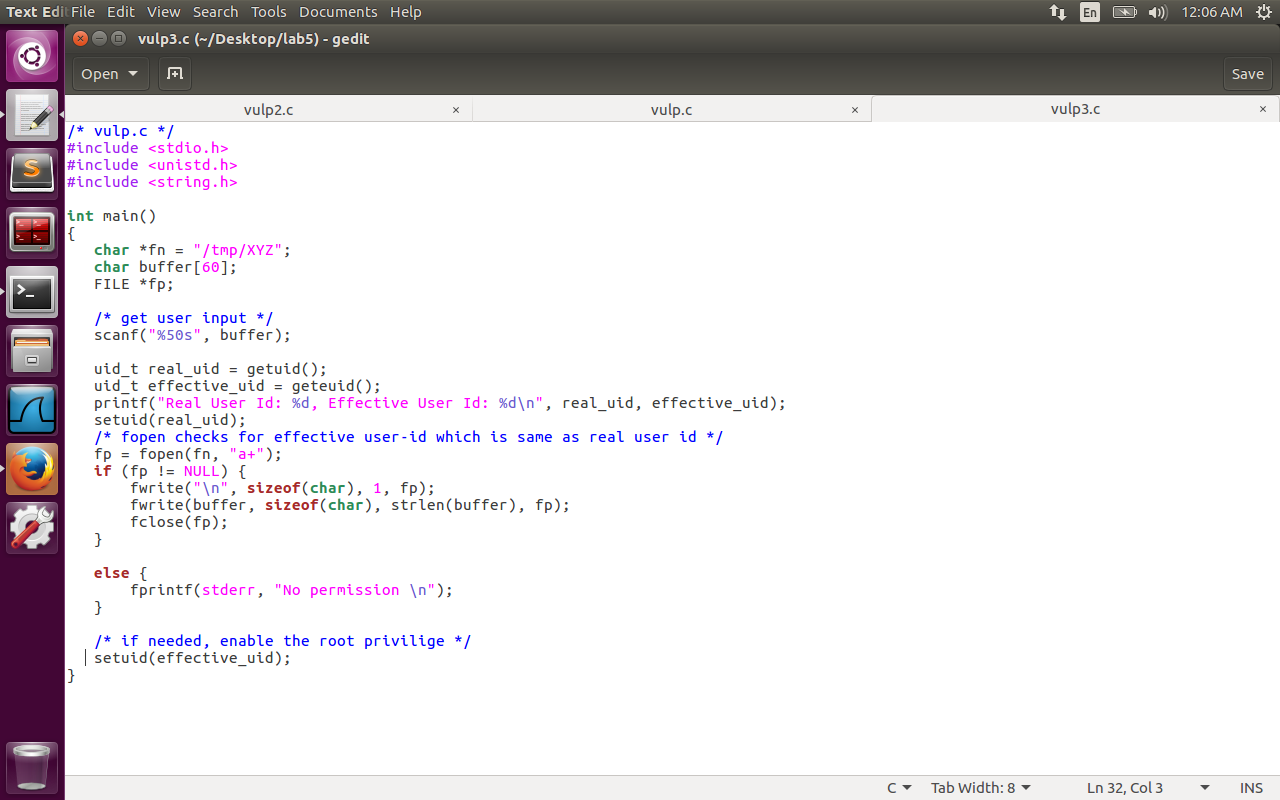




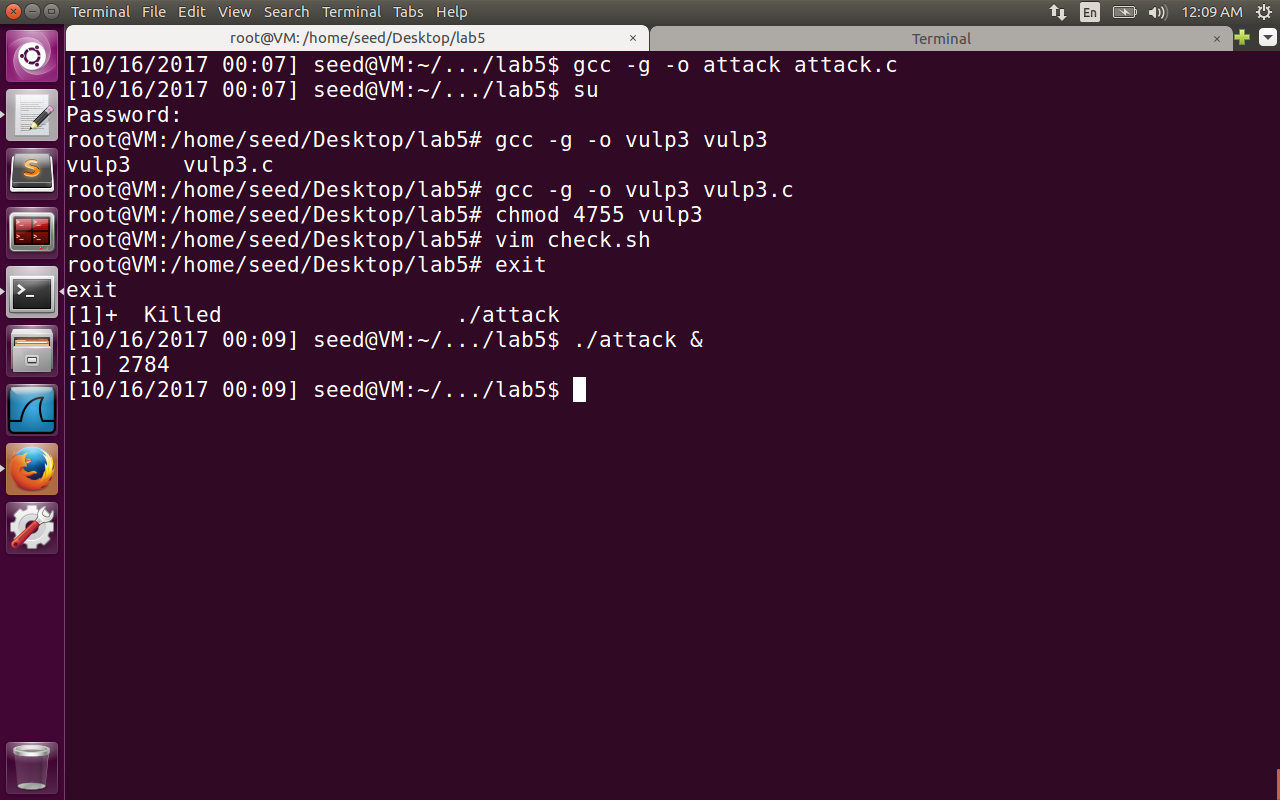
The output of running the script with vulp2 program prints the following output. If we observer here the second access permission is successful rarely and the probability of success for third access will be even small. Even if all the access() methods are successful, we happen to write the randomfile content instead of /etc/passwd file because the symbolic link at this instance was pointing to (/home/seed/Desktop/lab5/randomfile). The above script was tried for nearly half an hour and couldnot be succeeded. If the symbolic link changes between the fopen() methods , we print the output as “Race condition detected”, because the inodes of randomfile(/home/seed/Desktop/lab5/randomfile) and /etc/passwd are different.

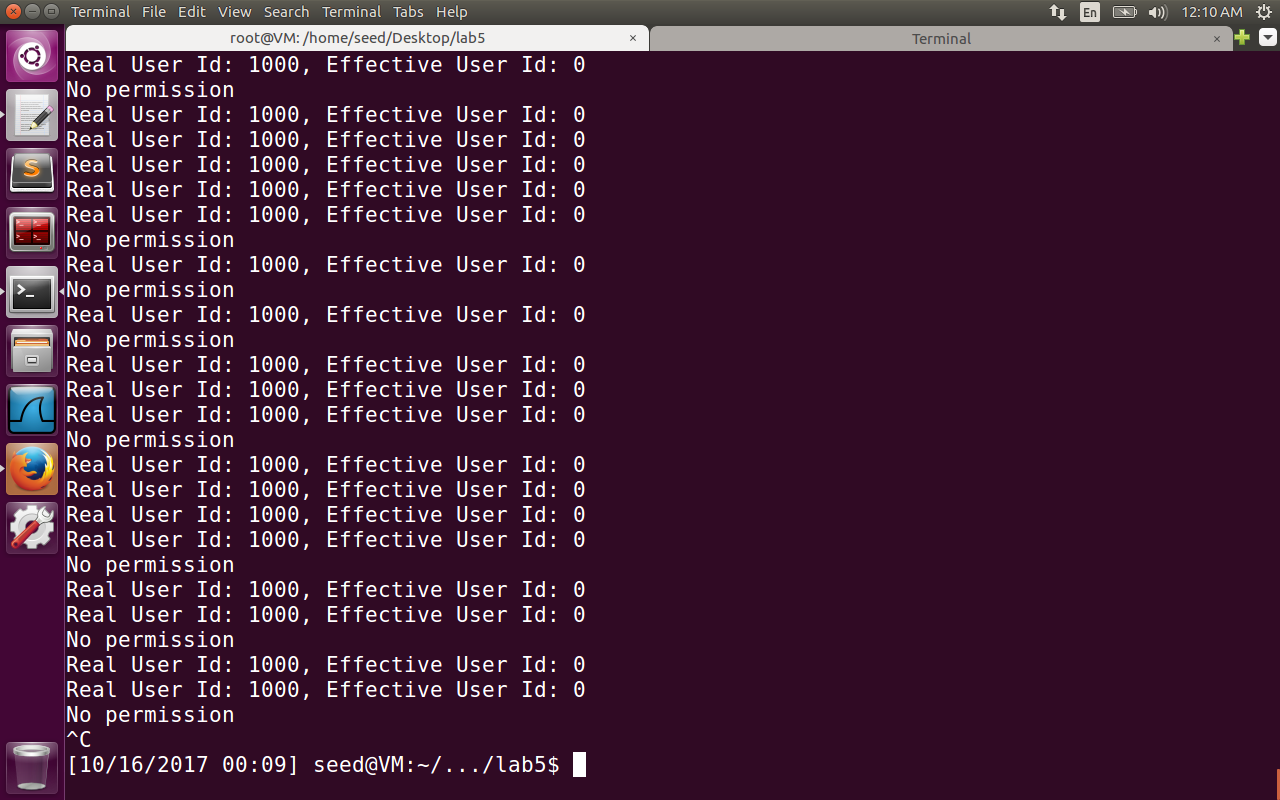
**2.5 Task 3: Protection Mechanism B: Principle of Least Privilege**

The problem with the vulp.c program was violation of Principle of least privilege. The programmer knows that the user who runs the program can be powerful, so uses access() function to limit user’s power. The proper way is to disable the privileges if not needed as shown in the vulp3.c program. Here the effective user id is set to be real user id. So the randomfile owned by seed user can be opened and written, but the race condition is always prevented, because the real user(i..e., seed) cannot open the “/etc/passwd” file.

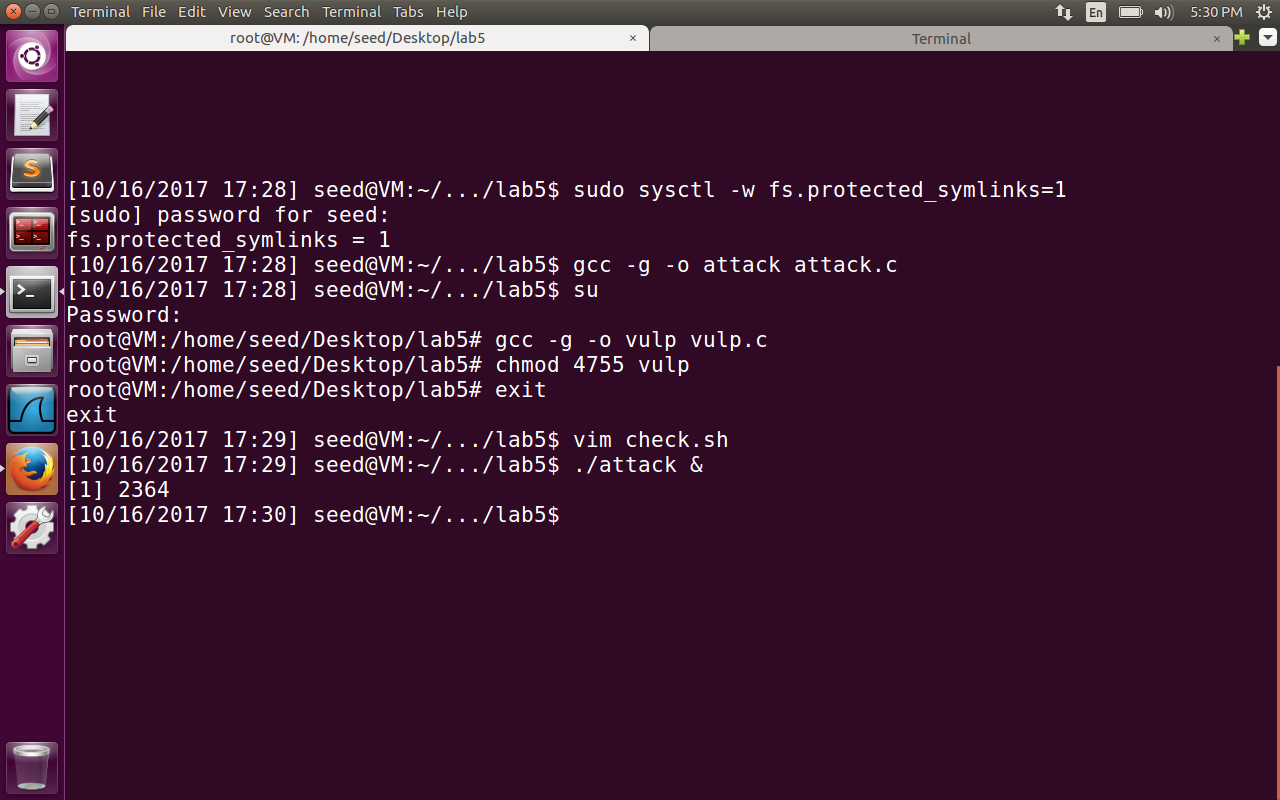


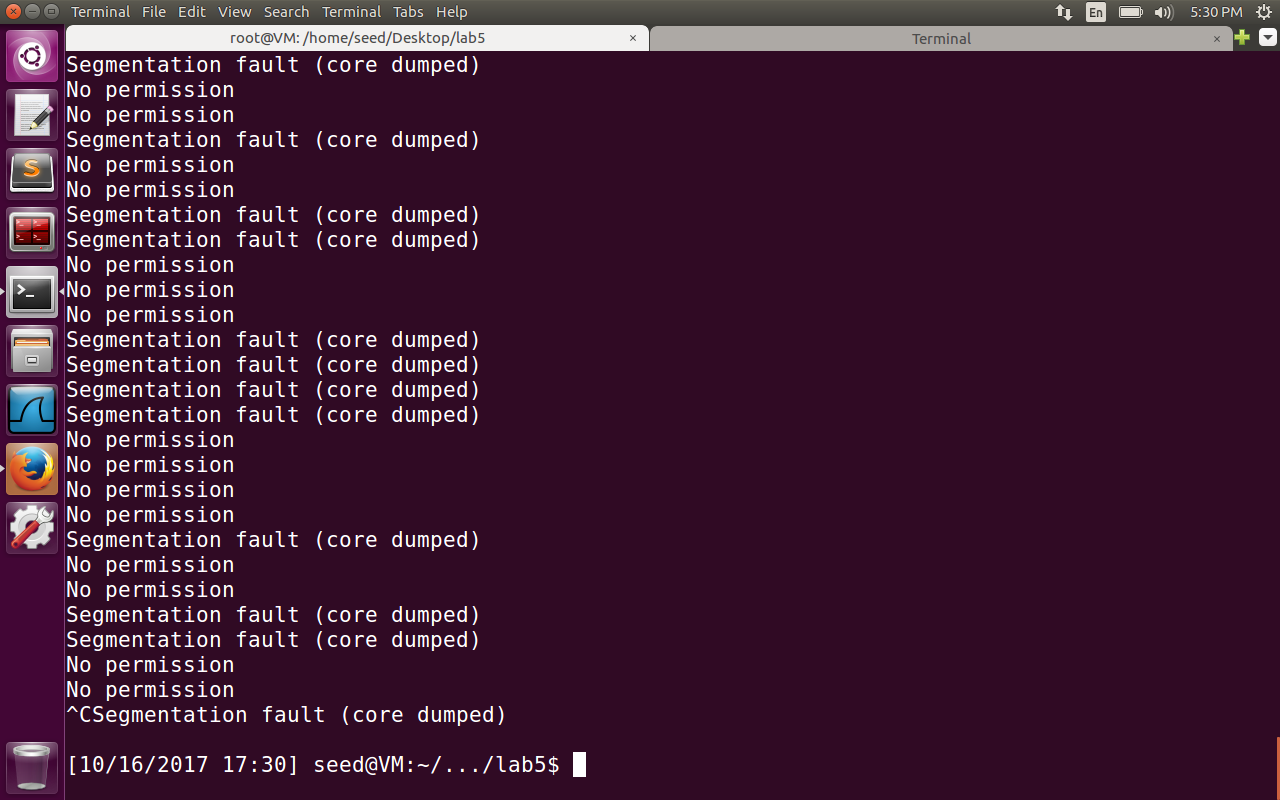
The check.sh program is changed to ./vulp3 < inputfile. Same process is repeated as in task1 , but the race condition attack was not successful always with the below output.  
The program will be successful to open random file and will be written with content, but opening of password always throws stderr (permission denied).





**Task 4: Protection Mechanism C: Ubuntu’s Built-in Scheme**

****

****

For protection against race conditions, we have enabled the sticky bit. Here /tmp folder is world writable directory. When the user following the symbolic link is different from the actual owner of the file, the link will not be resolved. This is the patch applied to resolve race condition attacks. Here the seed user wants to resolve /tmp/XYZ symbolic link to be pointed to /etc/passwd file. The owner of password file is root, different from seed user. Hence the attack is not successful. Hence segmentation fault is thrown as we try to access the file whose owner is different from user.

(1) Why does this protection scheme work?

In world writable folders like /tmp, other users should-not be able to delete the files created by user. Sticky bit is the user ownership access flag that is assigned to files and folders on unix systems. Typicaly this is applied to /tmp folder to prevent the ordinary users from renaming or deleting other user’s files. In our case when we try to unlink the /tmp/XYZ folder when it points to password file, the segmentation fault might be thrown, which is what we need to prevent the attack.

(2) Is this a good protection? Why or why not?

This depends on the use case like for example this is a good protection to prevent such race condition attacks. However, if the users are not aware of the protection enabled, sometimes the behavior at run time might be difficult for the user to understand.

(3) What are the limitations of this scheme?

When multiple users of application are trying to access the symbolic links in world writable directories like /tmp, it cannot be resolved. So the users cannot work on such applications when such a scheme is enabled on the system.