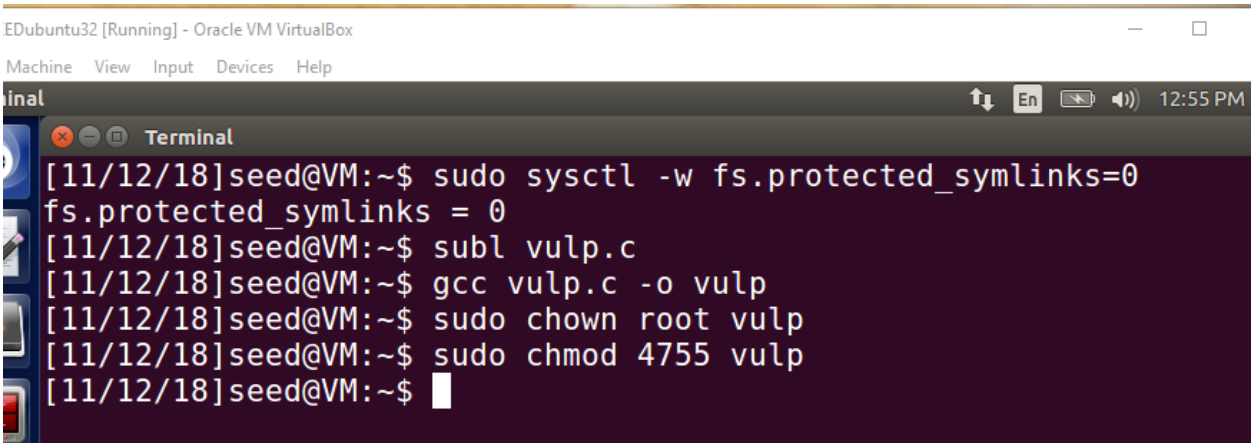
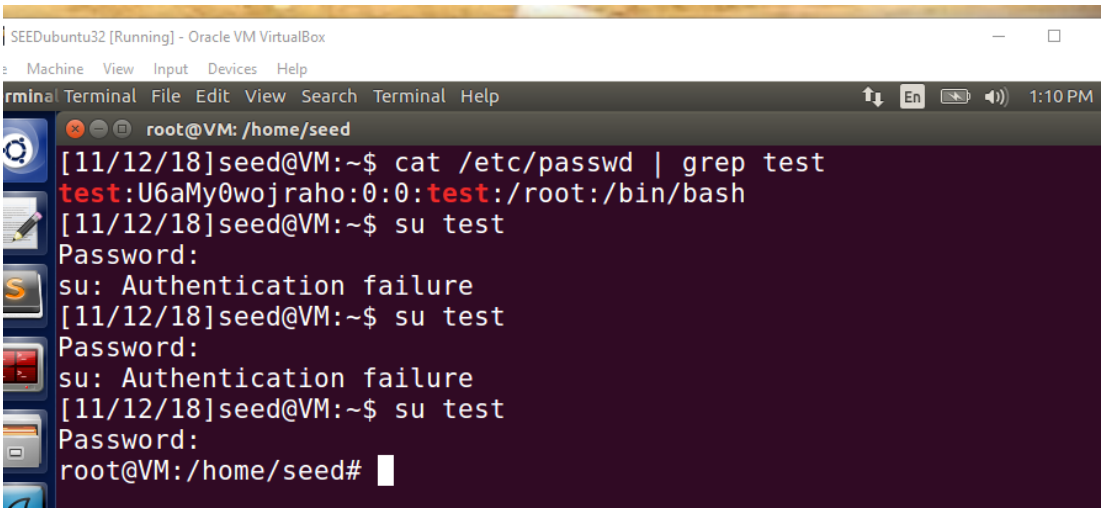
Race Condition Vulnerability Lab- Sample 2



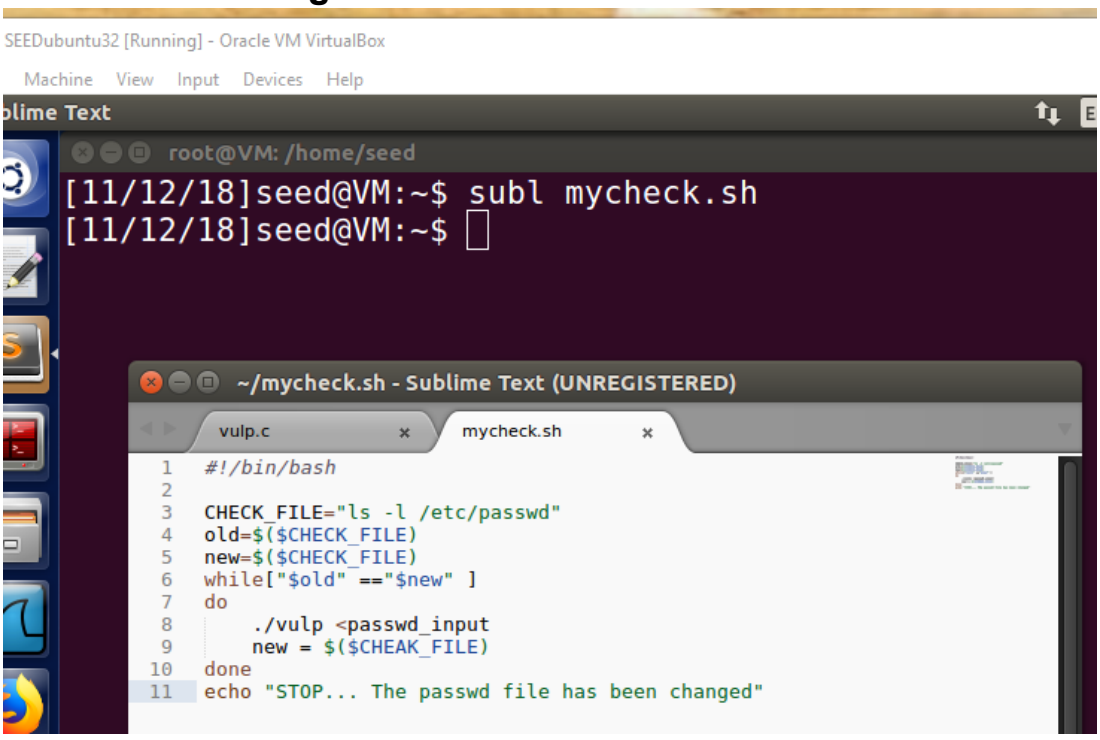
Observation: According to the documentation, “symlinks in world-writable sticky directories (e.g. /tmp) cannot be followed if the follower and directory owner do not match the symlink
owner.” In this lab, we need to disable this protection. You can achieve that using the following commands:
sudo sysctl -w fs.protected_symlinks=0
I compile the above code, and turn its binary into a Set-UID program that is owned by the root.

Task 1: Choosing Our Target



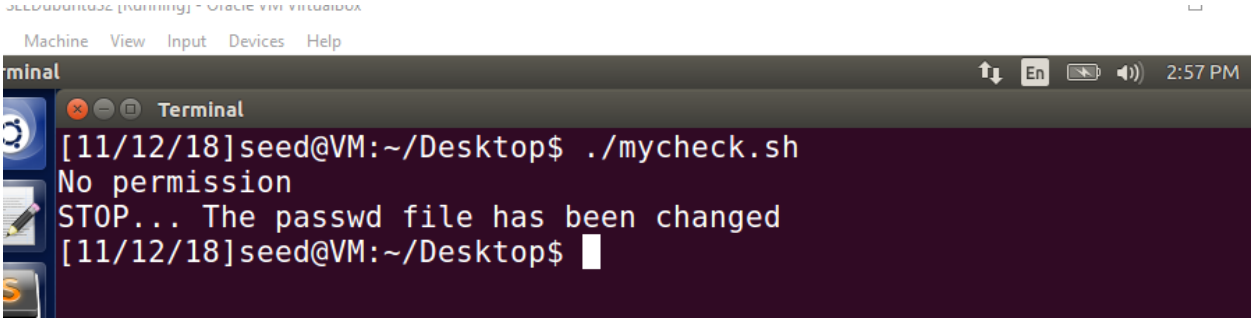
Observation: To verify whether the magic password works or not, i manually (as a superuser) add the following entry (*test:U6aMy0wojraho:0:0:test:/root:/bin/bash*) to the end of the */etc/passwd* file. I can log into the test account without typing a password, and have the root privilege.

Task 2: Launching the Race Condition Attack



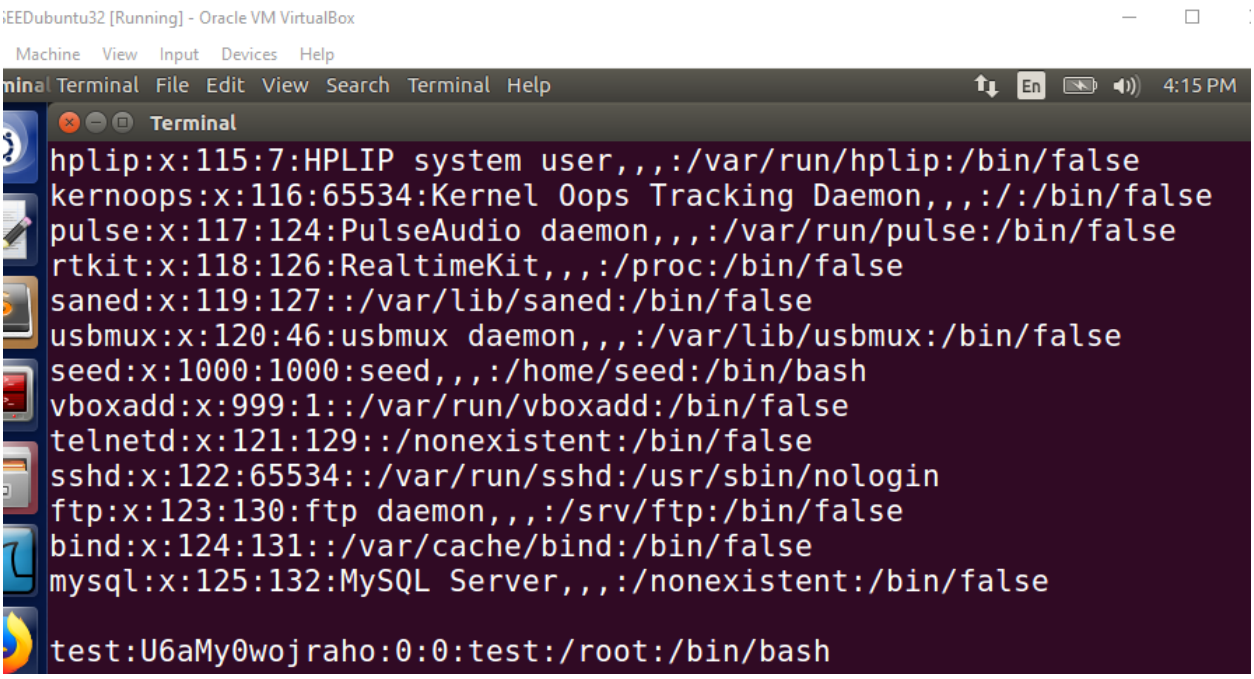
passwd_input.txt

Observation: Since i need to run the attacks and the vulnerable program for many times, i need to write a program to automate the attack process. To avoid manually typing an input to the vulnerable program *vulp*, i can use input redirection. Namely, i save my input in a file, and ask *vulp* to get the input from this file using "*vulp* < *passwd_input*".
Knowing whether the attack is successful. Since it may take a while before our attack can successfully modify the password file, we need a way to automatically detect whether the attack is successful or not. The following shell script runs the "*ls -l*" command, which outputs several piece of information about a file, including the last modified time. By comparing the outputs of the command with the ones produced previously, we can tell whether the file has been modified or not.



exploit.c

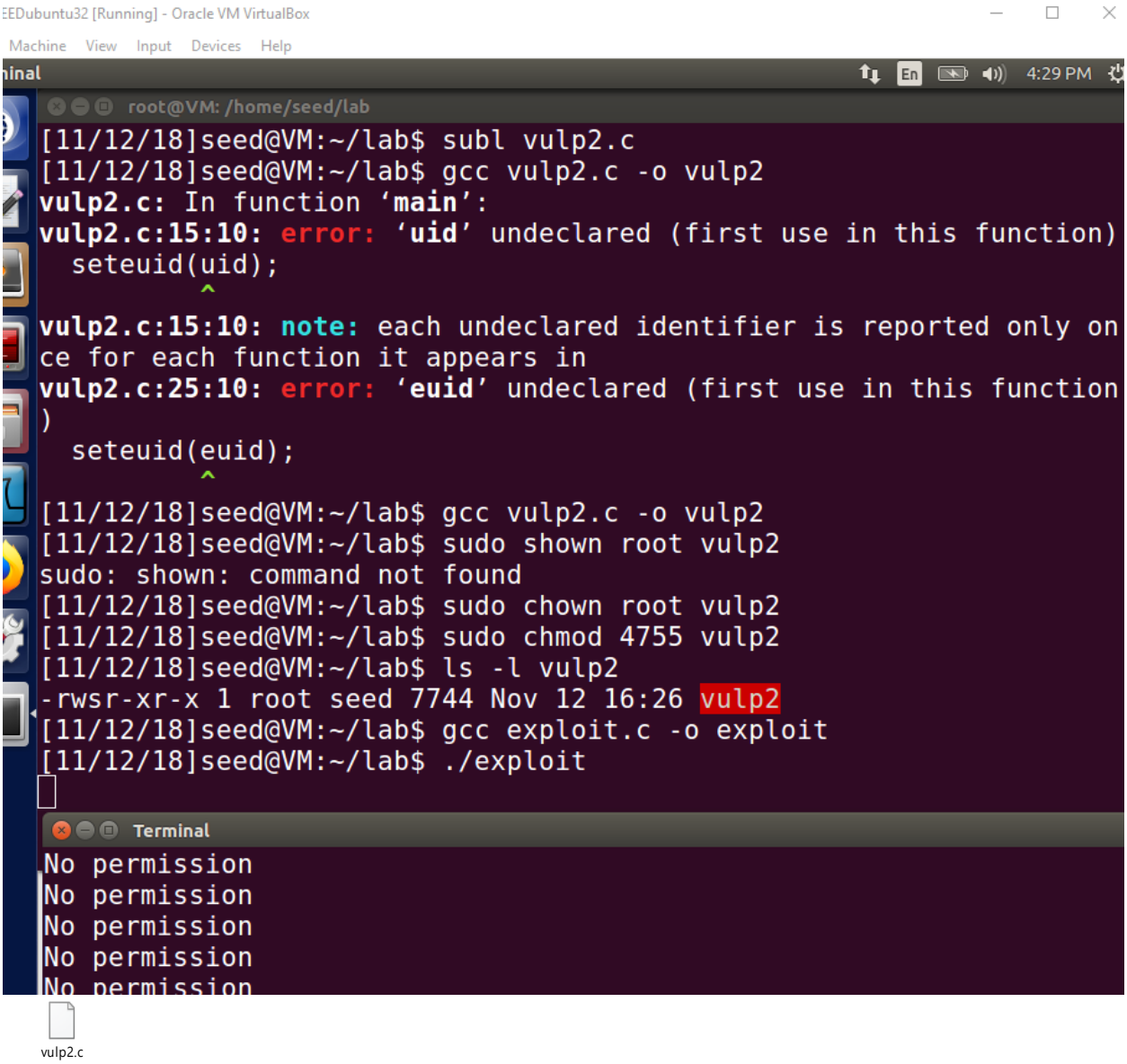
Observation: We can see after multiple attempts at exploiting the race condition, our attack runs and the message is displayed by *mycheck.sh* .



Observation: It can be observed that our *passwd* file has been appended with a new user with root privileges.

Explanation: In this task we are trying to use race condition vulnerability and trying to exploit the time frame between time of check and time of use. */tmp* and */var/tmp* are world writable directories. So anyone can write into these directories. But only the user can delete or move his files in this directory because of the sticky bit protection. So in this experiment, we turn off the sticky symlinks protection so that a user can follow the symbolic link even in the world writable directory. If this is turned on, then we cann't follow the symbolic link of another user inside the sticky bit enabled directory like */tmp*. We will try to make it point to root owned file initially and then unlink it and make it point to root owned file so that we can exploit this and make changes to the root owned file. To protect against set UID programs making changes to files, the program uses *access()* to check the real UID and *fopen()* checks for the effective UID. So our goal is to point to a user owned file to pass the *access()* check and point to a root owned file like password file before the *fopen()* since this is a set UID program and the EUID is root, this will pass the *fopen()* check and we gain access to the password file and we add a new user to the system. With multiple attempts from user, we are able to exploit this window.

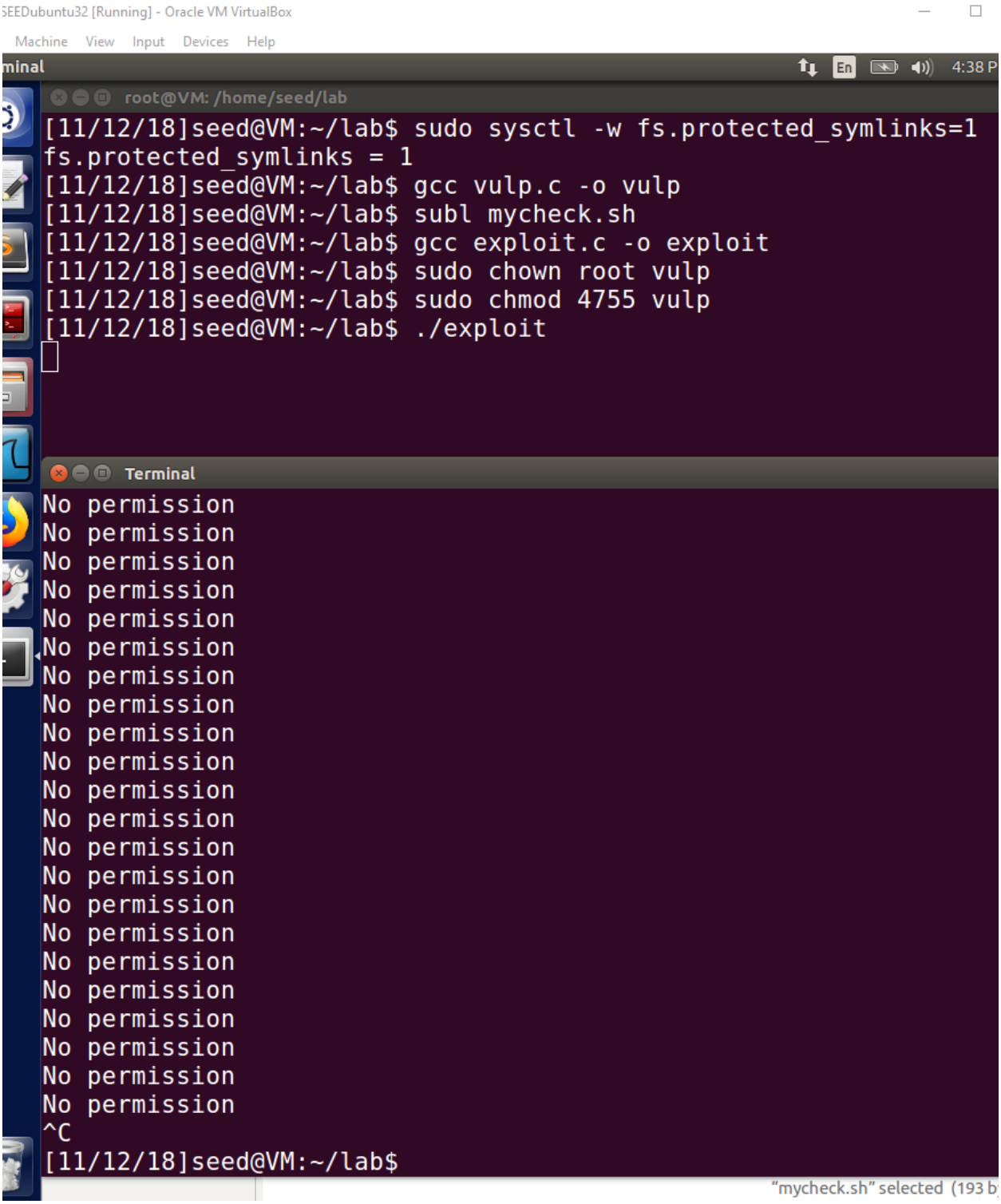
Task 3: Countermeasure: Applying the Principle of Least Privilege



Observation: The above screenshot and program show that we modified the vulnerable program so that we downgrade the privileges before the cheeks and then revise the privileges at the end of the program. If we perform the same attack again, it doesn't work and we can't update the root owned password file and hence we do not create a new user in the system.

Explanation: The above program downgrade the privileges just before the check. So the EUID will be the real UID. So this passes the accesss check as usual since the symbolic file points to seed owned file initially. *fopen()* cheeks for the EUID and here the EUID is downgraded to that of the real UID of seed and when the symbolic link points to protected file, seed doesn't have permissions to open that file and the attack fails since we cann't access and modify root owned files.

Task 4: Countermeasure: Using *Ubuntu*'s Built-in Scheme



Observation: In the above case we turn on the sticky bit protection and perform the same attack. We find that the attack is not successful as we cann't follow the symlinks from the */tmp* directory.

Explanation: My attack failed, because this is a built in protection mechanism to prevent such attacks.
The protection scheme worked since in this case, the follower is root, and owner of the */tmp* directory is *root* and the symlink owner is seed. Thus access will be denied. This isn't a good protection mechanism as this has a few limitations. The mechanism works only for sticky bit directories like */tmp* or */var/tmp*. So the attacker can exploit the race condition in other directories and gain access.