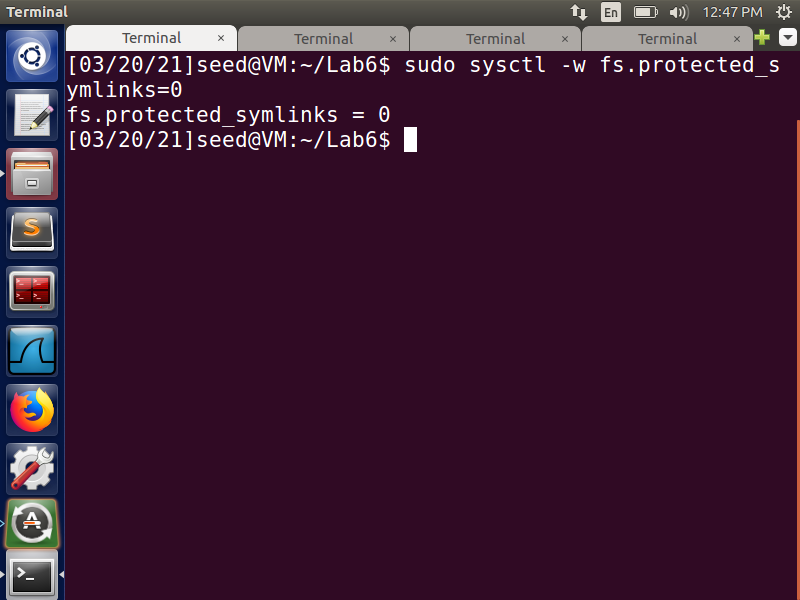
**Homework 6 :- Race Condition Vulnerability Lab**

**Name :- Aparna Krishna Bhat**

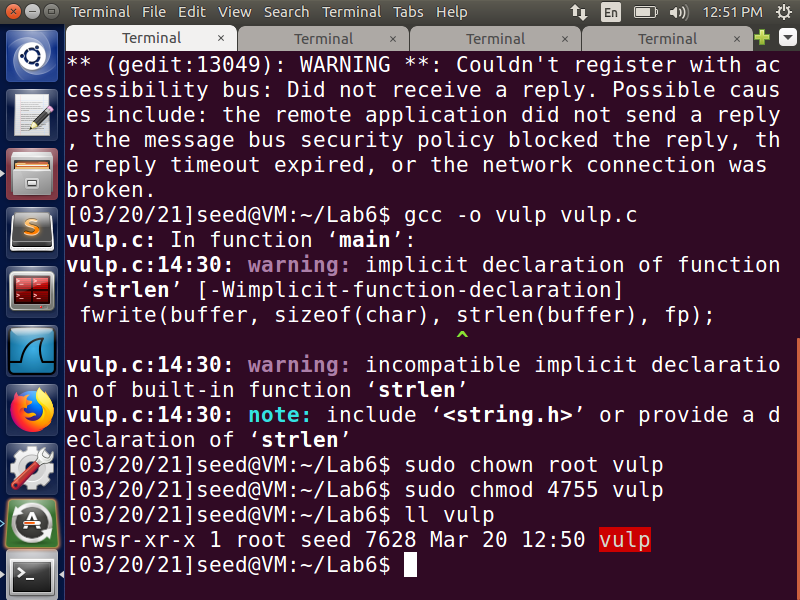
**ID :- 1001255079**

**Initial setup**

First, we disable the built-in protection against race condition by executing the following command



Next, compile the given vulp.c program and make it a Set-UID root program. These operations can be seen in the Fig 1 screenshot.

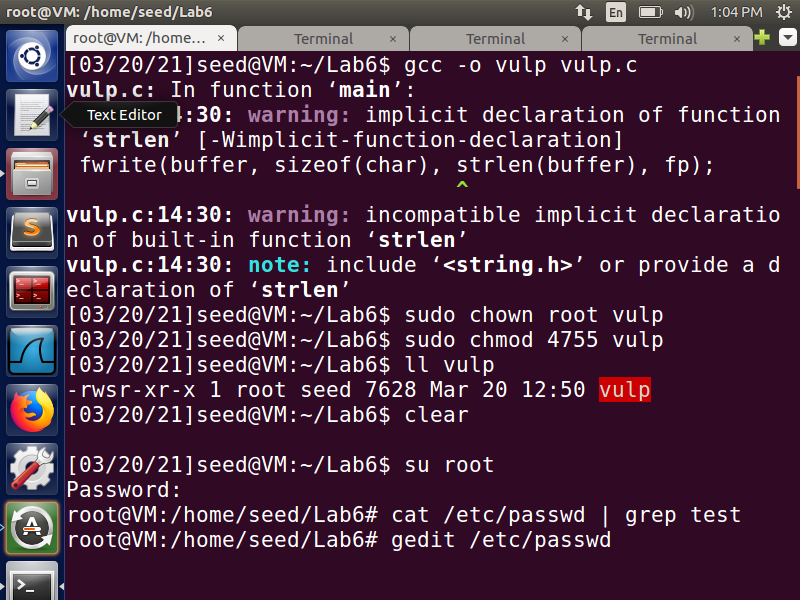


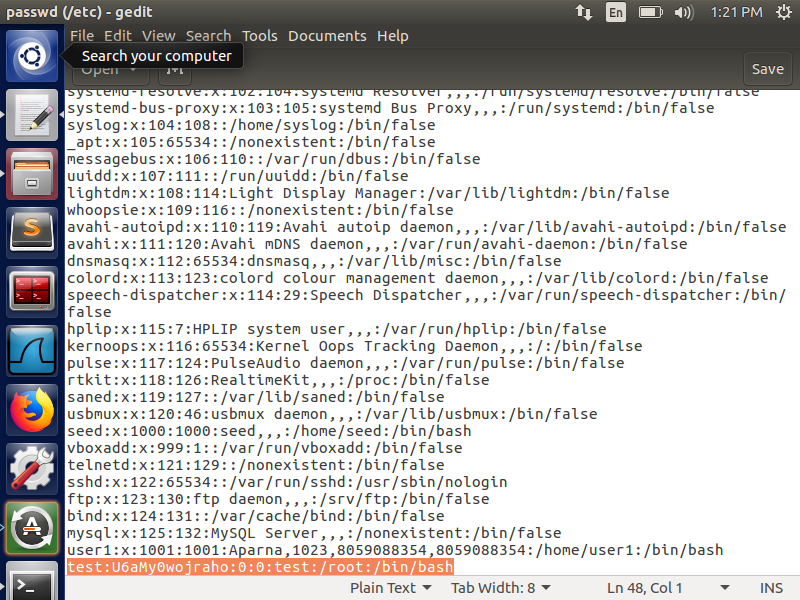
**Fig 1:- Making vulp.c a Set-UID program**

**Task 1: Choosing Our Target**

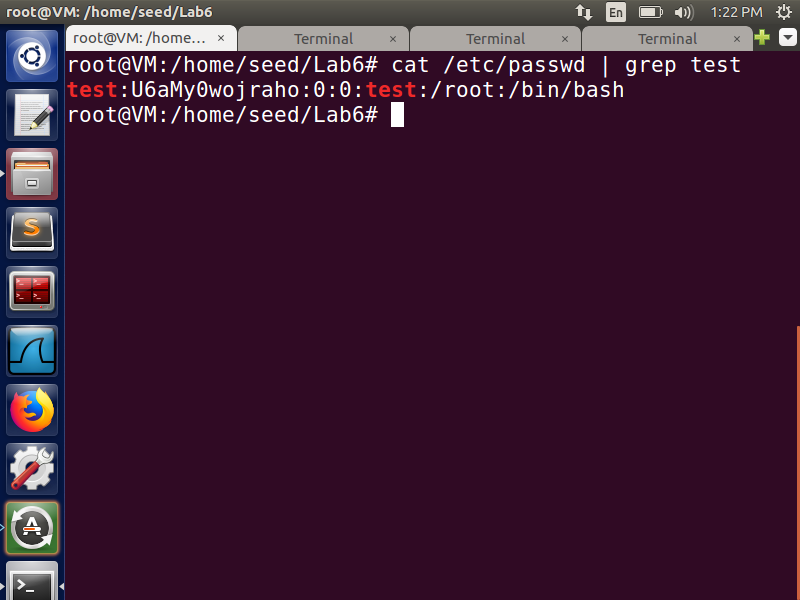
We first check to see if there is a user called ***"test".*** We can see from the below screenshot that there is no one with that name. So, from the root account, we can manually enter the following text in the ***/etc/passwd*** file.

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

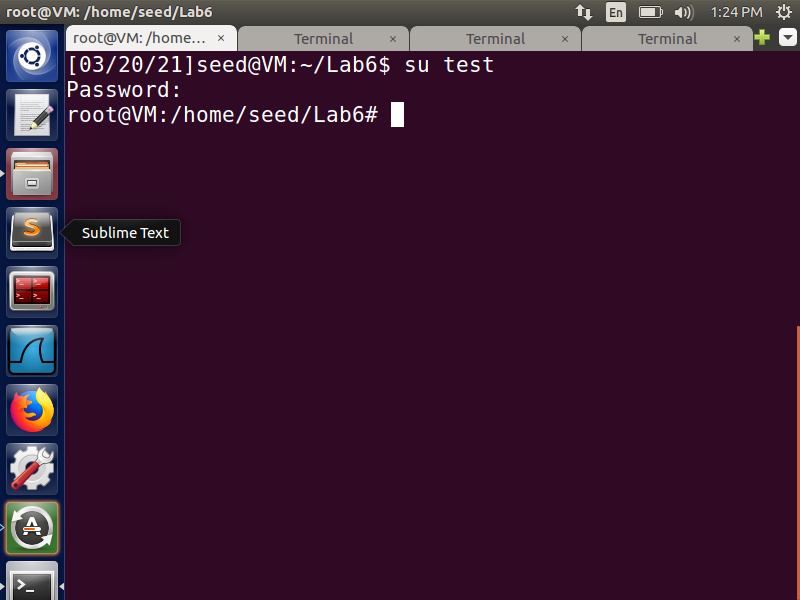




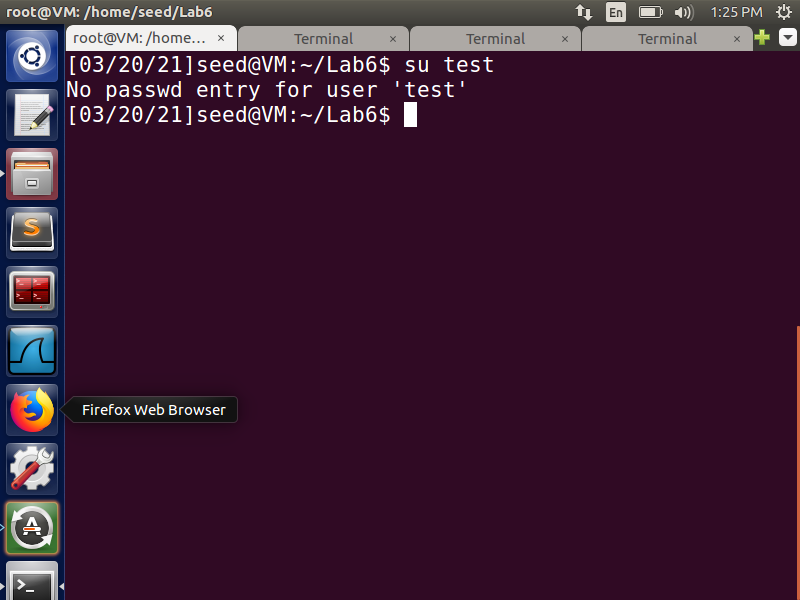
After editing the /etc/passwd file, we check the file again and see that the entry has been made.



We then switch from seed to test and press enter when prompted for the password to see if the new user was successfully created. We have successfully switched to the test user account, and the ‘#' indicates that it is a root account

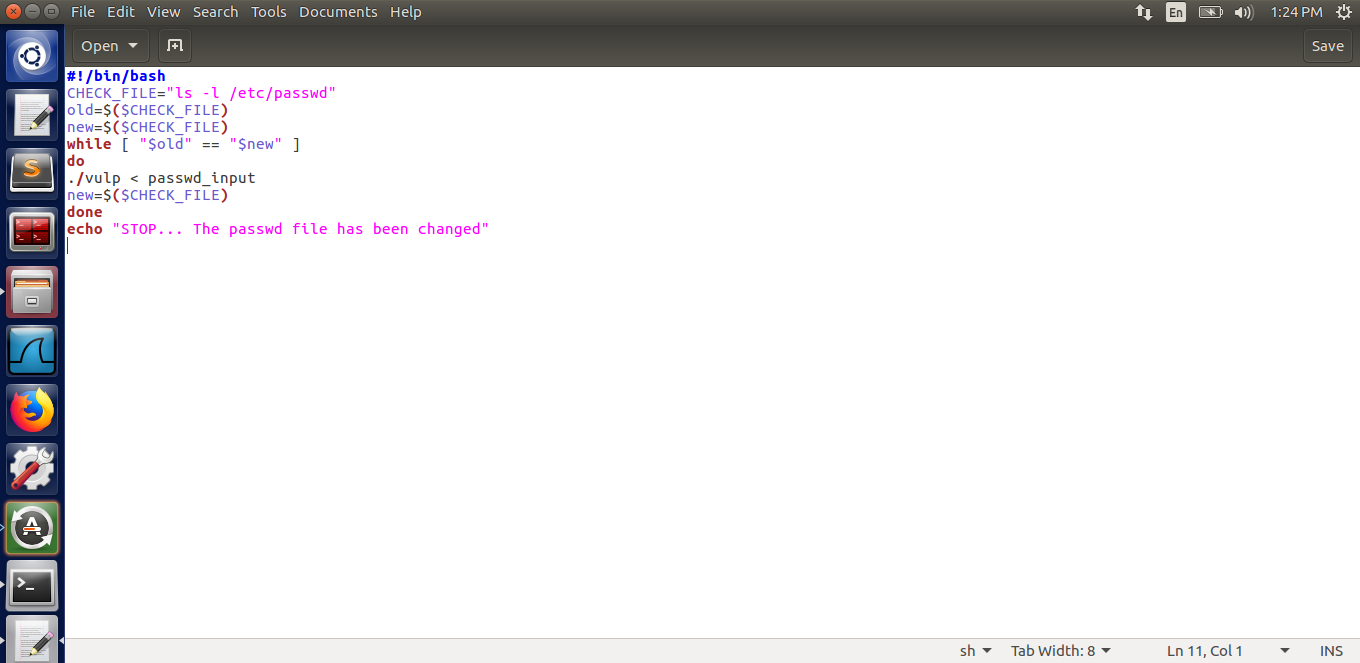


We delete the manual entry from /etc/passwd after checking that our entry works, and this is checked by doing the following



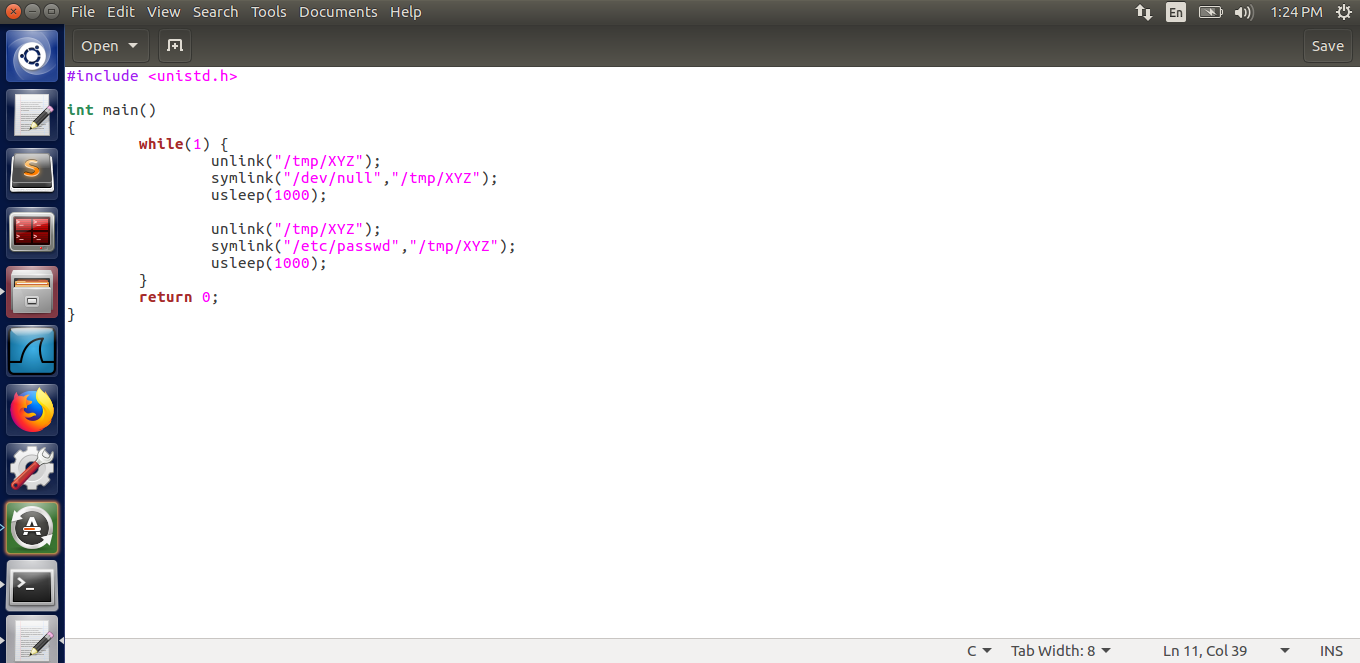
**Task 2:- Launching the Race Condition Attack**

We will need to run two processes at the same time to launch the attack. Unless the passwd file is modified, the target.sh runs the privileged program in a loop. The user entry from task 1 is saved in the passwd\_input file and passed to the privileged program as an input. The code in the below screenshot shows how the file's timestamp is used to break the loop; it stops when the timestamp changes, indicating that the file has been modified.

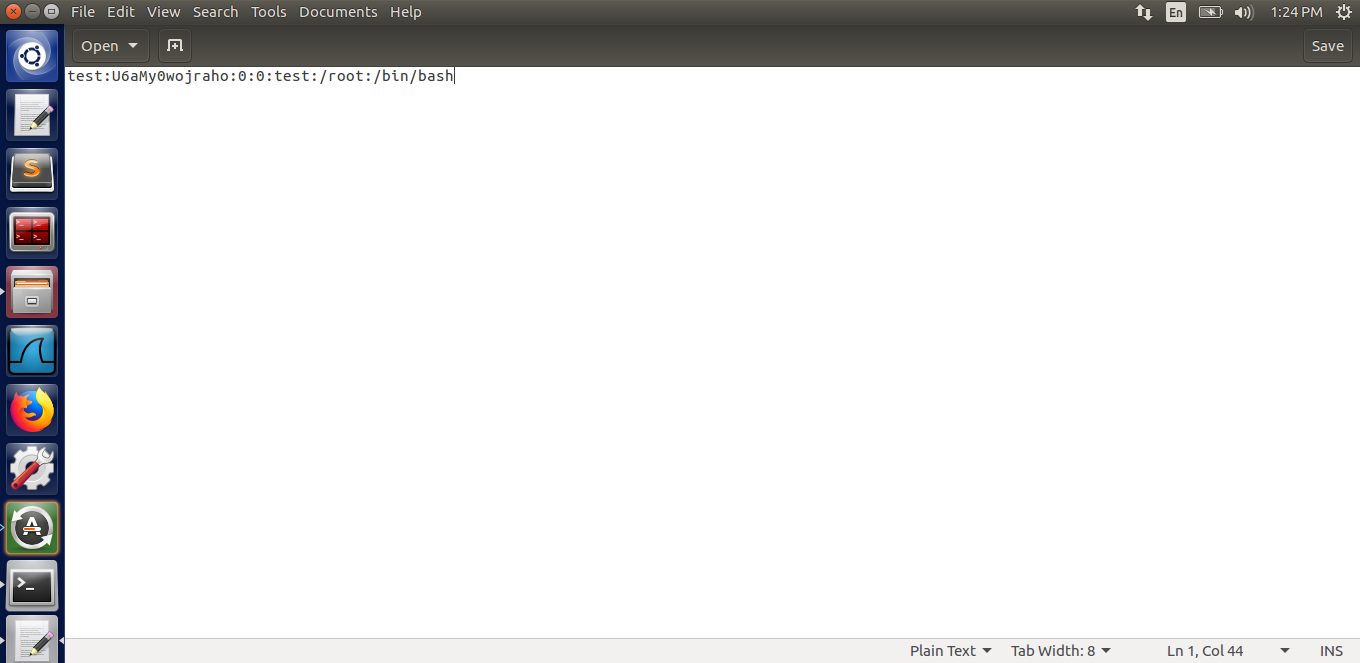


**Fig :- target.sh**

The attack program, on the other hand, runs in parallel with the target.sh process and attempts to change the location of ***“/tmp/XYZ”*** so that it points to the file we want to write to using target.sh program. We use unlink to remove the old link and then symlink to make a new one. To pass the access check, we make the ***“/tmp/XYZ” point to “/dev/null”*** at first. Since, “/dev/null” is writable by anyone and is a special file that discards anything written to it, we can pass this check. After that, ***we put the process to sleep for a while and then point the “/tmp/XYZ” file to the “/etc/passwd”*** file, which is the one we want to write to. This process is done in a loop to race against the target.sh program.

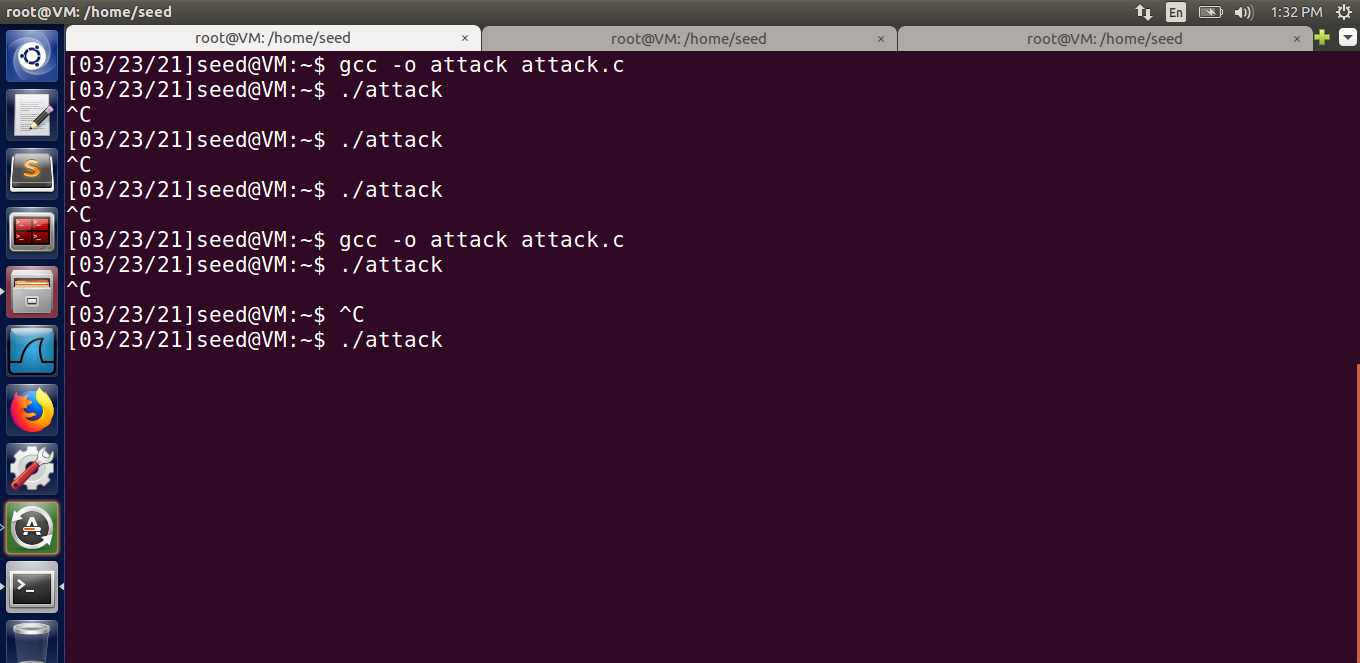


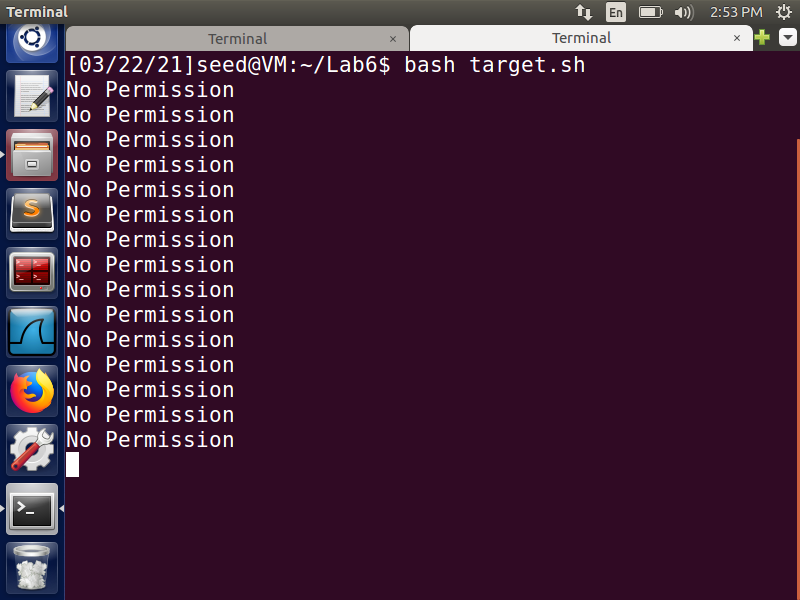
**Fig: attack.c**



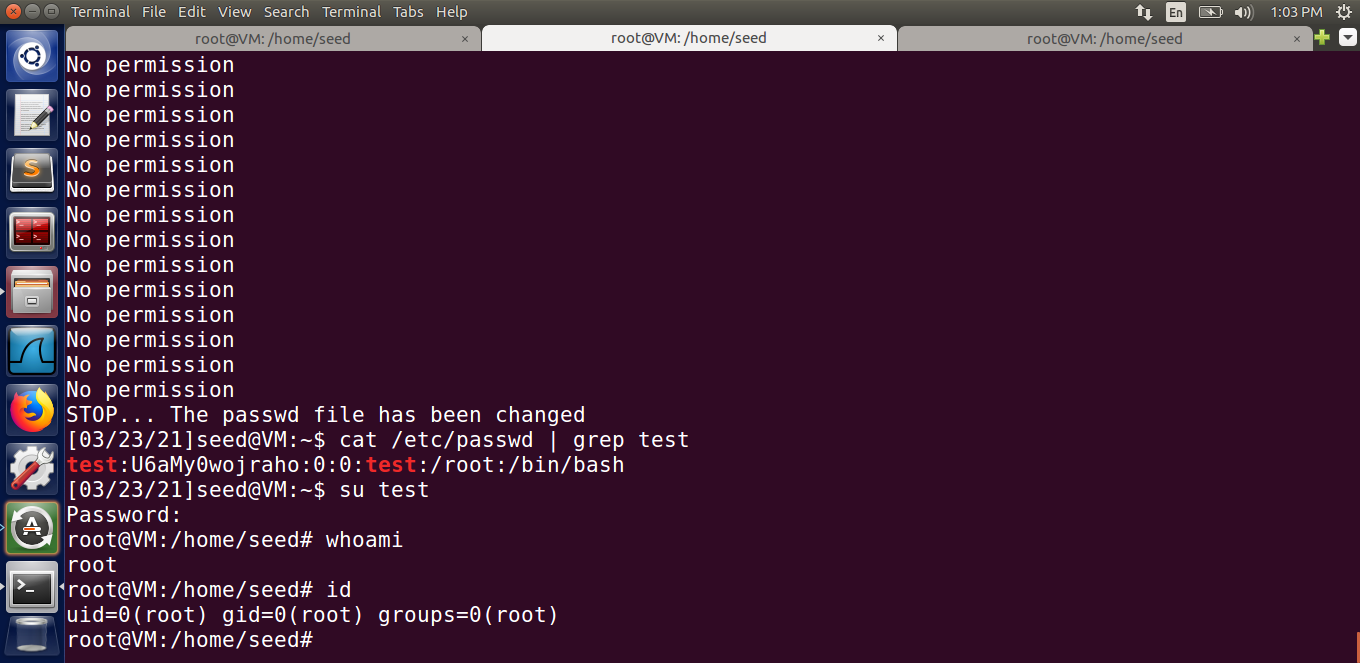
**Fig: passwd\_input file**

Next, compile and run the ./attack in one terminal, and then execute target.sh in another terminal.



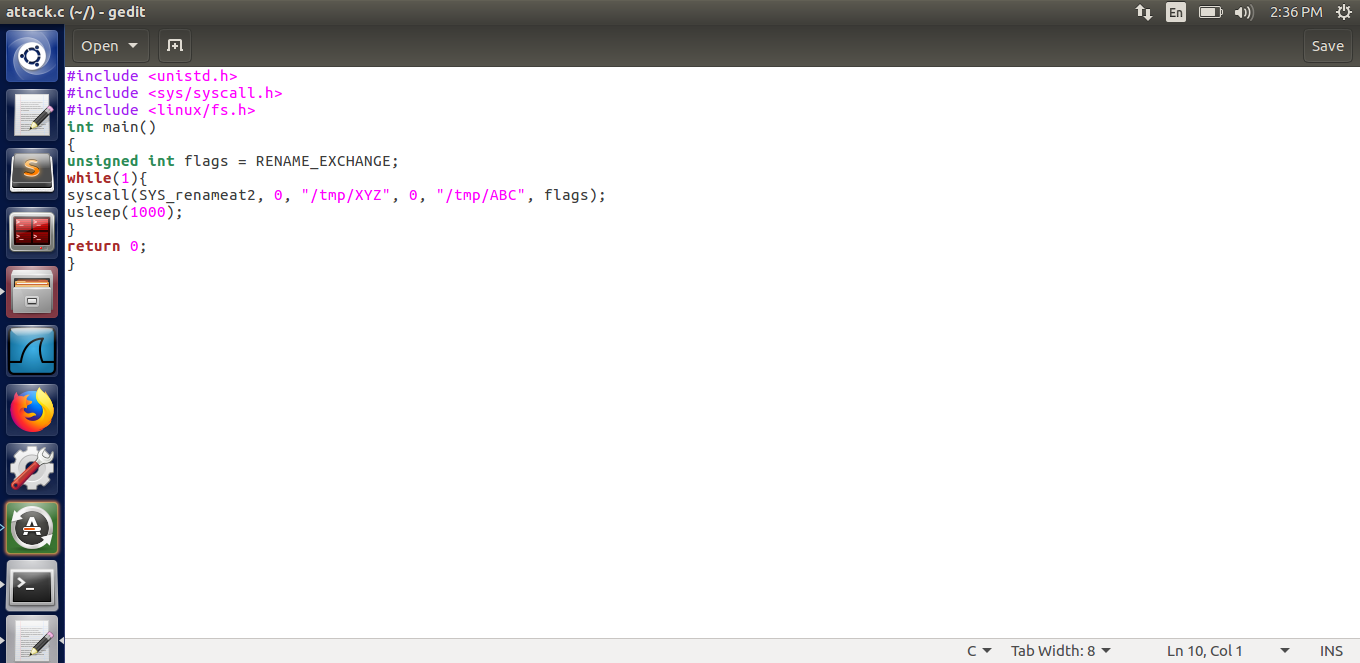


The target.sh terminates when the passwd file has been successfully modified, as defined in the file's condition. Then we look in the file for any new entries. To test it, we switch to the test user and simply type in the password when asked. This shows that our attack was successful, and we were able to escalate our privileges and write to the passwd file by exploiting the race condition. Also, we created a new root user, gaining root access to the system, which can be extremely dangerous.



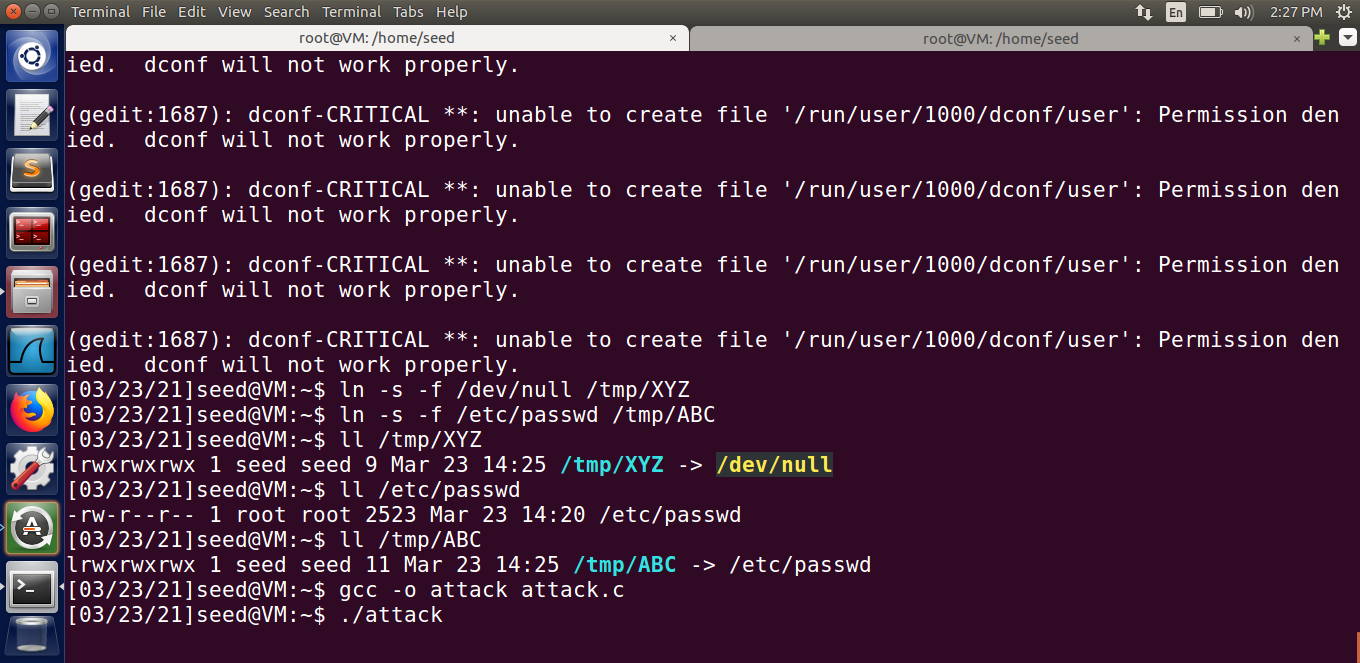
In the attack process, there is a race condition between the unlink and symlink commands. Our attack will always fail if the file in the target.sh is opened after the unlink command and before the symlink command, because the file was generated by the Set-UID program, and the owner will be root. Unlinking will be impossible in this case, and our attack will always fail.

To get around this problem, we would use the ***renameat2()*** system call, which avoids the race condition caused by the unlink and symlink windows. When the RENAME EXCHANGE flag is set, the renameat2() function switches the symbolic links. This swapping will assist in making /tmp/XYZ point to /dev/null at times and /etc/passwd at other times. The race condition attack will be successful if it points to /etc/passwd after the access check and before the file open.

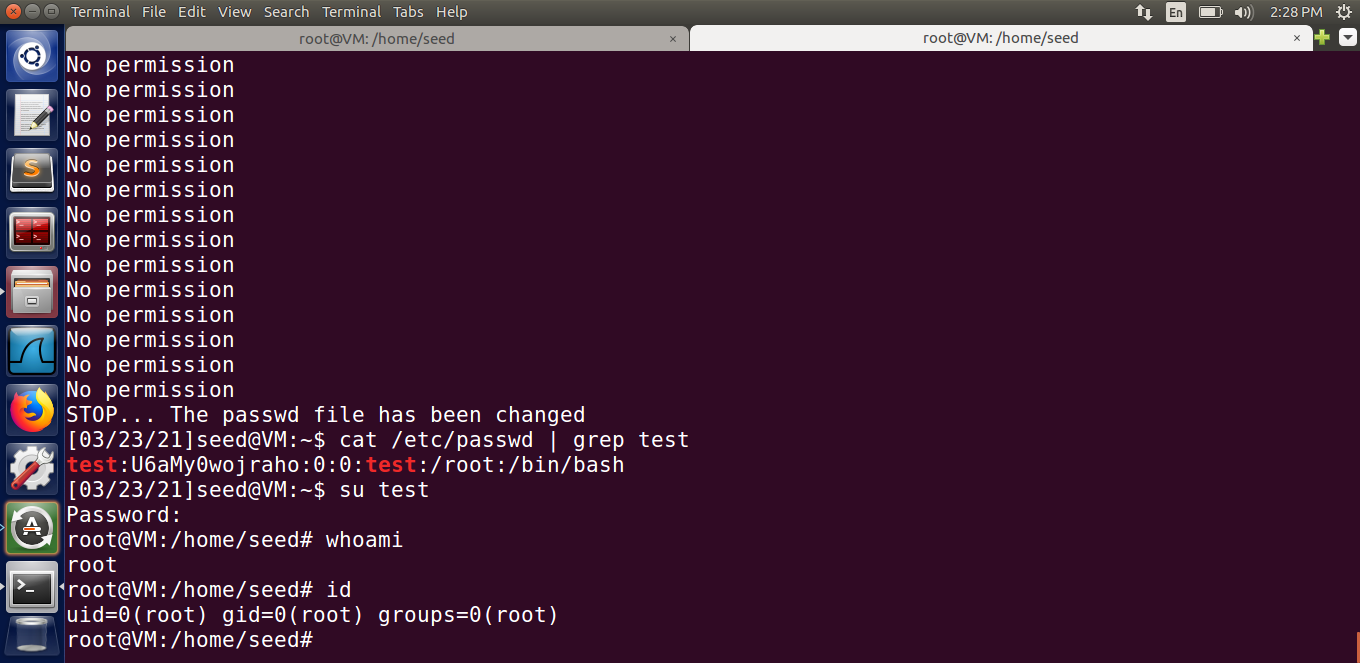


**Fig: Updated attack.c file**

We ***set the symbolic links so that /tmp/XYZ points to /dev/null and /tmp/ABC points to /etc/passwd***. ***The renameat2() system call swaps /tmp/XYZ to point to /etc/passwd and /tmp/ABC to point to /dev/null because of this***. This continues indefinitely, and the attack is ***successful only when /tmp/XYZ points to /etc/passwd between the access check and file open.***

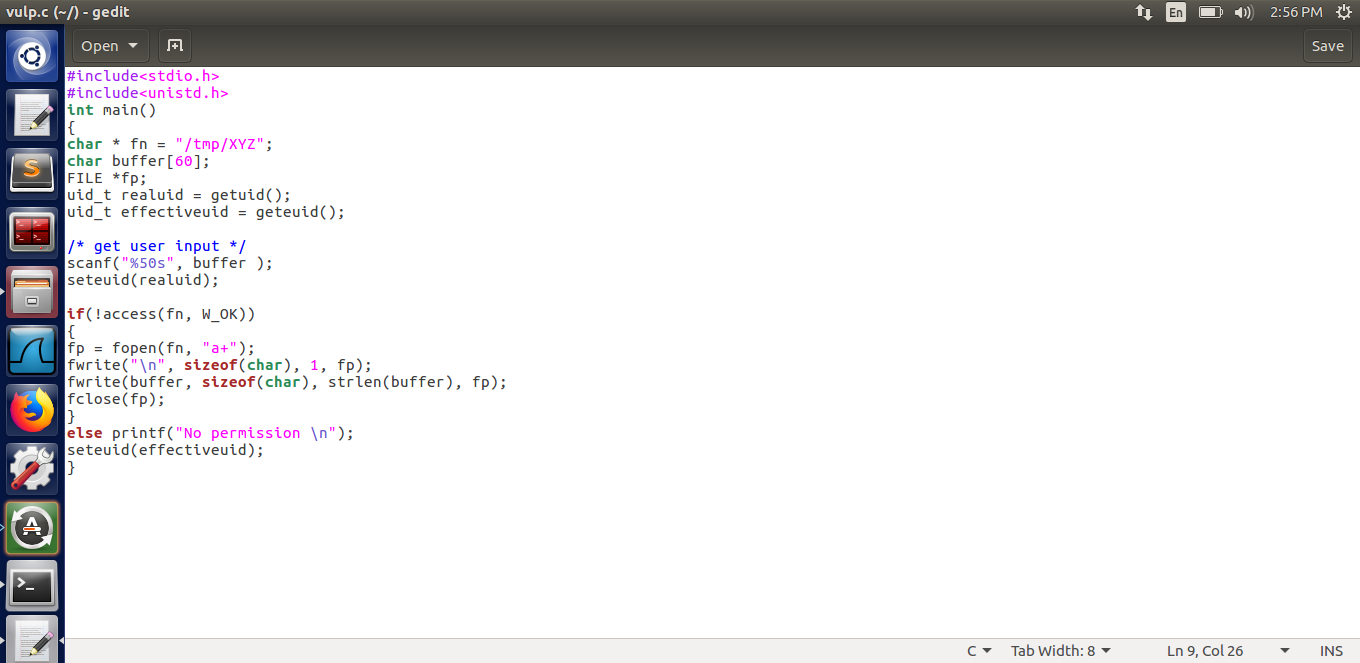


We run the target.sh, which is looping the privileged program, to see if it stops when the stopping condition is met, which is the modification of the passwd file. This indicates that the attack was successful, and we can now rename files using the renameat2() system call, avoiding the race condition caused by the unlink and symlink windows and thus ensuring the success of our race condition attack. The below screenshots show the successful attack using the updated attack.c program.

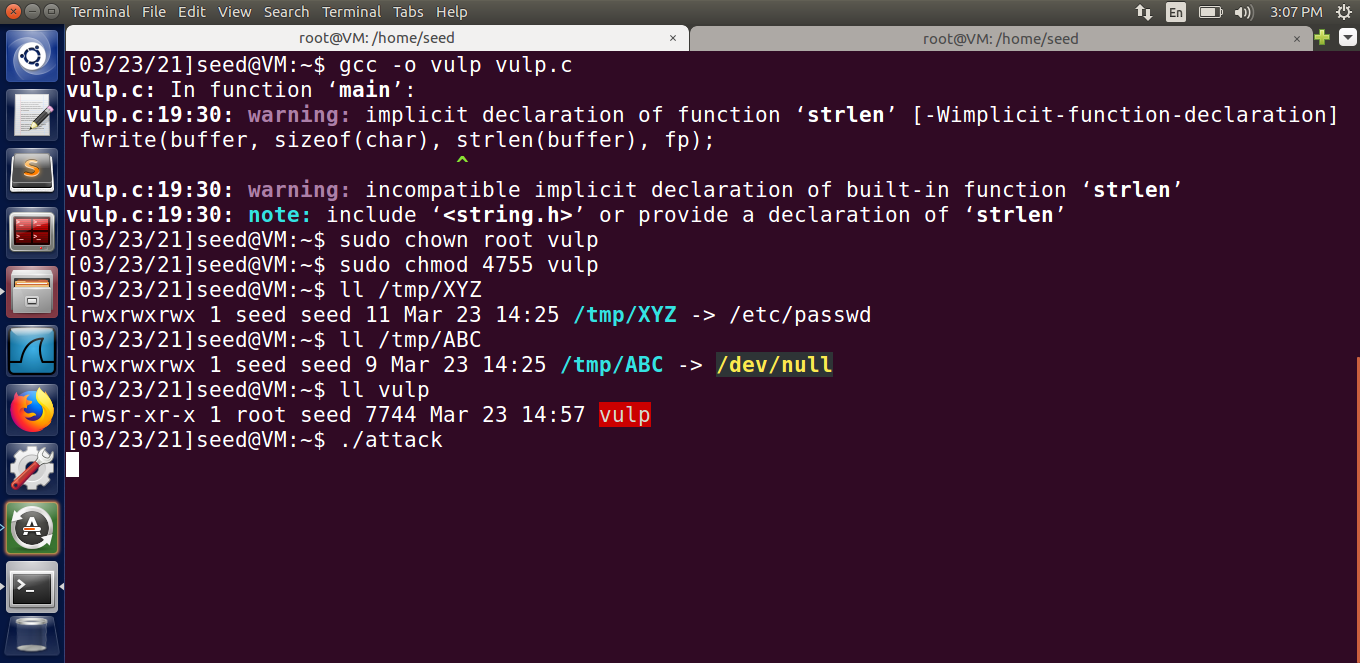


**Task 3: Countermeasure: Applying the Principle of Least Privilege**

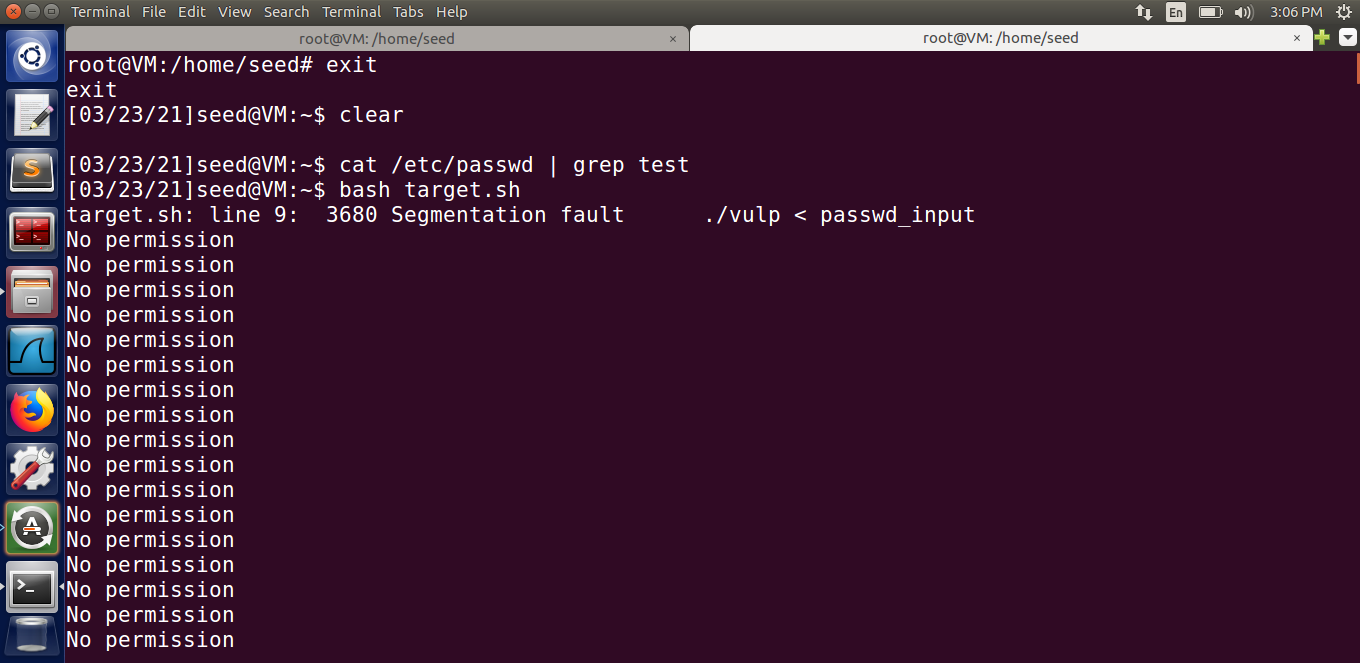
We make changes to the vulnerable program to ensure that the principle of least privilege is followed. We use the program's real UID and effective UID. We set the effective UID to the same as the real UID before checking for access. This drops the privileges, and at the end of the program, we restore the privileges by changing the effective UID to the same as before. Next, compile the program and make it a root owned Set-UID program.

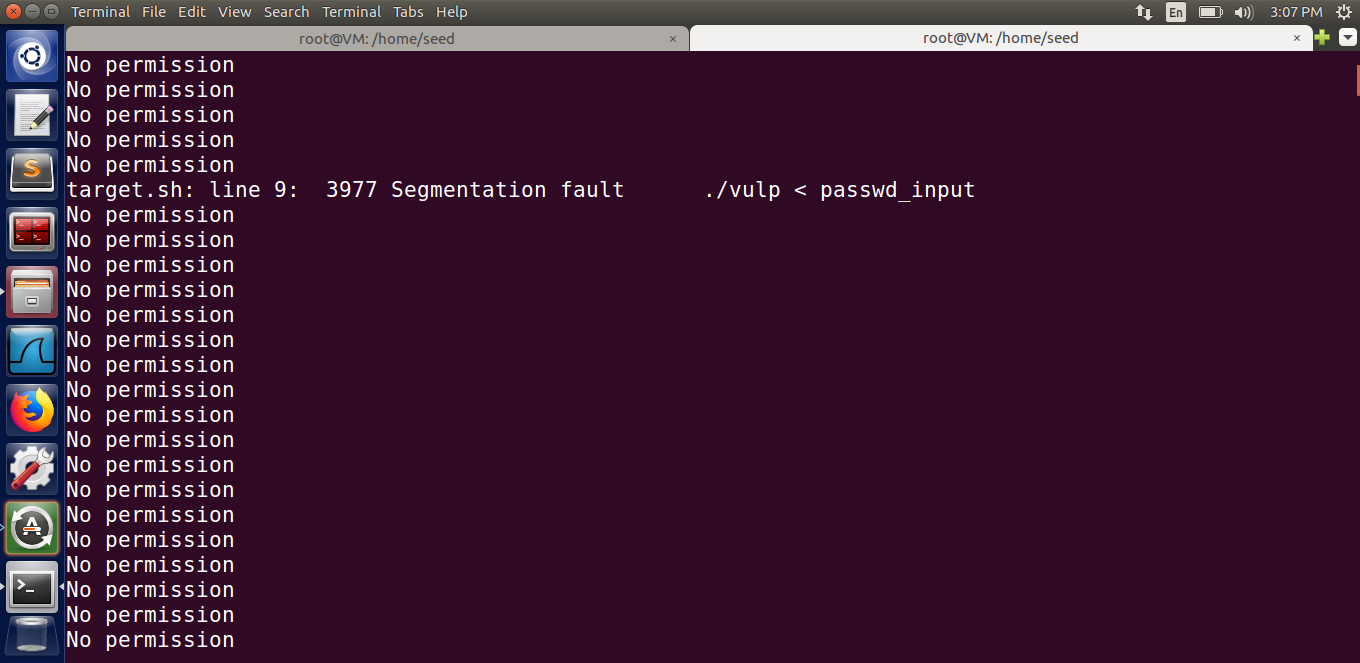


**Fig: Updated vulp.c file**



The attack is not successful, as shown in the screenshot below. Following the principle of least privilege, the Set-UID program's privilege is temporarily revoked because the effective user ID is the same as the real user ID. As a result, the vulnerable program would no longer be vulnerable. This is because the fopen command, which previously used the effective user ID to open the privileged file /etc/passwd, no longer has access to the file because the effective user ID is that of the seed, who does not have the privilege to open it. We reject the extra privileges that are not needed by setting the effective user id to the real user id at the time of execution.

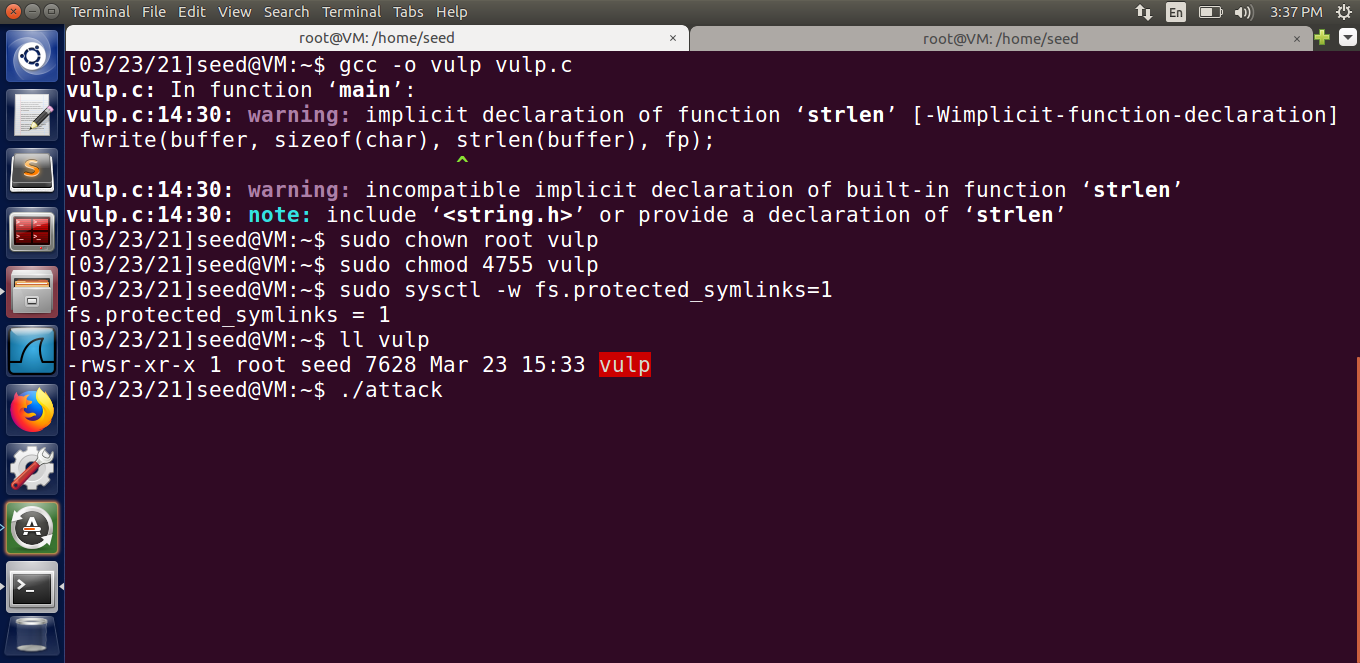




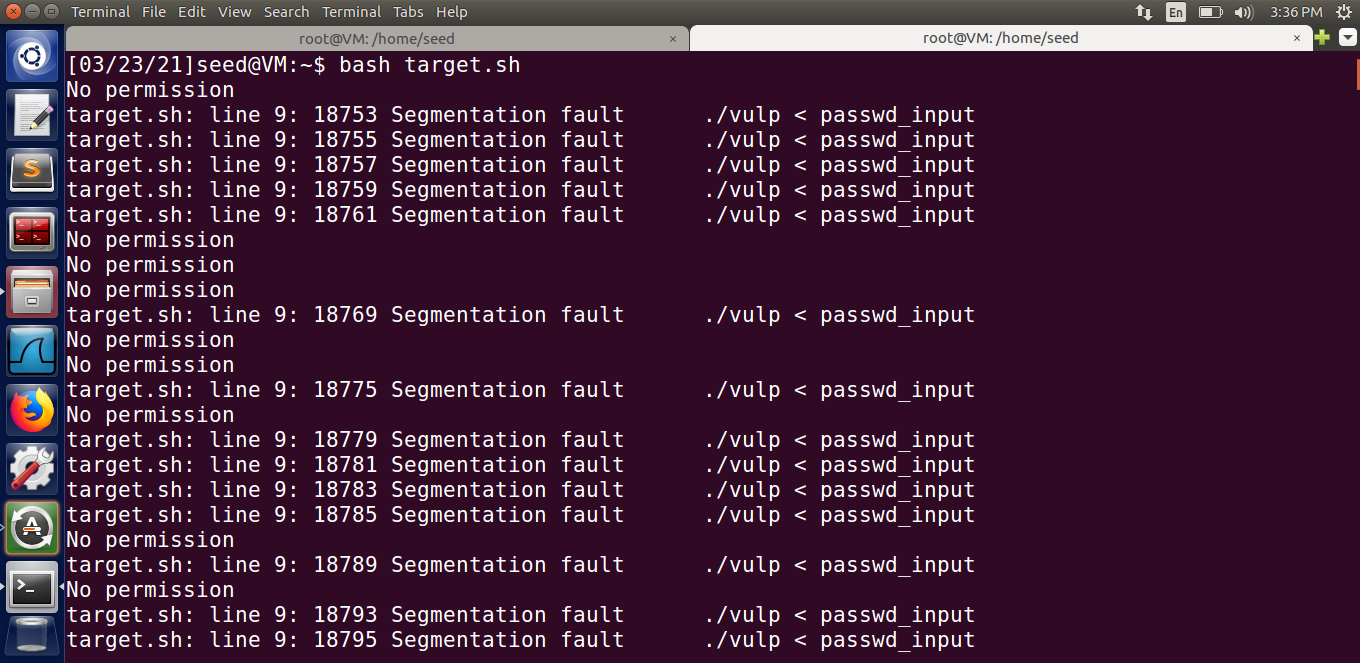
**Fig: Segmentation fault**

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

We start by editing the program from the previous step, deleting the least privilege conditions and re-vulnerabilizing it. We recompile the program after the modifications and make it a Set-UID root program. After that, we allow Ubuntu's built-in race condition protection



We run the attack program and the target.sh program in two different terminals to see if we can still carry out the attack with this condition enabled. The attack is unsuccessful, as can be seen in the screenshot below, and we get Segmentation Fault and No permission in the output. This continues to run in a loop, indicating that our attack was not successful.



**Fig: Attack failed**

1. ***How does this protection scheme work?***

The TOCTTOU race condition vulnerability was exploited in this case, and it included symbolic links in the /tmp directory. As a result, prohibiting programs from following symbolic links under specific conditions may be a way to circumvent this flaw. That is precisely what this built-in prevention method accomplishes. When the sticky symlink prevention option is turned on, symbolic links within a sticky world-writable directory can only be followed if the symlink's owner matches either the follower or the directory owner. Because the program was run with root privileges and the /tmp directory is also owned by root, it will not be able to follow any symbolic links that were not created by root. In our case, the perpetrator, who was also the seed account, formed the symbolic link. As a result, this link will not be pursued, resulting in a crash, as seen. As a result, even though the program has a race condition vulnerability, other users will be unable to access protected files.

1. ***What are the limitations of this scheme?***

This safeguarding scheme does not prevent the race condition from occurring, but it does prevent it from causing harm. As a result, it is a good access control mechanism that allowed only the intended users to access the files but did not prevent the race condition from occurring. This protection only applies to word-writable sticky directories like /tmp, and not to other kinds of directories.