

## Race Condition Vulnerability

### Setup:

We turn off the countermeasures for race conditions and make the compiled vulp.c file a root set-UID program. When we go to run vulp thereafter, the uid and eid will be different because we're not running it as a root user.

```
[02/09/21]seed@VM:~/.../Labsetup$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
[02/09/21]seed@VM:~/.../Labsetup$ sudo sysctl fs.protected_regular=0
fs.protected_regular = 0
[02/09/21]seed@VM:~/.../Labsetup$ gcc -o vulp vulp.c
[02/09/21]seed@VM:~/.../Labsetup$ sudo chown root vulp
[02/09/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 vulp
[02/09/21]seed@VM:~/.../Labsetup$ ls
target_process.sh  vulp  vulp.c
```

### Task 1:

After using *sudo su* to become a root user, I used *cat* on the *passwd* file and *grep* to specify that I wanted data inside that contained the string "test". Nothing was returned so I know there is no user called test. Then, I ran *gedit* to open the *passwd* file and added a line *test:U6aMy0wojraho:0:0:test:/root:/bin/bash* to the end. This creates a user "test" who does not have a password to login.

```
root@VM:/home/seed/Downloads/Lab3/Labsetup# cat /etc/passwd
| grep test
root@VM:/home/seed/Downloads/Lab3/Labsetup# gedit /etc/passwd
```

I used the same *cat|grep* command as before and the output indicated that there was now a user named test in the *passwd* file. I further confirmed the existence of the user test by signing out of the root account and doing *su test*, and I was granted access to the test account without typing in a password. This is because the line that we added to the *passwd* file sets test's password to be null, meaning test doesn't have a password. As seen in the screenshot, the test user is also a root account. Finally, I deleted the line in */etc/passwd* so that when we try the race condition attack, the attack program can write-in the test user.

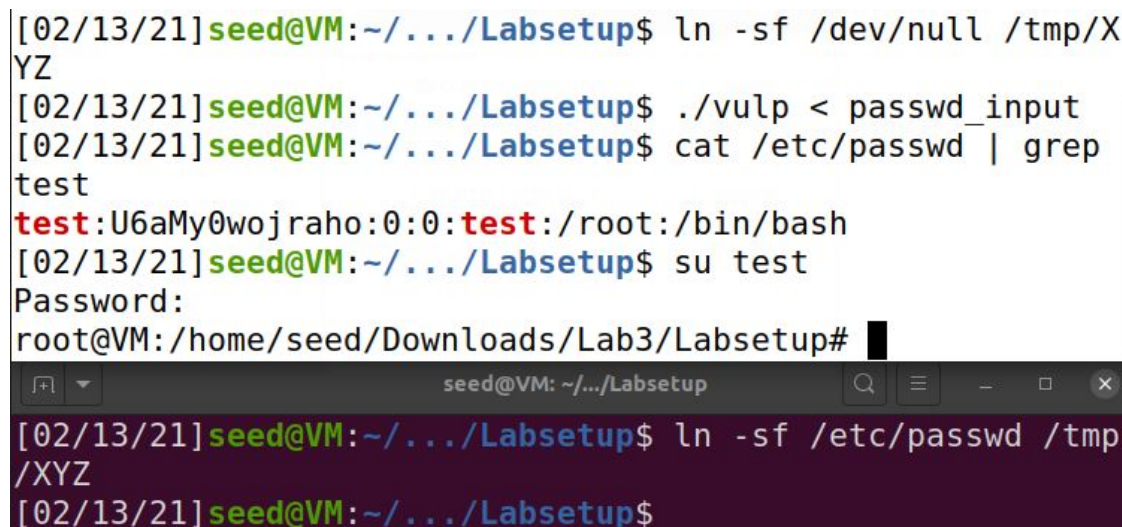
```
root@VM:/home/seed/Downloads/Lab3/Labsetup# cat /etc/passwd
| grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
root@VM:/home/seed/Downloads/Lab3/Labsetup# su test
root@VM:/home/seed/Downloads/Lab3/Labsetup# exit
exit
root@VM:/home/seed/Downloads/Lab3/Labsetup# exit
exit
[02/09/21]seed@VM:~/.../Labsetup$ su test
Password:
root@VM:/home/seed/Downloads/Lab3/Labsetup#
```

## Task 2:

### Part A:

For this part, we have a vulnerable process, vulp, that writes to an arbitrary file /tmp/XYZ. Before writing, vulp checks if the user who's running vulp has the privilege to write to /tmp/XYZ. So, our first step is to link /tmp/XYZ to /dev/null. This way, when vulp checks if we have permission to write to /tmp/XYZ, it actually checks if we have permission to write to /dev/null, which we do. Then, we put the specifications of our test user in a file called passwd\_input.

Using the *ln* command I make /tmp/XYZ point to /dev/null. Then, I run vulp and pass in passwd\_input. Within the ten second window, I open another terminal and use the *ln* command to change /tmp/XYZ to point to /etc/passwd. At this point, vulp thinks we're writing to /dev/null so it lets us write, but since we changed /tmp/XYZ to link to /etc/passwd, we actually write to the passwd file. Then, when I *cat|grep test* on the passwd file, we see that our test user has been added, and we can use *su test* to sign in as this user.



```
[02/13/21] seed@VM:~/.../Labsetup$ ln -sf /dev/null /tmp/XYZ
[02/13/21] seed@VM:~/.../Labsetup$ ./vulp < passwd_input
[02/13/21] seed@VM:~/.../Labsetup$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/13/21] seed@VM:~/.../Labsetup$ su test
Password:
root@VM:/home/seed/Downloads/Lab3/Labsetup#
```

### Part B:

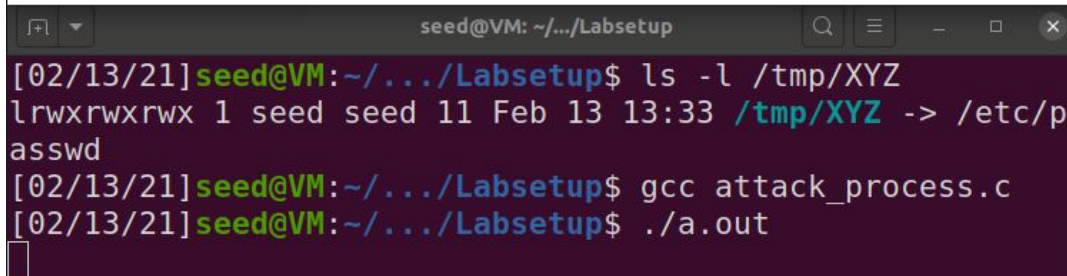
For part B, we introduce two new files: target\_process.sh and attack\_process.c. The bash script contains a loop to run vulp and pass in passwd\_input. The loop stops when the passwd file's timestamp is updated, meaning /etc/passwd has been changed. Our attack\_process and vulp both deal with /tmp/XYZ. While vulp tries to write its input to the XYZ file, attack\_process does the job of the *ln* commands from part A using *unlink* and *symlink*.

First, attack\_process links /tmp/XYZ to /dev/null and sleeps for ten seconds. In this time, we hope the operating system runs vulp and checks if we have permission to write to /dev/null. After that, we want the OS to switch back to attack\_process so the program can change /tmp/XYZ to point to /etc/passwd. Then, when the OS returns to vulp, it should write to the passwd file.

A message of no permission indicates that attack\_process may have changed the links before vulp got to check whether or not we have permission to write to the file. In other words, attack\_process runs completely, unlinking and linking everything, then vulp runs and checks if we have permission to write to /etc/passwd and tells us we don't.

For this task we had to remove the sleep in vulp, recompile it, and make it a root set-UID program again. Then, in one terminal we run our attack\_process, which infinitely unlinks and links /tmp/XYZ to /dev/null and /etc/passwd. In the other terminal we run the bash script, which in turn runs vulp. I was having issues with /tmp/XYZ becoming a root program and thus the attack never succeeded, so before running attack\_process I used a *chown* command to make /tmp/XYZ's owner seed and then used *ls -l* to verify that it was seed-owned.

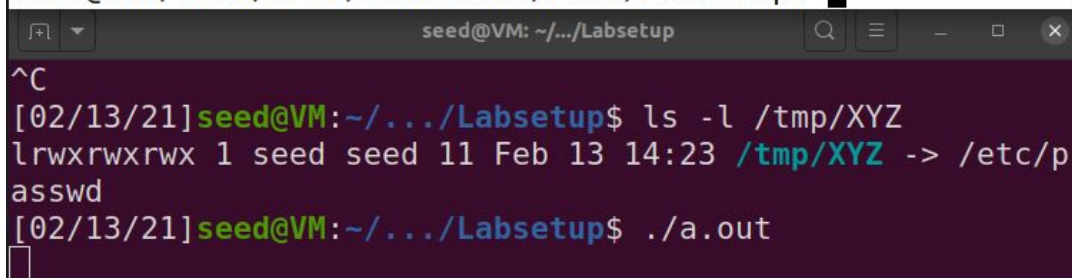
```
[02/13/21] seed@VM: ~/.../Labsetup$ gcc -o vulp vulp.c
[02/13/21] seed@VM: ~/.../Labsetup$ sudo chown root vulp
[02/13/21] seed@VM: ~/.../Labsetup$ sudo chmod 4755 vulp
[02/13/21] seed@VM: ~/.../Labsetup$ ls -l vulp
-rwsr-xr-x 1 root seed 17016 Feb 13 13:39 vulp
[02/13/21] seed@VM: ~/.../Labsetup$ bash target_process.sh
```



The terminal window shows the user 'seed' at 'VM' in the directory '~/.../Labsetup'. They compile 'vulp.c' into 'vulp', then use 'sudo chown root vulp' and 'sudo chmod 4755 vulp' to make it a root set-UID program. They then run 'ls -l vulp' which shows the file is owned by 'root' and 'seed' with permissions '-rwsr-xr-x'. Finally, they run 'bash target\_process.sh'.

After a series of messages saying “no permission”, the bash script stopped and indicated that the attack was successful. To verify this, I used a *cat|grep* command on /etc/passwd and the output showed two users named test. This is because I never deleted the test user from part A. But, the second test user is a direct result of the attack executed here. I then logged into the test user account successfully.

```
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
[02/13/21] seed@VM: ~/.../Labsetup$ cat /etc/passwd | grep
test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/13/21] seed@VM: ~/.../Labsetup$ su test
Password:
root@VM: /home/seed/Downloads/Lab3/Labsetup#
```



The terminal window shows the output of the 'bash target\_process.sh' script. It displays four 'No permission' messages, followed by 'STOP... The passwd file has been changed'. Then, the user runs 'cat /etc/passwd | grep test', which shows two entries for 'test' users. Finally, the user runs 'su test' and enters the password, successfully logging into the 'test' user account as 'root@VM'.

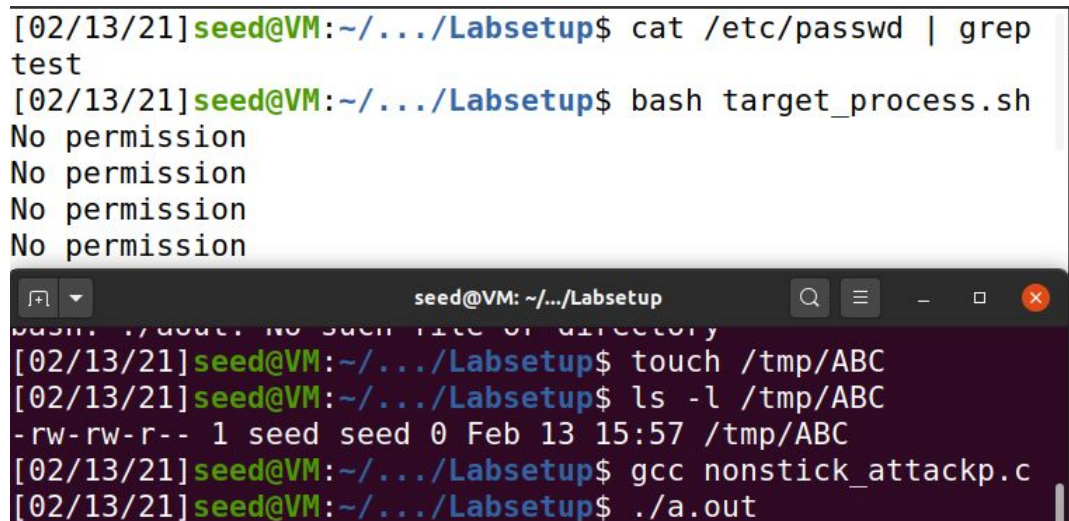


### Part C:

In part B, the owner of /tmp/XYZ would sometimes change to root and the attack would fail. This is because during the time it takes to unlink and link again within the attack\_process, the OS switched and began running vulp. Vulp then created /tmp/XYZ as a root-owned program and the sticky bit on /tmp/XYZ prevented it from being linked by anyone who isn't the root user. To fix this, I can use renameat2() to make a system call and simultaneously swap links between two files. To achieve this, I create a new arbitrary file /tmp/ABC which I will use to swap.

I also create a new attacking process titled nonstick\_attackp. This attack starts by setting up a link from /tmp/XYZ to /dev/null and another link from /tmp/ABC to /etc/passwd. Then, nonstick\_attackp infinitely swaps where /tmp/XYZ and /tmp/ABC point to; /tmp/XYZ will take on /tmp/ABC's link to /etc/passwd and /tmp/ABC will take on the link to /dev/null. This attack program will run until the swap makes /tmp/XYZ point to /etc/passwd after vulp makes it's access check (to check our permissions) and before it opens XYZ to write to.

I deleted the test users from part A and B and thus cat|grep didn't give any output. Then, in another terminal, I created /tmp/ABC and double checked it's user is seed. I compiled nonstick\_attackp.c and ran it, and I ran the bash script in the original terminal. This attack was a lot faster than the ones in parts A and B, which I assume is because the swapping is faster and guarantees that we will eventually link things such that we can write to /etc/passwd.



```
[02/13/21] seed@VM: ~/.../Labsetup$ cat /etc/passwd | grep
test
[02/13/21] seed@VM: ~/.../Labsetup$ bash target_process.sh
No permission
No permission
No permission
No permission
[02/13/21] seed@VM: ~/.../Labsetup$ touch /tmp/ABC
[02/13/21] seed@VM: ~/.../Labsetup$ ls -l /tmp/ABC
-rw-rw-r-- 1 seed seed 0 Feb 13 15:57 /tmp/ABC
[02/13/21] seed@VM: ~/.../Labsetup$ gcc nonstick_attackp.c
[02/13/21] seed@VM: ~/.../Labsetup$ ./a.out
```

The following screenshot confirms the existence of one test user with root privileges whom I can log into.

```
No permission
No permission
STOP... The passwd file has been changed
[02/13/21] seed@VM: ~/.../Labsetup$ cat /etc/passwd | grep
test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[02/13/21] seed@VM: ~/.../Labsetup$ su test
Password:
root@VM: /home/seed/Downloads/Lab3/Labsetup#
```

### Task 3:

#### Part A:

The vulnerable program tried to use the access function to check if the user could write to the given file, but this creates a vulnerability that was exploited in the previous task. Instead, to fix the vulnerability, we can use seteuid() to change the effective UID to be the same as the real UID before we call access() in vulp.c. At the end of the program, we set the effective UID back to its original value.

I recompile vulp.c and make it a root-owned set-UID program. Then, even though vulp is root-owned, the program changes its effective UID to that of the real UID, which is not root, and the program runs without root-privilege. The access check will always fail as long as we are not the root user.

```
[02/13/21]seed@VM:~/.../Labsetup$ gcc -o vulp vulp.c
[02/13/21]seed@VM:~/.../Labsetup$ sudo chown root vulp
[02/13/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 vulp
[02/13/21]seed@VM:~/.../Labsetup$ bash target_process.sh
No permission
No permission
No permission
No permission
```

#### Part B:

Without having to worry about UID's, Ubuntu provides its own countermeasure against race condition attacks which we disabled in the setup process. We now re-enable it and change vulp to as it was before, without the seteuid() function so the program is vulnerable again. Then I run the attack.

```
[02/13/21]seed@VM:~/.../Labsetup$ sudo sysctl -w fs.protected_symlinks=1
fs.protected_symlinks = 1
[02/13/21]seed@VM:~/.../Labsetup$ gcc -o vulp vulp.c
[02/13/21]seed@VM:~/.../Labsetup$ sudo chown root vulp
[02/13/21]seed@VM:~/.../Labsetup$ sudo chmod 4755 vulp
[02/13/21]seed@VM:~/.../Labsetup$ bash target_process.sh
No permission
No permission
target_process.sh: line 10: 1232907 Segmentation fault
./vulp < passwd_input
target_process.sh: line 10: 1232909 Segmentation fault
./vulp < passwd_input
```

The protection scheme seems to work by preventing certain links from being functional. Our attack relies on changing symbolic links at a certain time so we can write to the passwd file. However, once we enabled "protected\_symlinks" the attack didn't work and instead we either got "no permission" or a segmentation fault. The protection might check UIDs and owners of the files it wants to link up to see if these things match up. If they don't, then the link is created but the program crashes because the protection blocked whatever action was being performed across the link. Since the protection scheme still creates the links between files with different owners, this means that the actual race condition is not prevented, only the attack of writing to /etc/passwd is prevented.