Race Condition Vulnerability Lab

(CSI 424/524 - Fall 2023)

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. Detailed coverage of the race condition attack can be found in the following:

- Chapter 7 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du.

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

2 Environment Setup

2.1 Turning Off Countermeasures

Ubuntu has 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." Ubuntu 20.04 introduces another security mechanism that prevents the root from writing to the files in `/tmp` that are owned by others. In this lab, we need to disable these protections. You can achieve that using the following commands:

```
// On Ubuntu 20.04, use the following:
$ sudo sysctl -w fs.protected_symlinks=0
$ sudo sysctl fs.protected_regular=0

// On Ubuntu 16.04, use the following:
$ sudo sysctl -w fs.protected_symlinks=0

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

2.2 A Vulnerable Program

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

Listing 1: The vulnerable program (`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)){                    ①
      fp = fopen(fn, "a+");                  ②
      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`. 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 makes `/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 can thus gain the root privilege. The vulnerable program 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
```

3 Task 1: Choosing Our Target

We would like to exploit the race condition vulnerability in the 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.

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

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.

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.

Warning. In the past, some students accidentally emptied the `/etc/passwd` file during the attack (this could be caused by some race condition problems inside the OS kernel). If you lose the password file, you

will not be able to log in again. To avoid this trouble, please make a copy of the original password file or take a snapshot of the VM. This way, you can easily recover from the mishap.

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 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.

4.1 Task 2.A: Simulating a Slow Machine

Let us pretend that the machine is very slow, and there is a 10-second time window between the `access()` and `fopen()` calls. To simulate that, we add a `sleep(10)` between them. The program will look like the following:

```
if (!access(fn, W_OK)) {
     sleep(10);
     fp = fopen(fn, "a+");
     ...
```

With this addition, the `vulp` program (when re-compiled) will pause and yield control to the operating system for 10 seconds. Your job is to manually do something, so when the program resumes after 10 seconds, the program can help you add a root account to the system. Please demonstrate how you would achieve this.

You won't be able to modify the file name `/tmp/XYZ`, because it is hardcoded in the program, but you can use symbolic links to change the meaning of this name. For example, you can 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`. See the following example (the `"f"` option means that if the link exists, remove the old one first):

```
$ ln -sf /dev/null /tmp/XYZ
$ ls -ld /tmp/XYZ
lrwxrwxrwx 1 seed seed 9 Dec 25 22:20 /tmp/XYZ -> /dev/null
```

4.2 Task 2.B: The Real Attack

In the previous task, we have kind of "cheated" by asking the vulnerable program to slow down, so we can launch the attack. This is definitely not a real attack. In this task, we will launch the real attack. Before doing this task, make sure that the `sleep()` statement is removed from the `vulp` program.

The typical strategy in race condition attacks is to run the attack program in parallel to the target program, hoping to be able to do the critical step within that time window. Unfortunately, perfect timing is very hard to achieve, so the success of attack is only probabilistic. The probability of a successful attack might be quite low if the window is small, but we can run the attack many many times. We just need to hit the race condition window once.

Writing the attack program. In the simulated attack, we use the `"ln -s"` command to make/change symbolic links. Now we need to do it in a program. We can use `symlink()` in C to create symbolic

links. Since `Linux` does not allow one to create a link if the link already exists, we need to delete the old link first. The following C code snippet shows how to remove a link and then make `/tmp/XYZ` point to `/etc/passwd`. Please write your attack program.

```
unlink("/tmp/XYZ");
symlink("/etc/passwd","/tmp/XYZ");
```

Running the vulnerable program and monitoring results. Since we need to run the vulnerable program for many times, we will write a program to automate this process. To avoid manually typing an input to the vulnerable program `vulp`, we can use input redirection. Namely, we save our input in a file, and ask `vulp` to get the input from this file using `"vulp < inputFile"`. We can also use pipe (an example will be given later).

It may take a while before our attack can successfully modify the password file, so 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 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.

The following shell script runs the vulnerable program (`vulp`) in a loop, with the input given by the `echo` command (via a pipe). You need to decide what should be the actual input. If the attack is successful, i.e., the `passwd` is modified, the shell script will stop. You do need to be a little bit patient. Normally, you should be able to succeed within 5 minutes.

```
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]    ← Check if /etc/passwd is modified
do
   echo "your input" | ./vulp ← Run the vulnerable program
   new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

Verifying success When your script terminates, test the success of your exploit by logging in as the test user and verifying root privileges. Then terminate the attack program by pressing `Ctrl-C` in the Terminal window in which you started the program.

A Note. If after 10 minutes, your attack is still not successful, you can stop the attack, and check the ownership of the `/tmp/XYZ` file. If the owner of this file becomes root, manually delete this file, and try your attack again, until your attack becomes successful. Please document this observation in your lab report. In Task 2.C, we will explain the reason and provide an improved attack method.

4.3 Task 2.C: An Improved Attack Method

In Task 2.B, if you have done everything correctly, but still could not succeed in the attack, check the ownership of `/tmp/XYZ`. You will find out that `/tmp/XYZ`'s owner has become root (normally, it should be `seed`). If this happens, your attack will never succeed, because your attack program, running with the

`seed` privilege, can no longer remove or `unlink()` it. This is because the `/tmp` folder has a "sticky" bit on, meaning that only the owner of the file can delete the file, even though the folder is world-writable.

In Task 2.B, we let you use the root's privilege to delete `/tmp/XYZ`, and then try your attack again. The undesirable condition happens randomly, so by repeating the attack (with the "help" from root), you will eventually succeed in Task 2.B. Obviously, getting help from root is not a real attack. We would like to get rid of that, and do it without root's help.

The main reason for that undesirable situation is that our attack program has a problem, a race condition problem, the exact problem that we are trying to exploit in the victim program. How ironic! In the past, when we saw that problem, we simply advised students to delete the file and try the attack again. Thanks to one of my students, who was determined to figure out what the problem was. Because of his effort, we finally understand why and have an improved solution.

The main reason for the situation to happen is that the attack program is context switched out right after it removes `/tmp/XYZ` (i.e., `unlink()`), but before it links the name to another file (i.e., `symlink()`. Remember, the action to remove the existing symbolic link and create a new one is not atomic (it involves two separate system calls), so if the context switch occurs in the middle (i.e., right after the removal of `/tmp/XYZ`), and the target `Set-UID` program gets a chance to run its `fopen(fn, "a+")` statement, it will create a new file with root being the owner. After that, your attack program can no longer make changes to `/tmp/XYZ`.

Basically, using the `unlink()` and `symlink()` approach, we have a race condition in our attack program. Therefore, while we are trying to exploit the race condition in the target program, the target program may accidentally "exploit" the race condition in our attack program, defeating our attack.

To solve this problem, we need to make `unlink()` and `symlink()` atomic. Fortunately, there is a system call that allows us to achieve that. More accurately, it allows us to atomically swap two symbolic links. The following program first makes two symbolic links `/tmp/XYZ` and `/tmp/ABC`, and then using the `renameat2` system call to atomically switch them. This allows us to change what `/tmp/XYZ` points to without introducing any race condition.

```
#define _GNU_SOURCE

#include <stdio.h>
#include <unistd.h>
int main()
{
   unsigned int flags = RENAME_EXCHANGE;

   unlink("/tmp/XYZ"); symlink("/dev/null",   "/tmp/XYZ");
   unlink("/tmp/ABC"); symlink("/etc/passwd", "/tmp/ABC");

   renameat2(0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
   return 0;
}
```

Tasks. Please revise your attack program using this new strategy, and try your attack again. If everything is done correctly, your attack should be able to succeed.

5 Task 3: Countermeasures

5.1 Task 3.A: 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 `seteuid` 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.

5.2 Task 3.B: 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 16.04 and 20.04, use the following command:
$ sudo sysctl -w fs.protected_symlinks=1

// On Ubuntu 12.04, use the following command:
$ sudo sysctl -w kernel.yama.protected_sticky_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?