Race Condition Vulnerability Lab

1 Lab 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)){                  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 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
```

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.

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

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 < 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 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"
```

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 `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.

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?

3 Guidelines

Detailed guidelines can be found in Chapter 7 of the SEED book, *Computer Security: A Hands-on Approach*, by Wenliang Du. We summarize some of the guidelines in this section.

3.1 Creating Symbolic Links

You can call C function `symlink()` to create symbolic links in your program. 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`:

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

You can also use `Linux` command `"ln -sf"` to create symbolic links. Here the `"f"` option means that if the link exists, remove the old one first. The implementation of the `"ln"` command actually uses `unlink()` and `symlink()`.

3.2 An Undesirable Situation

While testing your attack program, you may find out that `/tmp/XYZ` is created with root being its owner. If this happens, you have lost the "race", i.e., the file was somehow created by the target program, which has the root privilege. Once that happens, there is no way you can remove this file. 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.

If this happens, you need to adjust your attack strategy, and try it again (of course, after manually removing the file from the root account). The main reason for this to happen is that the attack program is context switched out right after it removes `/tmp/XYZ`, but before it links the name to another file. 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. Think about a strategy that can minimize the chance to get context switched in the middle of that action.

3.3 Warning

In the past, some students accidentally emptied the `/etc/passwd` file during the attacks (we still do not know what has caused that). 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 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.