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.

2 Lab Tasks

Before working on the lab tasks, you should read the Guidelines in Section 3.

2.1 Initial setup

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 command:

```
$ 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>
#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");
    }
```

In this lab, **you need to compile the above program using the root account and make it as a set-uid program**. This program appends a string of user input to the end of a temporary file `/tmp/XYZ`. Since the code runs with the root privilege, it carefully checks whether the real user actually has the access permission to the file `/tmp/XYZ`; that is the purpose of the `access()` call. Once the program has made sure that the real user indeed has the right, the program opens the file and writes the user input into the file. **In this lab, you can use the seed account to create a file named XYZ**.

It appears that the program does not have any problem at the first look. However, there is a race condition vulnerability in this program: due to the window (the simulated delay) 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 `/etc/shadow`, the attacker can cause the user input to be appended to `/etc/shadow` (note that the program runs with the root privilege, and can therefore overwrite any file).

2.3 Task 1: Exploit the Race Condition Vulnerabilities

You need to exploit the race condition vulnerability in the above `Set-UID` program. More specifically, you need to add a line "cs532-lab" to the end of $/etc/shadow$ by using the race. Please also explain why your attack can success.

Actually, if you are able to modify the shadow file, you can use the same attack to modify the /etc/passwd file. As a result, you are be able to create any kind of accounts for the OS.

2.4 Task 2: Protection Mechanism A: Repeating

Getting rid of race conditions is not easy, because the check-and-use pattern is often necessary in programs. Instead of removing race conditions, we can actually add more race conditions, such that to compromise the security of the program, attackers need to win all these race conditions. If these race conditions are designed properly, we can exponentially reduce the winning probability for attackers. The basic idea is to repeat `access()` and `fopen()` for several times.

Please use this strategy to modify the vulnerable program, and repeat your attack. Report how difficult it is to succeed, if you can still succeed.

2.5 Task 3: Protection Mechanism B: 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 explanation.

2.6 Task 4: Protection Mechanism C: `Ubuntu`'s Built-in Scheme

As we mentioned in the initial setup, comes with a built-in protection scheme against race condition attacks. In this task, you need to turn the protection back on using the following command:

```
$ sudo sysctl -w fs.protected_symlinks=1
```

In your report, please describe your observations. Please also explain why does this protection scheme work?

3 Guidelines

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()`.

If you are using bash script to switch symbolic links, you can use following code:

```
#!/bin/bash

while [ 1 -le 1 ]
do
        ln -sf /tmp/myfile /tmp/XYZ
        ln -sf /etc/shadow /tmp/XYZ
done
```

This bash script keeps switching the symbolic for file XYZ between myfle and shadow file. Here, my file is the a file created by the seed account.

3.2 Improving success rate

The most critical step (i.e., pointing the link to our target file) of a race-condition attack must occur within the window between check and use; namely between the `access` and the `fopen` calls in `vulp.c`. Since we cannot modify the vulnerable program, the only thing that we can do is to run our attacking program in parallel with the target program, hoping that the change of the link does occur 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 (Hints: 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 `vulp`, you can use redirection. Namely, you type your input in a file, and then redirect this file when you run `vulp`. For example, you can use the following: `vulp < FILE`.

3.3 Knowing whether the attack is successful

Since the user does not have the read permission for accessing `/etc/shadow`, there is no way of knowing if it was modified. The only way that is possible is to see its time stamps. Also it would be better if we stop the attack once the entries are added to the respective files. The following shell script checks if the time stamps of `/etc/shadow` has been changed. It prints a message once the change is noticed.

```
#!/bin/sh

old=`ls -l /etc/shadow`
new=`ls -l /etc/shadow`
while [ "$old" = "$new" ]
do
    new=`ls -l /etc/shadow`
done
echo "STOP... The shadow file has been changed"
```

Copyright