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.

2 Lab Tasks

2.1 Initial setup

`Ubuntu` 10.10 and later comes with an 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 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>


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");
}
```

This is part of a `Set-UID` program (owned by `root`); it 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.

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/passwd`, the attacker can cause the user input to be appended to `/etc/passwd` (note that the program runs with the root privilege, and can therefore overwrite any file).

2.3 Task1: Targeting `/etc/passwd`

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 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.

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: Exploit the Race Condition Vulnerabilities

You need to exploit the race condition vulnerability in the above `Set-UID` program. The ultimate goal is to gain the root privilege.

2.4.1 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`.

2.4.2 Knowing whether the attack is successful

Since the user does not have the read permission for accessing `/etc/passwd`, 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/passwd` has been changed. It prints a message once the change is noticed.

```
#!/bin/sh

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

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

```
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=1
```

In your report, please describe your observations. Please also explain the followings: (1) Why does this protection scheme work? (2) Is this a good protection? Why or why not? (3) What are the limitations of this scheme?

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()`.

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 is somehow created by the root. 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 shadow file, you will not be able to login again. To avoid this trouble, please make a copy of the original password file.