Race Condition Vulnerability Lab

The development of this document is/was funded by three grants from the US National Science Foundation: Awards No. 0231122 and 0618680 from TUES/CCLI and Award No. 1017771 from Trustworthy Computing.

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

2.1 Initial setup

You can execute the lab tasks using our pre-built `Ubuntu` virtual machines. If you are using our `Ubuntu` 9.11 VM, you can skip this initial setup step. If you are using our `Ubuntu` 11.04 VM, you need to read the following. `Ubuntu` 11.04 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 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.

```
/*  vulp.c  */

#include <stdio.h>
#include<unistd.h>

#define DELAY 10000
```

```
int main()
{
   char * fn = "/tmp/XYZ";
   char buffer[60];
   FILE *fp;
   long int  i;

   /* get user input */
   scanf("%50s", buffer );

   if(!access(fn, W_OK)){
       /* simulating delay */
        for (i=0; i < DELAY; i++){
           int a = i^2;
        }

        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/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 achieve the followings:

1. Overwrite any file that belongs to `root`.
2. Gain root privileges; namely, you should be able to do anything that root can do.

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 `open()` for several times; at each time, we open the file, and at the end, we check whether the same file is opened by checking their `i-nodes` (they should be the same).

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

This task is only for those who use our `Ubuntu` 11.04 VM. As we mentioned in the initial setup, `Ubuntu` 11.04 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 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 Two Potential Targets

There are possibly many ways to exploit the race condition vulnerability in `vulp.c`. One way is to use the vulnerability to append some information to both `/etc/passwd` and `/etc/shadow`. These two files are used by `Unix` operating systems to authenticate users. If attackers can add information to these two files, they essentially have the power to create new users, including super-users (by letting uid to be zero).

The `/etc/passwd` file is the authentication database for a Unix machine. It contains basic user attributes. This is an ASCII file that contains an entry for each user. Each entry defines the basic attributes applied to a user. When you use the `mkuser` command to add a user to your system, the command updates the `/etc/passwd` file.

The file `/etc/passwd` has to be world readable, because many application programs need to access user attributes, such as user-names, home directories, etc. Saving an encrypted password in that file would

mean that anyone with access to the machine could use password cracking programs (such as `crack`) to break into the accounts of others. To fix this problem, the shadow password system was created. The `/etc/passwd` file in the shadow system is world-readable but does not contain the encrypted passwords. Another file, `/etc/shadow`, which is readable only by root contains the passwords.

To find out what strings to add to these two files, run `mkuser`, and see what are added to these files. For example, the followings are what have been added to these files after creating a new user called `smith`:

```
/etc/passwd:
-------------
  smith:x:1000:1000:Joe Smith,,,:/home/smith:/bin/bash

/etc/shadow:
-------------
  smith:*1*Srdssdsdi*M4sdabPasdsdsdasdsdasdY/:13450:0:99999:7:::
```

The third column in the file `/etc/passwd` denotes the UID of the user. Because `smith` account is a regular user account, its value 1000 is nothing special. If we change this entry to 0, `smith` now becomes `root`.

3.2 Creating symbolic links

You can manually create symbolic links using `"ln -s"`. You can also 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");
```

3.3 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`.

In the program `vulp.c`, we intentionally added a DELAY parameter in the program. This is intended to make your attack easier. Once you have succeeded in your attacks, gradually reduce the value for DELAY. When DELAY becomes zero, how much longer does it take you to succeed?

3.4 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"
```

3.5 Troubleshooting

While testing the program, due to untimely killing of the attack program, `/tmp/XYZ` may get into an unstable state. When this happens the OS automatically makes it a normal file with root as its owner. If this happens, the file has to be deleted and the attack has to be restarted.

3.6 Warning

In the past, some students accidentally emptied the `/etc/shadow` 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 shadow file.