Justin Gallagher

CSC 482

Race Condition Lab

A race condition occurs when multiple processes access and manipulate the same data concurrently, and the outcome of the execution depends on the 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, we will exploit a race condition vulnerability to both read and modify the contents of a program's memory. In addition to the attacks, we will also look at different protection schemes that can be used to counter the race-condition attacks.

Task 0: Initial Setup and The Vulnerable Program

Initially, Ubuntu comes with built-in protection against race conditions. We will need to disable this feature in order to proceed with the lab. To do this, we will use the command provided by the lab, `sudo sysctl -w fs.protected_symlinks=0`. Next, we download the program 'vulp.c' which will be the vulnerable program that we exploit using race conditions. 'vulp.c' 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, 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. If the real user ID indeed has the right, the program opens the file and appends the user input to the file. Although it may not seem like it at first, a race condition arises between the moments when the file is checked and when the file is used in the program. There is a possible chance that the program will check one file but use a completely different one. If we can make a symbolic link from /tmp/XYZ to a protected file, we could potentially gain root access. That is the goal in this lab. We end this task by ensuring the 'vulp.c' program is compiled as a root owned, Set-UID program. The commands to do this are listed below.

```
gcc vulp.c -o vulp
sudo chown root vulp
sudo chmod 4755 vulp
```

Task 1: Choosing our Target

In this task, we choose out target. It will be the /etc/passwd file since it is not normally writeable by a normal user. By exploiting the race condition vulnerability, we will add a record to the password file, with the goal of creating a new user 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 `root:x:0:0:root:/root:/bin/bash`. For the root user, the third field which is the user ID has a value zero. When a root user logs in, that value is set to zero and that is what gives that user root capabilities. If we want to create an account with the root privilege, we just need to put a zero in the user ID field. The password field does not hold the actual password but instead it holds a hash value that points to the password. To get the value for a password, we can add a new user in our own system using the `adduser` command, and then get the hash value of our password from the shadow file. There is a magic value used in Ubuntu live CD for a password-less account, and the magic value is `U6aMy0wojraho`. If we put this value in the password field of a user entry, we can bypass the password when prompted. We are given the entry for the user 'test', which is `test:U6aMy0wojraho:0:0:test:/root:/bin/bash`, and the goal is to add it to the password file and then see if we can access it without using a password and see if we have root access. To do this, I start by navigating to the etc directory and find the passwd file. Using a text editor, I copy in the entry for the user at the end of the file. I then check to see if the user is created by checking the contents of the /etc/passwd file. After confiming, I then attempt to log in as 'test' without the use of a password. I find that upon running the command to log in to we do not require a password and we have root privileges. Finally, we delete this user as this only served as a test to see if we could add a user via the password file (deleted to modified version of passwd and copied back in the original). The commands below were used in this task.

```
sudo su root
echo 'test:U6aMy0wojraho:0:0:test:/root:/bin/bash' >> passwd
sudo su seed
cat passwd | grep test
su test
```

Task 2.A: Launching the Race Condition Attack

In this task, we will gain root access via the race condition vulnerability. Making/tmp/XYZ point to the password file must occur within the window between check and use. We cannot modify the vulnerable program, but we can run our attacking program parallel to the target program, hoping to win the race condition. The probability of successful attack is dependent on the time the window is open. To start, I compiled the 'attack.c' program and named it 'attack.' I then wrote the shell script called 'check.sh' that runs the vulnerable program repeatedly with its input coming from 'passwd_input' which contains line after line of the root user entry from task 1. If successful, the shell script will stop and notify us that the password file has been modified. After opening two terminals, I ran the attack program in one and the shell script in the other. I let both processes run for about 5-10min, as the lab recommended. After about the 7-minute mark, I was met with the message "STOP. /etc/passwd has been changed." I then checked to see if the password file had been successfully modified by attempting to access the 'test' user without a password and with root privilege and upon doing so, I was successful. The lab had also noted that if the attack wasn't successful, then you should remove the XYZ file in /tmp but I didn't run into this issue. The commands used in this task are listed below.

```
gcc –o attack attack.c
./attack
bash check.sh
```

Task 2.B: An Improved Attack Method

In this task, we learn that task 2.A has a slight flaw in that the file created '/tmp/XYZ' has its ownership changed from seed to root. This makes it so that we are no longer able to link and unlink using the attack program. The workaround provided in task 2.A just involved us deleting the file as root so a new file with seed as the owner could be created again. Realistically, this is not an option for us so we will learn how to approach this problem. We learn that that attack program has a race condition of its own; between the unlink() and link() statements. If we could make these two statements atomic, meaning make a call to the system at the same time rather than one then the other, then we could potentially fix this problem. To start, we modify 'attack.c' so that it creates two symbolic links ABC and XYZ and then switches them using the `SYS_renameat2` system call rather than unlink and link one then unlink and link the other. After that, we compile 'attack.c' into a new program called 'attack2b.' I then run both 'attack2b' and 'check.sh' in separate terminals and wait a few minutes. After 5 minutes or so, the attack successfully added the entry into the password file. I confirm this by attempting to log into the 'test' user and I can without a password and have access to root privileges. The commands used for this task are listed below.

```
gcc –o attack2b attack.c
./attack2b
bash check.sh
```

Task 3: Countermeasure: Applying the Principle of Least Privilege

In this task, we learn that the 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 introduces 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; if users do not need certain privilege then privilege needs to be disabled. We can apply this by modifying the vulnerable program. We start by adding in the line of code 'seteuid(realUID);' before checking for access so we can drop privileges. Then at the end of the program, I added the line 'seteuid(effUID);' to set the UID back to what it was before to get the privileges back. We then compile the program and make it a root owned, SetUID program. I then modify 'check.sh' to change where the input is written (vulp-task3 instead of vulp) and rename it 'check-task3.sh'. After that, I run the attack program from task 2.B and check-task3.sh in different terminals. Upon running them, I noticed that in the terminal running check-task3.sh, we see Segmentation Faults. Ultimately, we have disabled the attack because in our attack program we set the effective UID to the real ID, which is that of seed. Since seed does not have permission to open /etc/passwd, this will result in the attack failing. By setting the effective UID to be the real UID at the beginning of the attack program, we no longer allow for unnecessary privileges. The commands used are listed below.

```
gcc –o vulp-task3 vulp.c
./attack2b
bash check-task3.sh
```

Task 4: Countermeasure: Using `Ubuntu's` Built-in Scheme

In the final task, we turn on Ubuntu's built-in protection against race conditions by using the command `sudo sysctl -w fs.protected_symlinks=1`. We then go ahead and run both the attack2b program and the check.sh script. Note that we are not using the same files from task 3, but instead task 2.B. Upon running these processes, I notice more Segmentation Faults. This proves that the attack no longer works and that the vulnerability is gone. The reason for this has to do with the built-in protection scheme. Basically this function works by only allowing symbolic links to be followed by their owners or followers. In our example, since /tmp is root owned and the program we are using is a root owned, SetUID program, then we will not be able to execute the attack because only root can access the symbolic links it owns. Although this protects against race conditions, it does not fully protect. Damage may still be done, but this a good mechanism to have in place to prevent unwanted users accessing files or programs intended for certain users. The commands used in this task are listed below.

```
./attack2b
bash check.sh
```