Race Condition Vulnerability Lab

Austin Hansen

1001530325

2.1 Initial Setup



First we disable the symlinks protection, which would otherwise stop our attack.

2.2 A Vulnerable Program

Text

Description automatically generated

After that, we compile our vulnerable program, making the binary into a Set-uid root program, since the program is no longer protected by the symlink guards we can exploit the time it takes for the program to verify that a file can be accessed by a user, to insert a symlink to a say more restricted file.

2.3 Task 1: Choosing Our Target

Text

Description automatically generated

For this exploit we are targeting /etc/passwd, in order to get the lay of the land so to say, we can use a “magic value” in /etc/passwd that indicates a passwordless account, in order to see if it works, we first as a super user add it to /etc/passwd.

Text

Description automatically generated

We insert “test:U6aMy0wojraho:0:0:test:/root:/bin/bash” into /etc/passwd, now to see if we can swap to the user “test”

Text

Description automatically generated

Success, easiest root that we have gotten yet!

2.4 Task 2: Launching the Race Condition Attack

**Task 2.A: Slow deterministic version of the attack:**

**Text

Description automatically generated**

First we edit vulp.c to sleep for 10, which will give us an opening to launch our attack.

Text

Description automatically generated

Text

Description automatically generated

Next, we create two programs (derived from the slides) to help us perform our attack, **target\_process.sh** and **attack.c**. target process repeatedly calls vulp, and attack attempts to insert our file of interest into fopen.

Text

Description automatically generated



Running target\_process.sh in window 1 and attack.c in window 2 we begin our attack! But what are the results?

A picture containing text

Description automatically generated

Logging in as test, we can see that our attack was successful, we have full root privileges.

**Task 2.B: Full version of attack**

**Text

Description automatically generated**

After resetting everything back to the status quo from the previous task, we increase the sleep time on attack.c, this increases the likelihood that we will infact succeed in our attack. (though I’m not entirely sure why)

Text

Description automatically generated



Repeating what we did above, we begin our attack and anticipate the results, this time however, vulp.c does not sleep…

Text

Description automatically generated

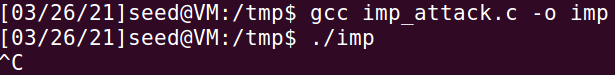
Aha, success after a bit of waiting, we now have root access!

**Task 2.C: An Improved Attack Method**

**Text, letter

Description automatically generated**

It was possible for our earlier attack method to have a race condition between the upper and lower sets of unlink and symlinks, so a new method of attack was devised. This method keeps the variables and unlink and symlink atomic, meaning that it is no longer possible for a race condition to exist within this program. Also, everything was set back to normal, I purged /etc/passwd of the test user, and vulp.c remains the same.

****

Here we quickly compile our new attack program, that I named **imp\_attack**. Let’s see what it can do.

**Text

Description automatically generated**

It seems that it can do quite a lot actually, this was the fastest time to have /etc/passwd change. But do we have root?

Text

Description automatically generated

Yes, we do.

2.5 Task 3: Countermeasure: Applying the Principle of Least Privilege

Text

Description automatically generated

Now we try to stop this exploit. One method is to adhere to the principle of least privilege, where a user can only run what they need to run and elevated privileges are only given out only when necessary. In order to apply that here, I added two variables, **real\_uid** and **eff\_uid**, they are of course our uid and euid, whats more is that **seteuid (real\_uid)** removes root privileges when this program is ran, and critically before our race condition occurs. **seteuid(eff\_uid)** restores any privileges that the user had. But will this stop our attack?

A picture containing text

Description automatically generated

Before we forget, lets make double sure that root owns vulp. Now, will the imp attack from the last task succeed?

Text

Description automatically generated

After letting it sit for a while, it appears that the attack has failed, also it seems to have somewhat frozen up, as I did not see any movement for a while. Perhaps that is a flaw with how I restricted the above program, as this should output some segfaults.

2.6 Task 4: Countermeasure: Using Ubuntu’s Built-in Scheme

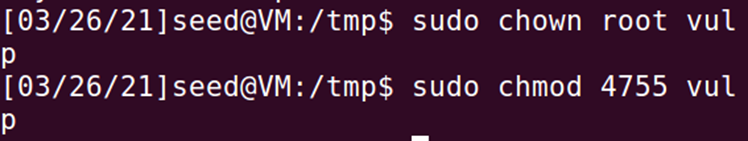
Text

Description automatically generated

Text

Description automatically generated

We have reengaged ubuntu’s defenses and reset vulp.c to its starting configuration. I purposely reset vulp.c back to its vulnerable state to see just how good ubuntu’s defenses are.



Can’t forget about giving root ownership over vulp! Lets launch the attack!

Text

Description automatically generated

Text

Description automatically generated

This attack went nowhere fast, target\_process.sh printed nothing nor accepted any string, and thus imp could not put the test user into /etc/passwd, our attack is a failure. It is clear, that almost all of the successful attacks required that the directories have sticky bits enabled. (1) How this protection works is found at the start of the lab, “symlinks in world-writable sticky directories (e.g., /tmp) cannot be followed if the follower and directory owner do not match the symlink owner”. (2) While this never came up while I was testing, I believe that it is possible for the attack to still work if say, root was the follower and seed owned both the directory and symlink, as root has elevated privileges compared to seed.