Daniel Oliveros

Garrett Bogart

Spring 2018 – Independent Study

Race-Condition Vulnerability Lab

**Background**

When programmers hear ‘race conditions’ they will probably think of two or more threads reading and writing from the same memory location. In the context of this lab the race condition is a time-of-check to time-of-use (TOCTTOU). This race condition occurs when a condition for a resource is checked before the resource is used. What we are trying to exploit is the context switches between the vulnerable program and the attack program. If these two programs ran independently and had no context switches then there would be no vulnerability. Since modern day computers have multiple cores context switches are incredibly common.

We are exploiting the time when the permissions check occurs in our vulnerable program and is then a context switch immediately occurs before the resource is used. Our attack program then needs to be switched in and the attack program then needs to change the symbolic link of the file that our vulnerable program is writing to. Then when the vulnerable file is switched back in it will then write to the new file that our symbolic link points to.

An important part of this lab is to understand the basic layout of the shadow file and the password file. These files contain rows of data. Each row contains seven pieces of information each divided by a colon.

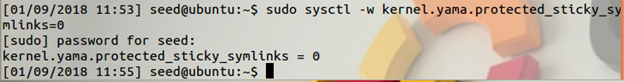
Example of password file: root:x:#:#:root:/root:/bin/bash

1. root : is the username of the user
2. x : is the password field. This indicates to go to the shadow file to find the hash of the password.
3. #: numeric user id. 0 is for root users
4. #: numeric group id: Red Hat uses group id's in a fairly unique manner for enhanced file security. Usually the group id will match the user id.
5. root: Full name of the user
6. /root: User’s home directory
7. /bin/bash: User’s shell account

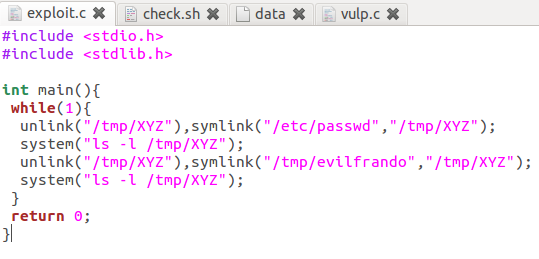
In our case, there are two important parts to note. The first, the password field, 2. Generally the password is stored in the shadow file. In our case we can actually store the password in this field. We don’t actually want to find what our password hashes to, so ideally we would like to not have a password. This is a known hash value, U6aMy0wojraho, and this will give us an empty password. The second, the user id should be 0 to give the user root access.

**Task 1**

First, we disabled the safety measures found natively in the machine. The countermeasure we’re turning off makes it so programs don’t follow symbolic links found in directories that can be modified by anyone.

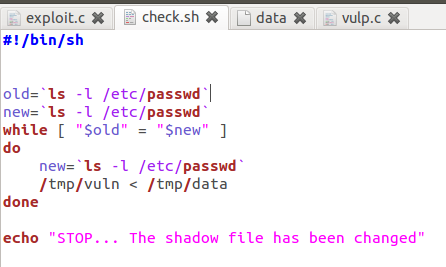


After this, we proceeded to write a program that would exploit the vulnerability. It took a while to get it to work, but once we understood well enough the idea behind it we came up with this program:

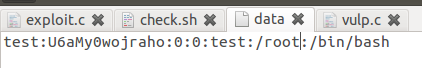


The file “evilfrando” is an empty text file created by the user. Since they have access to it, if the symbolic link switch happens right in between the check for access and the file being open we will end up opening any other file we create a link to.

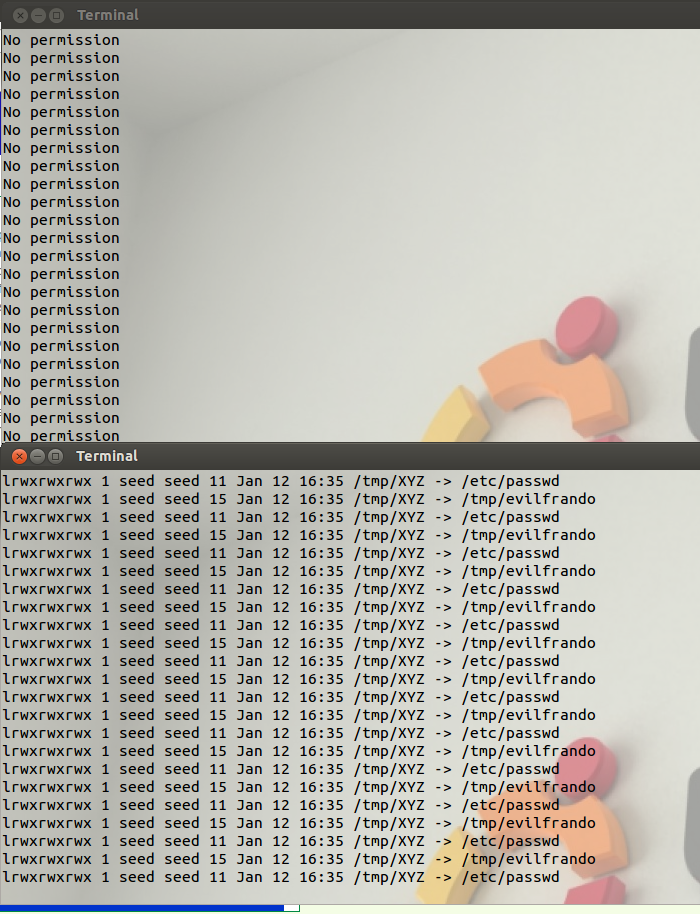
We also modified the check given to us slightly, this was done so we could also run the vulnerable program with the same script:



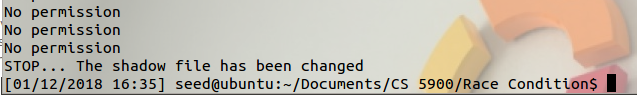
The data file we’re passing into our vulnerable program contains data for a user with root access called “test”, the password is nothing.



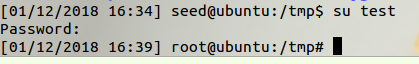
When running these programs alongside one another, this is what our output looked like



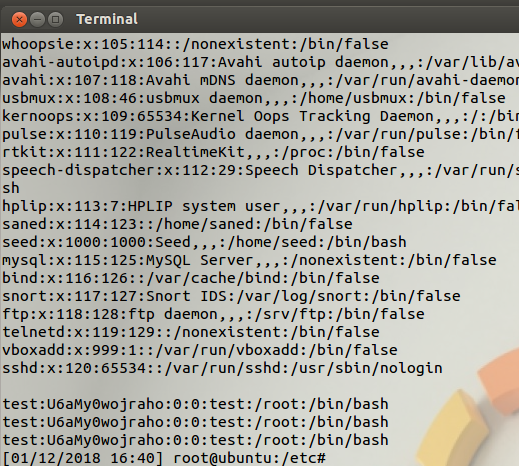
Until it works, which causes the top terminal to stop and display the message:



When this occurs, we are now able to log into the user we created and gain root access

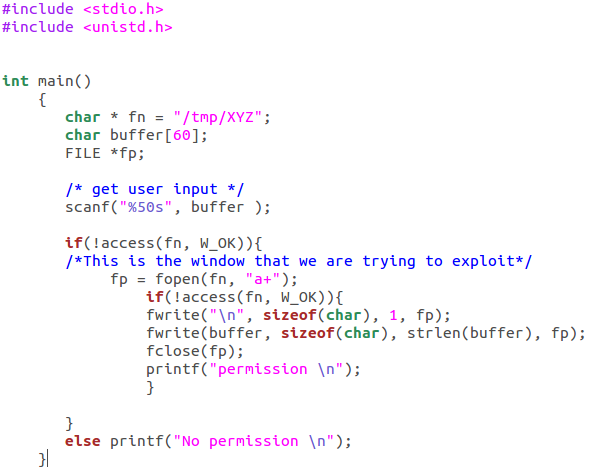


The modified passwd file looks like this



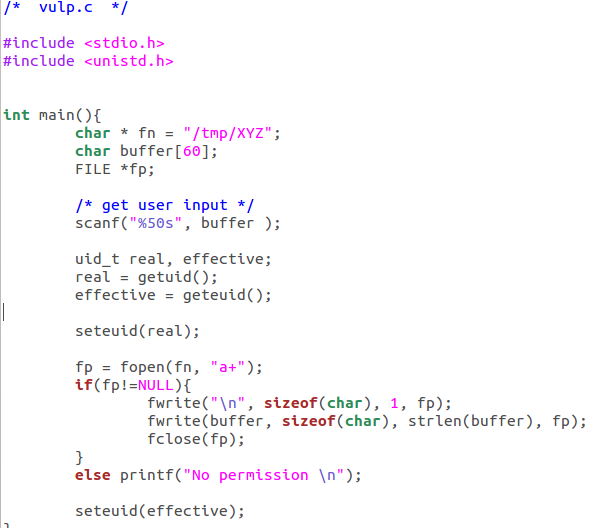
**Task 2**

To increase the odds of an exploit failing A second access check is added before the file is written to. This caused the exploit to not succeed in five minutes; whereas, the exploit succeeded in under 5 seconds before.



**Task 3**

To make it impossible for a user to use their effective user ID to edit files they wouldn’t have access to normally with their real ID we modified the code. The following code only allows a user to open a file if their real ID gives them the privilege to open it.

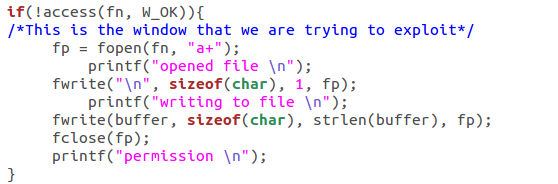


**Task 4**

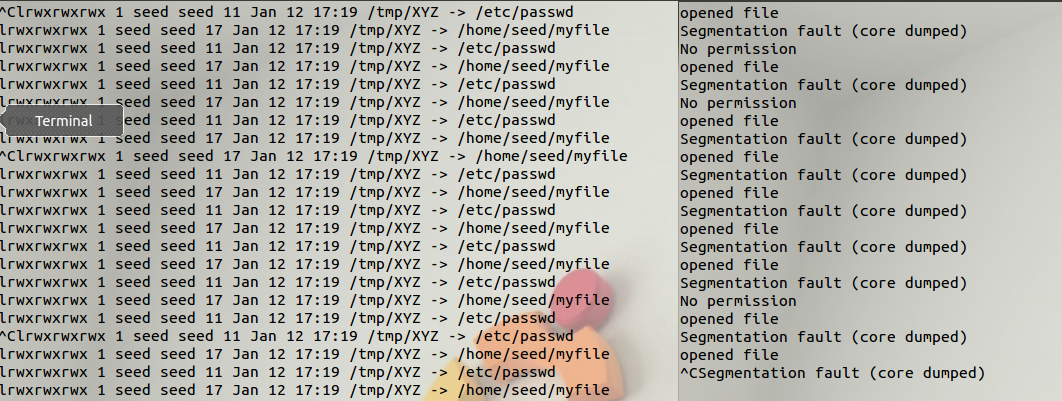
In this task we get to turn on the ubuntu security measures against race conditions. Ubuntu prevents the following of symbolic link unless that do not match the user’s ID. When the security is turn on using



Then I modified the vulnerable file slightly to see what would occur.



The purpose was to see how far the attack could get. Once the programs were running



What occurred was that anytime the exploit got past the first check the file would be opened. Then the file would try to be written to. At which point a segmentation fault would occur.

This protection scheme is valid for a large extent of programs.

A possible issue is if they launched the attack when a user’s ID is increased to accomplish a task, for example ping. Since the user’s ID is increased they could then launch an attack with a higher user ID than normal and could possible bypass the protection.

**Issues:**

1. The document itself does not go in depth enough about how to set up the exploit
   1. Does not go in depth on shadow file or password file layout
   2. From the book we learned that it is much easier to attack the password file rather than trying to attack the shadow file. A person is able to gain root access through attacking the password file only. This is done by having the password in the file be U6aMy0wojraho.
   3. The description does not state it, but in order for the vulnerable program to have the access we want we have to use the commands
2. For the shell command it is important to note the difference between the single quote, ‘, and the tilde, `. The tilde is needed otherwise the input is a string.

 vs 

1. Another good thing to add is the system() function in exploit.c. This is part of stdlib.h. This function allows the user to see the symbolic link changes that they are trying to implement.



1. There was some confusion as to where the user created file had to go. Daniel put his in /tmp/ ; whereas, I put mine in /home/seed. There was no difference between the two locations. The only thing that is needed is the correct path to the file.
2. An important note is that the shell program that was given to us checks the shadow file. We changed it to check the passwd file instead.