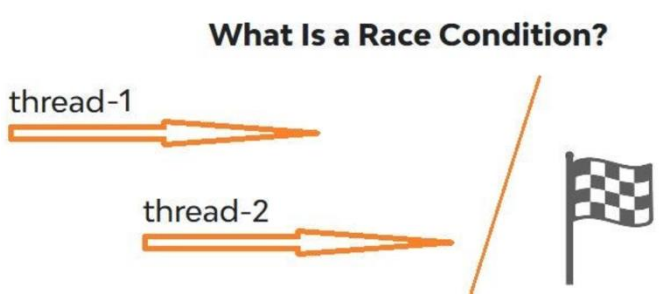
Race Condition Vulnerability

Lecture 13

Instructor: Dr. Cong Pu, Ph.D.

cong.pu@okstate.edu

Acknowledgment: Adapted partially from course materials from Dr. Wenliang Du at Syracuse University, Dr. Fengwei Zhang at Southern University of Science and Technology, and Dr. Steven M. Bellovin at Columbia University.



Introduction

- ***Race condition***
  - a situation where the ***output*** of a system or program is dependent on the ***time*** of other ***uncontrollable events***
  - what if a *privileged program* has a *race condition problem*?
    - attackers might affect the *output* of the *privileged program*
- ***Race condition*** in *software*
  - ***two concurrent*** *threads* or *processes* access a ***shared resource*** in a way that *unintentionally* produces *different results* depending on the *sequence* or *timing* of the *processes* or *threads*

General Race Condition Problem

- Example: the following code runs inside an ATM
  - when customer withdraws money from ATM, *withdraw()* checks remote database and sees whether the amount to be withdrawn is less than customer's current balance
    - if yes, authorize the withdraw and updates the balance

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       echo "You have withdrawn: $amount";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

Scenario:

- assuming you have $1,000
- will you be able to withdraw $1,800?

General Race Condition Problem

- How to withdraw $1,800 if the current balance is $1,000?
  1. need two ATM cards and an accomplice
  2. two of you withdraw $900 ***simultaneously***
     - *after* the *first ATM* finishes *checking balance*, but *before updates balance*, the *second ATM* comes to ask for balance
       - the *second ATM* still see $1,000 and authorize the withdraw

(There will still be $100 left on the balance.)

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        echo "You have withdrawn: $amount";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

Vulnerability: 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) Race Condition Vulnerability
  - occurs when checking for a condition ***before*** using a resource
    - the condition can ***change*** between the time of check and the time of use
- Dirty COW Race Condition Vulnerability
  - allows attackers to modify any protected file, as long as the file is readable to them
    - gain the root privilege
  - affects Android which is built on top of Linux

SET-UID (Set User ID upon execution) is a special permission bit used in Unix-like operating systems.

- When this permission is set on an executable file, the program runs with the privileges of the file's ***owner*** rather than the privileges of the ***user*** executing it.
- This is particularly useful for allowing regular users to execute certain tasks that require elevated privileges.
- Ref.: https://www.liquidweb.com/blog/how-do-i-set-up-setuid-setgid-and-sticky-bits-on-linux/

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 (EUID): root
- Real User ID (RUID): seed

- The above program writes to a file in the */tmp* directory (commonly used to store temporary data; world-writable)
- As the root can write to any file, this program can write to any file
- To prevent user from overwriting other people's file, this 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 permission to */tmp/X* file
    - 0 returned if the real user does not have permission
- After the check, the file is opened for writing

- Root-owned Set-UID program
- Effective UID (EUID): root
- Real User ID (RUID): seed

Race Condition Vulnerability (cont.)

```
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");
}
```

a window between access() and open() function calls

- *open()* checks user's permission: the effective user ID
  - since the root-owned Set-UID program runs with an effective user ID zero, the check of *open()* always succeeds
    - rationale: putting an additional check using *access()* before *open()*
  - however, there is a ***window*** *between the time when the file is checked* and *the time when the file is open*

- Root-owned Set-UID program
- Effective UID (EUID): root
- Real User ID (RUID): seed

Race Condition Vulnerability (cont.)

```
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");
}
```

a window between access() and open() function calls

- What can be done inside the ***window***?
- To help thinking, assuming the program is *running very slowly*
  - so slow that it takes one minute to execute one line of the code
- Goal: use the program's root privilege to write to a protected file, /etc/passwd (password file)
  - is it possible? you might say it is not possible
    - once a privileged program runs, its internal memory cannot be changed
    - cannot modify the program as normal users (no write permission to root-owned file)

- Root-owned Set-UID program
- Effective UID (EUID): root
- Real User ID (RUID): seed

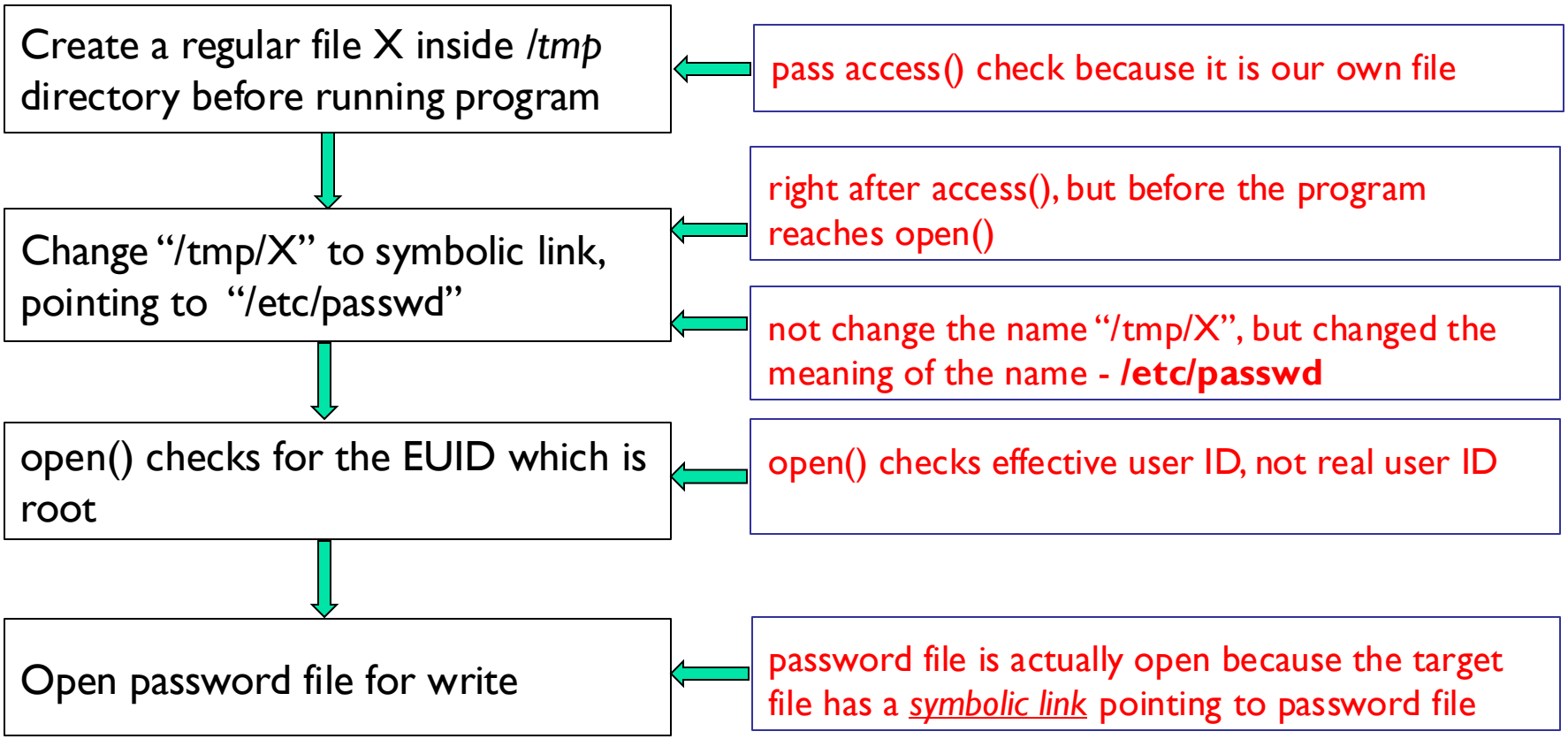
Race Condition Vulnerability (cont.)

```
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");
}
```

a window between access() and open() function calls

- What can be done inside the ***window***?
- Direction: figure out how to make */etc/passwd* become the target file, without changing the file name used in the program
  - *symbolic link* (soft link) helps us to achieve it
    - a special type of file that points to another file or directory
      - *independent*: if the symbolic link is deleted, the original file remains unaffected
      - *broken links*: if the target is moved or deleted, the symbolic link becomes a "broken link" and won't work

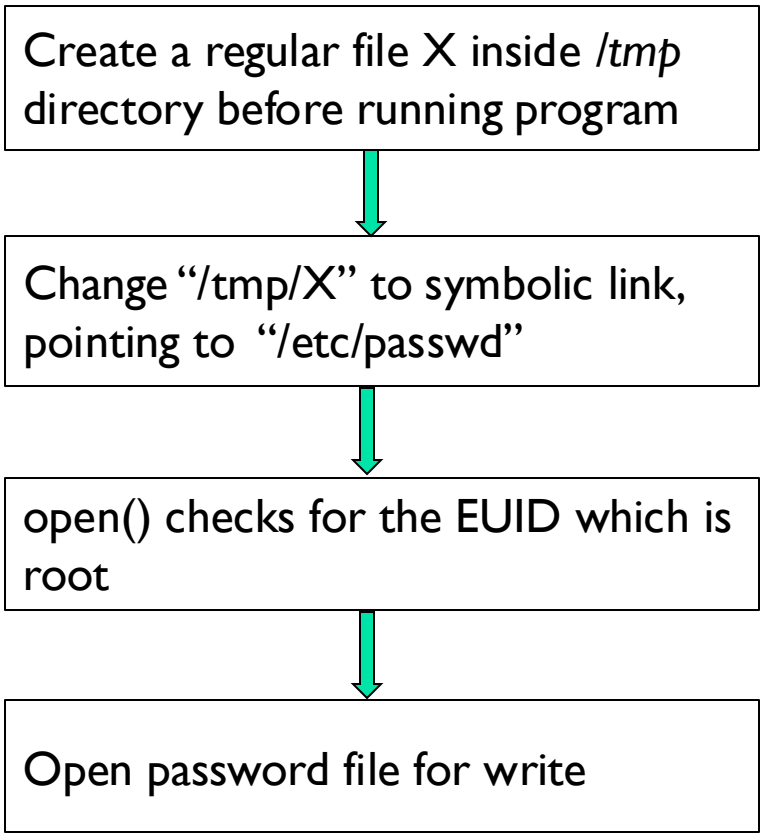
Race Condition Vulnerability (cont.)



```
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");
}
```

a window between access() and open() function calls

bility (cont.)



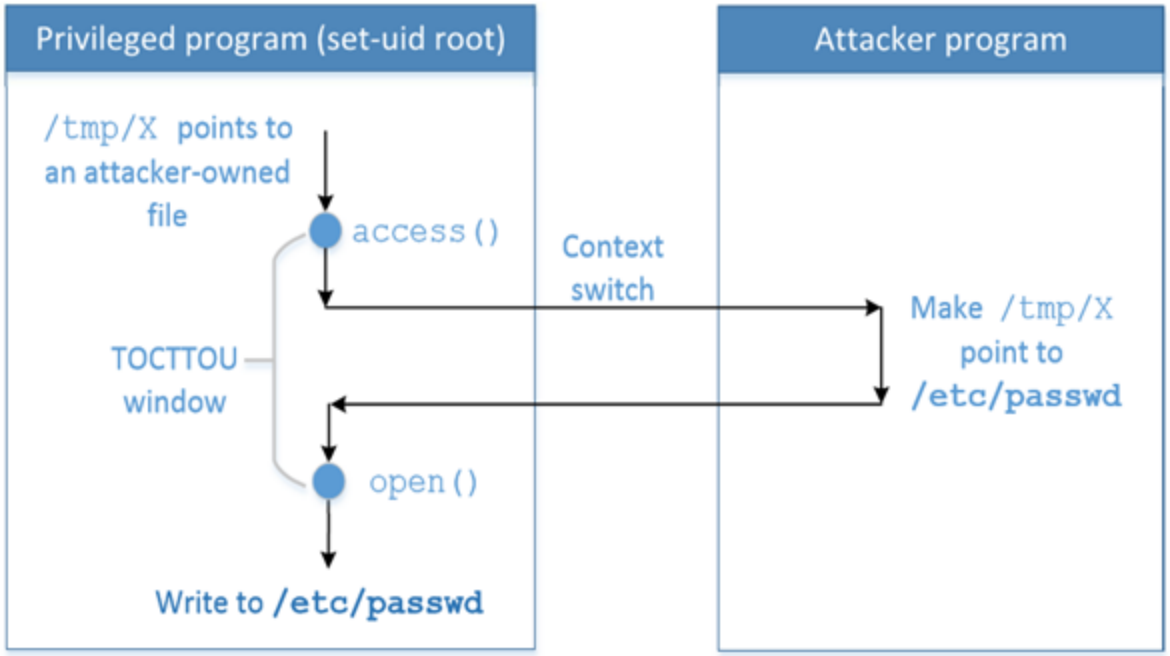
Issues:

- As the program runs billions of instructions per second, the window between the time to check, access(), and time to use, open(), 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
  - *must make the change during the window*

Solution:

- **try randomly**
  - the chance of hitting the window is low
  - try enough times, eventually be lucky

Race Condition Vulnerability (cont.)



- To win the race condition (TOCTTOU window), we need two processes:
  - one runs vulnerable program in a loop
  - the other runs the attack program

Understanding the Winning

Consider steps for two programs:

Attack Program:

- **A1** : Make "/tmp/X" point to a file owned by us
- **A2** : Make "/tmp/X" point to /etc/passwd

Vulnerable Program:

- **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)

- the way these two sequences are interleaved is difficult to control
  - depending on many factors such as CPU speed, context switch, etc.

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

Another Race Condition Example

Original intention: create a new file

```
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

Outcome: write to a protected file