Race Conditions

- A common cause of security bugs
- Usually involve multiprogramming or multithreaded programs
- Caused by different threads of control operating in unpredictable fashion
  - When programmer thought they'd work in a particular order

What Is a Race Condition?

- A situation in which two (or more) threads of control are cooperating or sharing something
- If their events happen in one order, one thing happens
- If their events happen in another order, something else happens
- Often the results are unforeseen

Security Implications of Race Conditions

- Usually you checked privileges at one point
- You thought the next lines of code would run next
  - So privileges still apply
- But multiprogramming allows things to happen in between

The TOCTOU Issue

- Time of Check to Time of Use
- Have security conditions changed between when you checked?
- And when you used it?
- Multiprogramming issues can make that happen
- Sometimes under attacker control

A Short Detour

- In Unix, processes can have two associated user IDs
  - Effective ID
  - Real ID
- Real ID is the ID of the user who actually ran it
- Effective ID is current ID for access control purposes
- Setuid programs run this way
- System calls allow you to manipulate it

Effective UID and Access Permissions

- Unix checks accesses against effective UID, not real UID
- So setuid program uses permissions for the program's owner
  - Unless relinquished
- Remember, root has universal access privileges

An Example

- Code from Unix involving a temporary file
- Runs setuid root

```
res = access("/tmp/userfile", R_OK);
If (res != 0)
  die("access");
fd = open("/tmp/userfile",O_RDONLY);
```

What's (Supposed to Be) Going on Here?

- Checked access on `/tmp/userfile` to make sure user was allowed to read it
  - User can use links to control what this file is
- `access()` checks real user ID, not effective one
  - So checks access permissions not as root, but as actual user
- So if user can read it, open file for read
  - Which root is definitely allowed to do
- Otherwise exit

What's Really Going On Here?

- This program might not run uninterrupted
- OS might schedule something else in the middle
- In particular, between those two lines of code

How the Attack Works

- Attacker puts innocuous file in

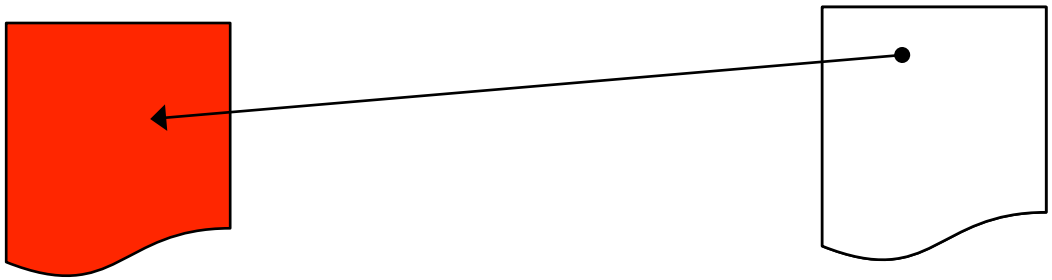
  `/tmp/userfile`
- Calls the program
- Quickly deletes file and replaces it with link to sensitive file
  - One only readable by root
- If timing works, he gets secret contents

The Dynamics of the Attack

Success!

Don't move that file again!

/etc/secretfile /tmp/userfile



1. Run program
2. Change file

```
res = access("/tmp/userfile", R_OK);
if (res != 0)
        die("access");
fd = open("/tmp/userfile",O_RDONLY);
```

How Likely Was That?

- Not very
  - The timing had to be just right
- But the attacker can try it many times
  - And may be able to influence system to make it more likely
- And he only needs to get it right once
- Timing attacks of this kind can work
- The longer between check and use, the more dangerous

Some Types of Race Conditions

- File races
  - Which file you access gets changed
- Permissions races
  - File permissions are changed
- Ownership races
  - Who owns a file changes
- Directory races
  - Directory hierarchy structure changes

A Real Example

- In the Linux SystemTap utility (2011)
  - Which gathers info about the system
- Allows modules to be loaded
- Checks privileges, but then there's a delay before loading the module
  - So it might load a different module
  - Allowing privilege escalation
- A genuine TOCTOU issue

Preventing Race Conditions

- Minimize time between security checks and when action is taken
- Be especially careful with files that users can change
- Use locking and features that prevent interruption, when possible
- Avoid designs that require actions where races can occur

Randomness and Determinism

- Many pieces of code require some randomness in behavior
- Where do they get it?
- As earlier key generation discussion showed, it's not that easy to get

Pseudorandom Number Generators

- PRNG
- Mathematical methods designed to produce strings of random-like numbers
- Actually deterministic
  - But share many properties with true random streams of numbers

Attacks on PRNGs

- Cryptographic attacks
  - Observe stream of numbers and try to deduce the function
- State attacks
  - Attackers gain knowledge of or influence the internal state of the PRNG

An Example

- ASF Software's Texas Hold'Em Poker
- Flaw in PRNG allowed cheater to determine everyone's cards
  - Flaw in card shuffling algorithm
  - Seeded with a clock value that can be easily obtained

Another Example

- Netscape's early SSL implementation
- Another guessable seed problem
  - Based on knowing time of day, process ID, and parent process ID
  - Process IDs readily available by other processes on same box
- Broke keys in 30 seconds

A Recent Case

- The chip-and-pin system is used to secure smart ATM cards
- Uses cryptographic techniques that require pseudo-random numbers
- Cambridge found weaknesses in the PRNG
- Allows attackers to withdraw cash without your card
- Seems to be in real use in the wild

How to Do Better?

- Use hardware randomness, where available
- Use high quality PRNGs
  - Preferably based on entropy collection methods
- Don't use seed values obtainable outside the program