COSC 458-647
Application Software Security

Race condition

Concurrency

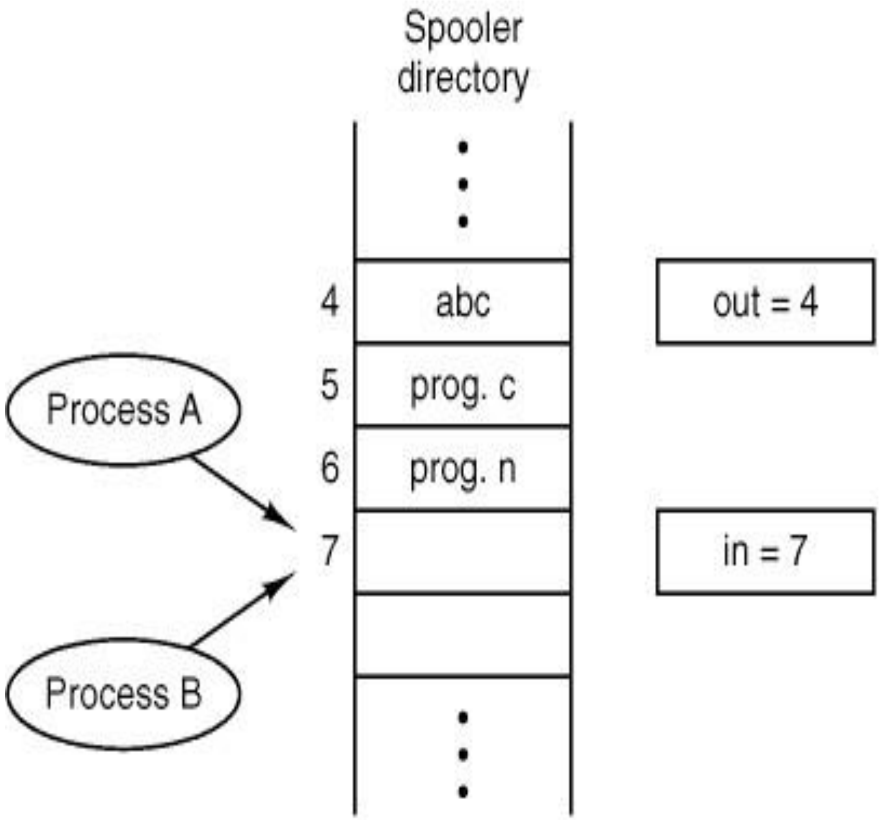
- Concurrency occurs when two or more separate execution flows are able to run simultaneously.
- Examples of independent execution flows include *threads*, *processes*, and *tasks*.
- Concurrent execution of multiple flows of execution is an essential part of a modern computing environment.

Thread example

```
#include <iostream>
#include <thread>
using namespace std;

void func(int x) {
   cout << "Inside thread " << x << endl;
}

int main() {
   thread th(&func, 100);
   th.join();
   cout << "Outside thread" << endl;
   return 0;
}
```

```
$ g++ -std=c++11 -o threads_1 threads_1.cpp -pthread
```

Race condition



- An unanticipated execution ordering of concurrent flows that results in undesired behavior is called a race condition
  - A software defect and frequent source of vulnerabilities.

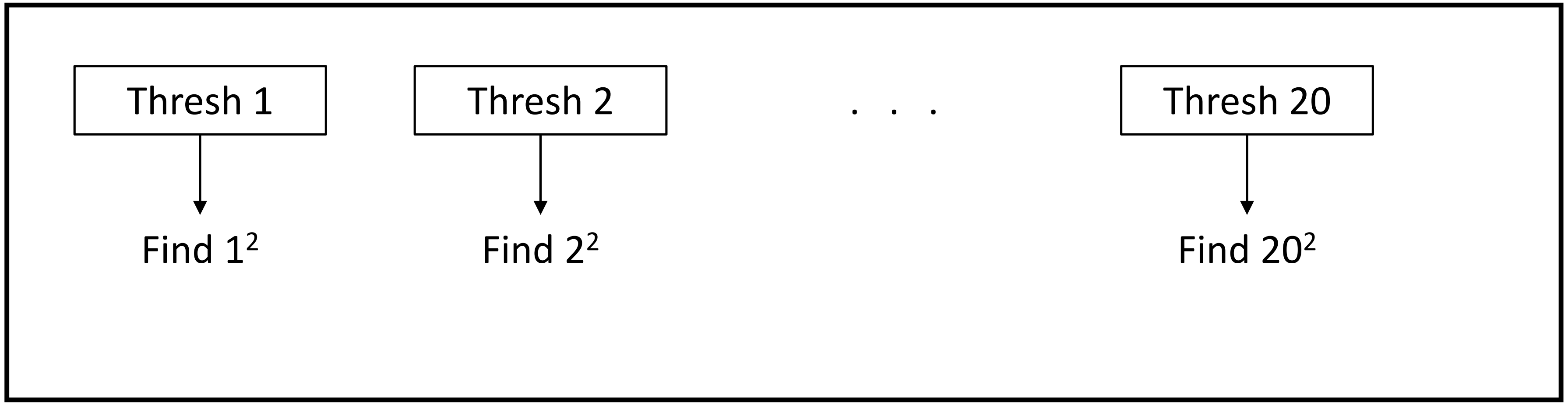
- E.g., two processes want to access shared memory at the same time.

- Race conditions result from runtime environments, including operating systems, that must control access to shared resources, especially through process scheduling.

Race condition: Example

- Let us imagine that x^2 is a very costly operation and we want to calculate the sum of squares up to a certain number.

- It would make sense to parallelize the calculation of each square across threads.



Example: Race condition

```
#include <iostream>
#include <vector>
#include <thread>
using namespace std;

int accum = 0;
void square(int x) {
    accum += x * x;
}
int main() {
    vector<thread> ths;
    for (int i = 1; i <= 20; i++) {
        ths.push_back(thread(&square, i));
    }
    for (auto& th : ths) {
        th.join();
    }
    cout << "accum = " << accum << endl;
    return 0;
}
```

```
$ g++ -std=c++11 -o threads_2 threads_2.cpp -pthread
```

This should sum all squares up to and including 20.

We iterate up to 20, and launch a new thread in each iteration that we give the assignment to.

After this, we call `join()` on all our threads, which is a blocking operation that waits for the thread to finish, before continuing the execution.

This is important to do before we print accum, since otherwise our threads might not be done yet. You should always join your threads before leaving main, if you haven't already.

Your tasks (1/3)

- Run the program once and observe the outcome
  - Is it `2870`?

- Run the program multiple times and observe the outcomes

```
$ for i in {1..40}; do ./threads_2; done
```

- Any inconsistencies yet? Better yet, let's list all distinct outputs from 1000 separate runs, including the count for each output:

```
$ for i in {1..1000}; do ./threads_2; done | sort | uniq -c
```

- Why are the inconsistencies?

Your tasks (2/3)

```
$ for i in {1..1000}; do ./threads_2; done | sort | uniq -c
```

- Plenty of incorrect answers, even though most of the time it gets it right.
- This is because of **race condition**.
  - When the compiler processes `accum += x * x;` reading the current value of accum and setting the updated value is not an atomic (meaning indivisible) event

Your tasks (3/3)

- Let's re-write `square()` to:

```
int temp = accum;
temp += x * x;
accum = temp;
```

?Does that solve the problem?

```
// Thread 1             // Thread 2
int temp1 = accum;      int temp2 = accum;      // temp1 = temp2 = 0
                        temp2 += 2 * 2;         // temp2 = 4
temp1 += 1*1;                                   // temp1 = 1
                        accum = temp1;          // accum = 1
accum = temp2;                                  // accum = 4
```

Race condition

- Race condition needs:
  - Concurrency
    - There must be at least two control flows executing concurrently.
  - Shared Object
    - A shared *race object* must be accessed by both of the concurrent flows.
  - Change State
    - At least one of the control flows must alter the state of the race object.

Critical sections

Critical Sections should be well-designed to avoid race conditions:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

```
#include <iostream>
#include <vector>
#include <thread>
using namespace std;

int accum = 0;

void square(int x) {
    accum += x * x;
}

int main() {
    vector<thread> ths;
    for (int i = 1; i <= 20; i++) {
        ths.push_back(thread(&square, i));
    }
    for (auto& th : ths) {
        th.join();
    }
    cout << "accum = " << accum << endl;
    return 0;
}
```

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Extra credit

- How can we fix the race condition in the previous example without introducing any mutex/lock variable?
  - Hint: Is keeping `accum` as a global variable a good style? What can you do better?

Extra credit

- Since keeping `accum` as a global variable is poor style, we would rather pass it into the thread.
- Add a parameter `int& accum` to `square()` header.
- It is important that it's a reference, since we want to be able to change the accumulator.
  - However, we can't simply call `thread(&square, accum, i)`, since it will make a copy of `accum` and then call `square()` with that copy.
  - To fix this, we wrap `accum` in `ref()`, making it `thread(&square, ref(accum), i)`. Try it.

Eliminating race conditions

- Identify race windows.
  - A code segment accesses the race object in a way that opens a window of opportunity during which other concurrent flows could “race in” and alter the race object.

- Eliminate race conditions by making conflicting race windows mutually exclusive.

Mutual Exclusion

- Only one competing thread is allowed to be in a critical section.

- C and C++ support several synchronization primitives:
  - Mutex variables,
  - Semaphores,
  - Pipes, named pipes,
  - Condition variables,
  - CRITICAL_SECTION objects,
  - Lock variables.

Mutex: Mutual Exclusion

A `mutex` object (mutual exlusion) allows us to encapsulate blocks of code that should only be executed in one thread at a time.

```
int accum = 0;

mutex accum_mutex;

void square(int x) {
    int temp = x * x;
    accum_mutex.lock();
    accum += temp;
    accum_mutex.unlock();
}
```

1. The first thread that calls `lock()` gets the lock.
2. During this time, all other threads that call `lock()`, will simply halt, waiting at that line for the mutex to be unlocked.
3. It is important to introduce the variable temp, since we want the x^2 calculations to be outside the lock-unlock block. Why?

Atomic

- C++11 offers even nicer abstractions to solve this problem. For instance, the atomic container:

```
#include <atomic>

atomic<int> accum(0);

void square(int x) {

    accum += x * x;

}
```

Introduce an atomic variable accum initialized to 0

We don't need to introduce temp here, since x*x will be evaluated before handed off to accum, so it will be outside the atomic event.

Tasks

- An even higher level of abstraction avoids the concept of threads altogether and talks in terms of *tasks* instead.

```
#include <iostream>
#include <future>
#include <chrono>
using namespace std;
int square(int x) {
   return x * x;
}
int main() {
   future<int> taskA = async(launch::async, &square, 10);
   int v = taskA.get();
   cout << "The thread returned " << v << endl;
   return 0;
}
```

Tasks

```
future<int> taskA = async(&square, 10);
int v = taskA.get();
```

- The `async` construct uses an object pair called a `promise` and a `future`. The former has made a promise to eventually provide a value.
- The `future` is linked to the `promise` and can at any time try to retrieve the value by `get()`.
- If the `promise` hasn't been fulfilled yet, it will simply wait until the value is ready.
- The `async` hides most of this for us, except that it returns in this case a `future<int>` object.

Use Tasks To Avoid Race condition

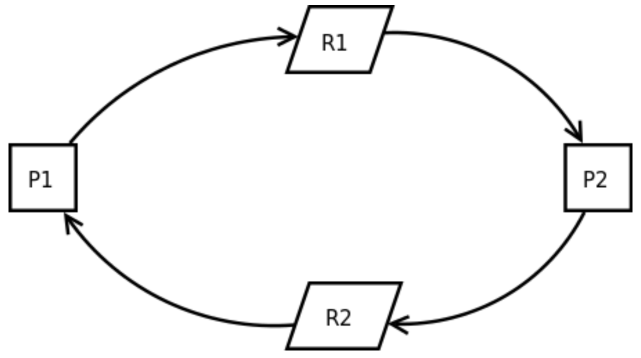
```
#include <iostream>
#include <vector>
#include <thread>
using namespace std;

int accum = 0;
void square(int x) {
    accum += x * x;
}
int main() {
    vector<thread> ths;
    for (int i = 1; i <= 20; i++) {
        ths.push_back(thread(&square, i));
    }
    for (auto& th : ths) {
        th.join();
    }
    cout << "accum = " << accum << endl;
    return 0;
}
```

Use async to solve the sum of squares problem.

1. Iterate up to 20 and add your future<int> objects to a vector<future<int>>.
2. Then, finally iterate all your futures and retrieve the value and add it to your accumulator.

Other concepts

- Deadlock:
  - Two or more competing processes are each waiting for the other to finish, and thus neither ever does.
  - System comes to a halt.



- Lifelock:
  - An entity never acquires a resource needed to finish.
  - But system continues to work.

Time of Check Time of Use

- Race Condition can result from trusted or untrusted sources.
  - Trusted sources are within the program.
  - Untrusted sources are separate applications or processes.

- TOCTOU race conditions occur during file I/O
  - Race window by checking for some race object and later accessing it.

Time of Check Time of Use

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

int main(int argc, char *argv[]) {
  FILE *fd;
  if (access("/some_file", W_OK) == 0) {
      printf("access granted.\n");
      fd = fopen("/some_file", "wb+");
      /* write to the file */
      fclose(fd);
  }
      . . .
  return 0;
}
```

The access() function is called to check if the file exists and has write permission.

Time of Check Time of Use

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

int main(int argc, char *argv[]) {
  FILE *fd;
  if (access("/some_file", W_OK) == 0) {
      printf("access granted.\n");
      fd = fopen("/some_file", "wb+");
      /* write to the file */
      fclose(fd);
  }
      . . .
  return 0;
}
```

the file is opened for writing

Time of Check Time of Use

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

int main(int argc, char *argv[]) {
  FILE *fd;
  if (access("/some_file", W_OK) == 0) {
      printf("access granted.\n");
      fd = fopen("/some_file", "wb+");
      /* write to the file */
      fclose(fd);
  }
      . . .
  return 0;
}
```

Race window
Between checking for access and opening file.

Time of Check Time of Use

- Vulnerability
  - An external process can change or replace the ownership of some_file.
  - If this program is running with an effective user ID (UID) of root, the replacement file is opened and written to.
  - *If an attacker can replace some_file with a link during the race window, this code can be exploited to write to any file of the attacker’s choosing.*

Time of Check Time of Use

- The program could be exploited by a user executing the following shell commands during the race window:

```
rm /some_file
ln /myfile /some_file
```

- The TOCTOU condition can be mitigated by replacing the call to `access()` with logic that drops privileges to the real UID, opens the file with `fopen()`, and checks to ensure that the file was opened successfully.

Time of Check Time of Use

Race Condition from GNU File Utilities (v4.1)

```
   ...
1. chdir("/tmp/a");
2. chdir("b");
3. chdir("c");
   // race window
4. chdir("..");
5. rmdir("c");
6. unlink("*");
```

This code relies on the existence of a directory with path /tmp/a/b/c.

Time of Check
Time of Use

Race Condition from GNU File Utilities (v4.1)

```
   ...
1. chdir("/tmp/a");
2. chdir("b");
3. chdir("c");
   // race window
4. chdir("..");
5. rmdir("c");
6. unlink("*");
```

race window

Time of Check
Time of Use

Race Condition from GNU File Utilities (v4.1)

```
   ...
1. chdir("/tmp/a");
2. chdir("b");
3. chdir("c");
   // race window
4. chdir("..");
5. rmdir("c");
6. unlink("*");
```

An exploit is:
mv /tmp/a/b/c /tmp/c

Programmer assumed to be now in /tmp/a/b
After exploit, programmer is in /tmp
If /tmp contains a directory c, this one is now removed

Time of Check
Time of Use

Race Condition from GNU File Utilities (v4.1)

```
   ...
1. chdir("/tmp/a");
2. chdir("b");
3. chdir("c");
   // race window
4. chdir("..");
5. rmdir("c");
6. unlink("*");
```

Programmer assumption about existence of the /tmp/a/b/c directory tree causes an implicit TOCTOU condition.

Files as Locks - File Locking

- Synchronization primitives cannot resolve race conditions from **independent** processes.

- Files can be used as locks.
  - The sharing processes must agree on a filename and a directory that can be shared.
  - A lock file is used as a proxy for the lock. If the file exists, the lock is captured; if the file doesn't exist, the lock is released.
  - One disadvantage of this implementation for a lock mechanism is that the `open()` function does not block.

Unix File Locking

```
int lock(char *fn) {
 int fd;
 int sleep_time = 100;
 while (((fd=open(fn,O_WRONLY|O_EXCL|O_CREAT,0))==-1) && errno==EEXIST) {
  usleep(sleep_time);
  sleep_time *= 2;
  if (sleep_time > MAX_SLEEP)
     sleep_time = MAX_SLEEP;
  }
  return fd;
}

void unlock(char *fn) {
    if (unlink(fn) == -1) {
       err(1, "file unlock");
    }
}
```

Processes agree on file name.
Typically something in /temp

Open does not block.
Therefore: spinlock with increasing sleep-times.

Files as Locks - File Locking

- A file lock can persist indefinitely if the locking process has crashed.
  - Improved version
    - Put the PID of holding process into the lock.
    - When lock is requested, check whether PID belongs to a running process.
    - Problems:
      - PID of crashed process can be reused.
      - Fix needs to be carefully implemented to prevent race conditions.
      - Shared resource might also have been corrupted by a crash.

Files as Locks - File Locking

- Windows Synchronization
  - Named Mutex
    - Have a namespace similar to the file system.
    - CreateMutex() creates mutex object and returns mutex handle.
    - Acquisition and release by
      - WaitForSingleObject()
      - ReleaseMutex()
    - Mutex use is voluntary between programs.

Files as Locks - File Locking

- Windows Synchronization
    - File locks
        - Locks for files or regions of files.
            - Shared Locks
                - Prohibit alteration of files.
            - Exclusive Lock
                - Allows read and write access for holder of lock.
                - Exclude everyone else.
    - File lock use is mandatory.

Files as Locks - File Locking

- Linux implements:
  - Advisory locks
    - Not enforced, hence not secure
  - Mandatory locks
    - Works only on local file system
      - Not on AFS, NFS, …
    - System must be mounted with support for mandatory locking
    - Locking relies on a group ID bit that can be turned off by another process.

File System Exploits - Symbolic Link

- Exploits based on race windows in file system.
  - Symbolic link exploit
    - Unix symbolic linking mechanism (symlink)
      - Referenced file turns out to include a symbolic link.
    - A TOCTOU vulnerability could be:
      - a call to access() followed by fopen(),
      - a call to stat() followed by a call to open(),
      - a file that is opened, written to, closed, and reopened by a single thread.
      - Within the race window, the attacker alters the meaning of the filename by creating a symbolic link.

File System Exploits - Symbolic Link

```
if (stat("/some_dir/some_file", &statbuf) == -1) {
      err(1, "stat");
}
if (statbuf.st_size >= MAX_FILE_SIZE) {
      err(2, "file size");
}

if ((fd=open("/some_dir/some_file", O_RDONLY)) == -1)
{
   err(3, "open - /some_dir/some_file");
}
// process file
```

stats /some_dir/some_file and opens the file for reading if it is not too large.

File System Exploits - Symbolic Link

```
if (stat("/some_dir/some_file", &statbuf) == -1) {
      err(1, "stat");
}
if (statbuf.st_size >= MAX_FILE_SIZE) {
      err(2, "file size");
}

if ((fd=open("/some_dir/some_file", O_RDONLY)) == -1)
{
   err(3, "open - /some_dir/some_file");
}
// process file
```

The TOCTOU *check* occurs with the call of stat()

TOCTOU *use* is the call to fopen()

File System Exploits - Symbolic Link

- Attacker executes the following during the race window :
  - **`rm /some_dir/some_file`**
  - **`ln -s attacker_file /some_dir/some_file`**

- The file passed as an argument to `stat()` is not the same file that is opened.

- The attacker has hijacked `/some_dir/some_file` by linking this name to `attacker_file`.

File System Exploits - Symbolic Link

- Symbolic links are used because
  - Owner of link does not need any permissions for the target file.
  - The attacker only needs write permissions for the directory in which the link is created.
  - Symbolic links can reference a directory. The attacker might replace /some_dir with a symbolic link to a completely different directory

File System Exploits - Symbolic Link

- Example: passwd() functions of SunOS and HP/UX
  - passwd() requires user to specify password file as parameter
    1. Open password file, authenticate user, close file.
    2. Create and open temporary file ptmp in same directory.
    3. Reopen password file and copy updated version into ptmp.
    4. Close both files and rename ptmp as the new password file.

File System Exploits - Symbolic Link

1. Attacker creates bogus password file called .rhosts
2. Attacker places .rhosts into attack_dir
3. Real password file is in victim_dir
4. Attacker creates symbolic link to attack_dir, called symdir.
5. Attacker calls passwd passing password file as /symdir/.rhosts.
6. Attacker changes /symdir so that password in steps 1 and 3 refers to attack_dir and in steps 2 and 4 to victim_dir.
7. Result: password file in victim_dir is replaced by password file in attack_dir.

File System Exploits - Symbolic Link

- Symlink attack can cause exploited software to open, remove, read, or write a hijacked file or directory.
- Other example: StarOffice
  - Exploit substitutes a symbolic link for a file whose permission StarOffice is about to elevate.
  - Result: File referred to gets permissions updated.

Temporary File Open Exploits

- Temporary files are vulnerable when created in a directory to which an attacker has access.
  - Simplest vulnerability not based on race conditions:

```
int fd = open("/tmp/some_file",O_WRONLY|O_CREAT|O_TRUNC,0600);
```

  - If a `/tmp/some_file` file already exists then that file is opened and truncated.
  - If `/tmp/some_file` is a symbolic link, then the target file referenced by the link is truncated.

Temporary File Open Exploits

- An attacker needs to create a symbolic link called /tmp/some_file before this instruction executes.
- This vulnerability can be prevented including the flags O_CREAT and O_EXCL when calling open().

Temporary File Open Exploits

- Mitigation:

```
int fd = open("/tmp/some_file",O_WRONLY|O_CREAT|O_EXCL|O_TRUNC,0600);
```

  - This call to open fails whenever `/tmp/some_file` already exists, including when it is a symbolic link.
  - The test for file existence and the file creation are guaranteed to be atomic.
  - Less susceptible to race conditions.

Temporary File Open Exploits

- Similar secure file opens are possible using `fopen_s()` and `freopen_s()` from ISO/IEC WDTR 2473.
  - These two functions are specified to open files in a safe mode, giving exclusive access.
  - Stream functions have no atomic equivalent, following the deprecation of the ios::nocreate flag.

Temporary File Open Exploits

```
1.  #include <iostream>
2.  #include <fstream>
3.  #include <string>
4.  using namespace std;

5.  int main() {
6.     ofstream outStrm;
7.     ifstream chkStrm;
8.     chkStrm.open("/tmp/some_file", ifstream::in);
9.     if (!chkStrm.fail())
10.     outStrm.open("/tmp/some_file",ofstream::out);
11.      ...
```

The code checks first whether the temp file exists.

Because the test for file existence in lines 8 and 9 and the file open in line 10 both use file names, this code contains a TOCTOU vulnerability

Temporary File Open Exploits

```
1.  #include <iostream>
2.  #include <fstream>
3.  #include <string>
4.  using namespace std;

5.  int main() {
6.      ofstream outStrm;
7.      ifstream chkStrm;
8.      chkStrm.open("/tmp/some_file", ifstream::in);
9.      if (!chkStrm.fail())
10.      outStrm.open("/tmp/some_file",ofstream::out);
11.       ...
```

This code is proposed as a mitigation for the existing open temp file problem.

The code can be exploited by the creation of a symbolic link, named / tmp/some_file, during the race window between the execution of lines 8 and 10.

Temporary File Open Exploits: Mitigation

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
 int fd;
 FILE *fp;
 if ((fd = open("/tmp/some_file",O_EXCL|O_CREAT|O_TRUNC|O_RDWR, 0600)) == -1){
     err(1, "/tmp/some_file");
 }
 fp = fdopen(fd, "w");
```

use of the O_EXCL argument with the open() function

the stream is opened, not with a filename, but with a file descriptor **fd**

Closing the Race Window

- Race condition vulnerabilities are based on the existence of a race window

- Mutual Exclusion Mitigation
  - Unix and especially Windows support many synchronization primitives
  - Object Oriented alternative:
    - Use wrapper functions (decorators):
    - All access to shared resources goes through wrapper functions that check for mutual exclusion.

Closing the Race Window

- Watch out for signals
  - Signals can interrupt normal execution flow at any time
  - Unhandled signals usually default to program termination
  - Signal handler can be invoked at any time, **even in the midst of a mutually excluded section of code**
  - An attacker can lengthen the race window by *sending a signal that is processed during the race window*.

Closing the Race Window

- Watch out for signals: Air Line Reservation System
  - Program is not multi-threaded
  - Uses a signal handler from a special keyboard interrupt to handle flight cancellation
    - Signal handler first reads passenger manifest from file
    - Reassigns passengers to other flights
    - Unlinks the manifest file to prevent further bookings
  - What happens if signal is received while handling a booking?
    - Booking passenger might not be reassigned to another flight, but written to manifest before manifest file is unlinked

Closing the Race Window

- Thread-safe functions
  - Invoked functions can be source of race conditions.
  - They should be ***thread-safe***
    - Can be called by concurrent programs without generating race conditions.
    - Not all API functions are thread-safe!
- Use of Atomic Operations
  - E.g. Linux with open() called with O_CREAT | O_EXCL flag
- Checking File Properties Securely
  - TOCTOU results from the need to check file properties.
    - Linux error: stat() followed by open()
      - Instead: open() followed by fstat()
    - Windows mitigation
      - Use GetFileInformationByHandle() instead of FindFirstFile() or FindFirstFileEx()
    - Linux lstat() allows to guard against symlink vulnerabilities

Eliminating the Race Object

- Race conditions can only exists if there are shared objects.
- Can be difficult to identify race objects.
  - Especially system-shared resources
    - Directory operation might be impacted by another process changing the directory tree further up.
- Software developers can minimize use of system-supplied resources

Eliminating the Race Object

- Use file descriptors, not file names
  - Race object is often not the file, but the directory.
- Avoid Shared Directories
  - Have greatest potential for sharing and deception.

Eliminating the Race Object: Temporary Files

- Unix /tmp is the source of many vulnerabilities

- Mitigation list:
  - Never reuse file names, especially temporary file names. File name reuse creates race conditions.

  - Use random file names for temporary files.

  - When implementing random file names use good techniques

  - Temporary files should be unlinked at the earliest possible opportunity.
    - (1) unlinking removes the file from file system view, even while the file continues to be open to the using process;
    - (2) it minimizes the size of the race window always associated with a call to unlink();
    - (3) if the process crashes, the file will not remain for a tmp cleaner.

Race detection tools

- Static Analysis
  - NP-complete problem, but checkers exist
    - Warlock
    - Extended Static Checking
- Dynamic Analysis
  - RaceGuard: Unix kernel extension for secure temporary files
  - Tsyrklevich & Yee: Examine all file accesses heuristically for race conditions
  - Alcatraz: File modification cache that isolates actual file system from unsafe accesses.

  - Create real runtime environment and analyze actual execution flows.
  - Dynamic tools fail to consider execution paths not taken.
  - Significant runtime overhead
    - Erasure, MultiRace intercepts ops on runtime locks
    - Thread Checker