Secure Coding in C and C++
Race Conditions

Robert C. Seacord & David Svoboda

Agenda

Concurrency

Time of Check, Time of Use

Files as Locks and File Locking

File System Exploits

Mitigation Strategies

Summary

Concurrency

Concurrency occurs when two or more separate execution flows are able to run simultaneously [Dijkstra 65].

Examples of independent execution flows include

- threads
- processes
- tasks

Concurrent execution of multiple flows of execution is an essential part of a modern computing environment.

Race Conditions

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

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

Race Condition Properties

There are three properties that are necessary for a race condition to exist:

1. Concurrency Property. There must be at least two control flows executing concurrently.

2. Shared Object Property. A shared race object must be accessed by both of the concurrent flows.

3. Change State Property. At least one of the control flows must alter the state of the race object.

Eliminating Race Conditions

Eliminating race conditions begins with identifying race windows.

A race window is a code segment that 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.

Race conditions are eliminated by making conflicting race windows mutually exclusive.

Mutual Exclusion

Once a potential race window begins execution, no conflicting race window can be allowed to execute until the first race window has completed.

Race windows are referred to as critical sections because it is critical that the two conflicting race windows not overlap execution.

Treating each critical section as an atomic unit with respect to conflicting race windows is called mutual exclusion.

Synchronization Primitives

Language facilities for implementing mutual exclusion are called synchronization primitives.

C and C++ support a variety of synchronization primitives:

- mutex variables
- semaphores
- pipes
- named pipes
- condition variables
- `CRITICAL_SECTION` objects
- lock variables

Incorrect use of synchronization primitives can lead to deadlock.

Deadlock

Deadlock occurs whenever two or more control flows block each other in such a way that none can continue to execute.

Deadlock results from a cycle of concurrent execution flows in which each flow in the cycle has acquired a synchronization object that results in the suspension of the subsequent flow in the cycle.

Deadlock can result in a denial-of-service attack.

- VU#132110 Apache HTTP Server versions 2.0.48 and prior contain a race condition in the handling of short-lived connections.
- When using multiple listening sockets, a short-lived connection on a rarely-used socket may cause the child process to hold the accept mutex, blocking new connections from being served until another connection uses the socket.

Exploiting Deadlock 1

Deadlock can result from altering

- processor speeds
- changes in the process or thread scheduling algorithms
- different memory constraints imposed at the time of execution
- any asynchronous event capable of interrupting the program’s execution
- the states of other concurrently executing processes

Exploiting Deadlock 2

Often an attack is an automated attempt to vary conditions until the race behavior is exposed.

By exposing the computer system to an unusually heavy load, it may be possible to effectively lengthen the time required to execute a race window.

The possibility of deadlock should always be viewed as a security flaw.

Agenda

Concurrency

Time of Check, Time of Use

Files as Locks and File Locking

File System Exploits

Mitigation Strategies

Summary

Trusted/Untrusted Control Flows

Race conditions can result from trusted or untrusted control flows.

Trusted control flows include tightly coupled threads of execution that are part of the same program.

An untrusted control flow is a separate application or process, often of unknown origin, that executes concurrently.

Multitasking Systems w/ Shared Resources

Any system that supports multitasking with shared resources has the potential for race conditions from untrusted control flows.

Time of check, time of use (TOCTOU) race conditions can occur during file I/O.

TOCTOU race conditions form a race window by first testing (checking) a race object property and then later accessing (using) the race object.

Linux TOCTOU Example

```
#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.

Race Window

File opened for writing

TOCTOU Exploit 1

An external process can change or replace the ownership of `some_file`.

If the vulnerable program is running with elevated privileges, a not-normally accessible file may be 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.

TOCTOU Exploit 2

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
- opens the file with `fopen()`
- checks to ensure that the file was opened successfully

Race Condition (GNU File Utilities v4.1)

```
chdir("/tmp/a");

chdir("b");

chdir("c");

chdir("..");

rmdir("c");

unlink("*");
```

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

Race Window

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

Agenda

Concurrency

Time of Check, Time of Use

Files as Locks and File Locking

File System Exploits

Mitigation Strategies

Summary

Files as Locks and File Locking

Race conditions from independent processes cannot be resolved by synchronization primitives.

Processes don’t have shared access to global data (such as a mutex variable).

These kinds of concurrent control flows can be synchronized using a file as a lock.

Simple File Locking in Linux

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.

Simple File Lock Function

`lock()` and `unlock()` are passed the name of a file that serves as the shared lock object

```
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");
}
```

Disadvantages of File Locking

A disadvantage of this implementation for a lock mechanism is that the `open()` function does not block.

The `lock()` function must call `open()` repeatedly until the file can be created.

- This repetition is sometimes called a busy form of waiting or a spinlock.
- Spinlocks consume compute time executing repeated calls and condition tests.

A file lock can remain indefinitely if the process holding the lock fails to call `unlock()`.

This could occur, for example, if a process crashed.

Solution for Removing Lock Files

The `lock()` function can be modified to write the process's ID (PID) in the lock file.

Upon discovering an existing lock, the new `lock()` searches the active process list for the saved PID.

If the process that locked the file has terminated

- the lock is acquired
- the lock file is updated to include the new PID

Problems remain…

- The PID of the terminated process may have been reused.
- The fix may contain race conditions.

Windows Synchronizing Processes

Windows supports a better alternative for synchronizing across processes— the named mutex object.

Named mutexes have a name space similar to the file system.

`CreateMutex()`

- is called with the mutex name
- creates a mutex object (if it didn't already exist) and returns the mutex handle

Acquire and release is accomplished by

- `WaitForSingleObject()`
- `ReleaseMutex()`

File Locks

Files, or regions of files, are locked to prevent two processes from concurrent access.

Windows supports file locking of two types:

- shared locks
  - prohibit all write access to the locked file region
  - allow concurrent read access to all processes
- exclusive locks
  - grant unrestricted file access to the locking process
  - deny access to all other processes

Windows File Locks

A call to `LockFile()` obtains shared access; exclusive access is accomplished via `LockFileEx()`.

The lock is removed by calling `UnlockFile()`.

Shared locks and exclusive locks eliminate the potential for a race condition on the locked region.

The exclusive lock is similar to a mutual exclusion solution.

The shared lock eliminates race conditions by removing the potential for altering the state of the locked file region.

Mandatory vs. Advisory Locks

Windows file-locking mechanisms are called mandatory locks, because every process attempting access to a locked file region is subject to the restriction.

Linux implements both mandatory locks and advisory locks.

An advisory lock is not enforced by the operating system, which severely diminishes its value from a security perspective.

Linux Mandatory Locks

The mandatory file lock in Linux is impractical for the following reasons:

- Mandatory locking works only on local file systems and does not extend to network file systems.
- File systems must be mounted with support for mandatory locking, and this is disabled by default.
- Locking relies on the group ID bit, which can be turned off by another process (thereby defeating the lock).

Agenda

Concurrency

Time of Check, Time of Use

Files as Locks and File Locking

File System Exploits

Mitigation Strategies

Summary

File System Exploits 1

Files and directories commonly act as race objects.

A file is opened, read from or written to, and closed by separate functions called over a period of execution time.

File access sequences are fertile regions for race windows.

Open files are shared by peer threads, and file systems have exposure to other processes.

File System Exploits 2

The exposure comes in the form of

- file permissions
- file naming conventions
- file system mechanisms

Most executing programs leave a file in a corrupted state whenever the program crashes.

This extreme form of a race condition is an unavoidable vulnerability and another reason for data backup.

Symbolic Linking Exploits 1

A symbolic link is a directory entry that references a target file or directory.

A symlink vulnerability involves a programmatic reference to a file name that unexpectedly turns out to include a symbolic link.

The most common symlink vulnerability involves a TOCTOU condition.

Symbolic Linking Exploits 2

TOCTOU vulnerabilities include:

- a call to `access()` followed by `fopen()`
- a call to `stat()` followed by a call to `open()`
- a file that is opened, written, closed, and reopened by a single thread

Within the race window, the attacker alters the meaning of the file name by creating a symbolic link.

TOCTOU Vulnerability with `stat()`

```
if (stat("/dir/some_file", &statbuf) == -1) {
   err(1, "stat");
}
```

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

```
if (statbuf.st_size >= MAX_FILE_SIZE) {
  err(2, "file size");
}
```

TOCTOU check occurs with the call of `stat()`

TOCTOU use is the call to `fopen()`

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

Exploiting TOCTOU Vulnerability

This vulnerability can be exploited by executing the following commands during the race window:

```
rm /dir/some_file
ln -s attacker_file /dir/some_file
```

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

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

Symbolic Linking Exploits

Symbolic links are used in exploits because they can be created even when the owner of the link lacks permissions to access the target file.

The attacker needs write permissions for the directory in which the link is created.

Symbolic links can reference a directory. The attacker might replace `/dir` with a symbolic link to a completely different directory.

C++ TOCTOU Vulnerability

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main() {
  ofstream outStrm;
  ifstream chkStrm;
  chkStrm.open("/tmp/some_file", ifstream::in);
  if (!chkStrm.fail())
    outStrm.open("/tmp/some_file", ofstream::out);
  ...
}
```

Can be exploited by the creation of a symbolic link, named `/tmp/some_file`, during the race window

Time of check

Time of use

O_CREAT and O_EXCL

If **O_CREAT** and **O_EXCL** are set, **open()** fails if the file exists.

The check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other threads executing **open()** naming the same filename in the same directory with **O_EXCL** and **O_CREAT** set.

If **O_EXCL** and **O_CREAT** are set, and *path* names a symbolic link, **open()** fails and set **errno** to [EEXIST], regardless of the contents of the symbolic link.

If **O_EXCL** is set and **O_CREAT** is not set, the result is undefined.

Mitigation for C++ File Open TOCTOU

```
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 with a file descriptor and not a file name.

C++ TOCTOU

In pre-standard C++, certain implementations of `<fstream.h>` offered the flags `ios::nocreate` and `ios::noreplace` for controlling file creation.

These flags were too platform-specific and never made it into the standard `<fstream>` library, which supersedes the deprecated, pre-standard `<fstream.h>` header.

Unfortunately, `fstream` has no atomic equivalent, following the deprecation of the `ios::nocreate` flag.

`unlink()` Race Exploit

Opening a UNIX file and unlinking it later creates a race condition.

By replacing the named open file with another file or symbolic link, an attacker can cause **`unlink()`** to be applied to the wrong file.

This problem can be avoided with proper permissions on the file’s containing directories.

Using `realpath()` with `pathconf()`

The `pathconf()` function can be used to determine the system-defined limit on the size of file paths.

This value may then be used to allocate memory to store the canonicalized path name returned by `realpath()`

However, a `PATH_MAX` value obtained from a prior `pathconf()` call is out-of-date by the time `realpath()` is called.

Agenda

Concurrency

Time of Check, Time of Use

Files as Locks and File Locking

File System Exploits

Mitigation Strategies

Summary

Mitigation Strategies

Race-related vulnerabilities can be mitigated by

- removing the concurrency property
- eliminating the shared object property
- controlling access to the shared object to eliminate the change state property

UNIX and Windows have synchronization primitives for the implementation of critical sections within a single multithreaded application.

Mutual Exclusion 1

Conflicting race windows should be protected as mutually exclusive critical sections.

Synchronization primitives capable of implementing critical sections within a single multithreaded application

- mutex variables
- semaphores
- pipes
- named pipes
- condition variables
- `CRITICAL_SECTION` objects
- lock variables

Mutual Exclusion 2

An object-oriented alternative for removing race conditions relies on the use of decorator modules to isolate access to shared resources.

This approach requires all access of shared resources to use wrapper functions that test for mutual exclusion.

Avoiding Deadlock

The standard avoidance strategy for deadlock is to require that resources be captured in a specific order.

Deadlock is avoided as long as no process may capture resource r_k unless it first has captured all resources r_j, where $j < k$.

In a multithreaded application, it is insufficient to ensure that there are no race conditions within the application’s own instructions.

It is also possible that invoked functions could be responsible for race conditions.

Use of Atomic Operations

Synchronization primitives rely on operations that are atomic.

These functions must not be interrupted until they have run to completion.

- `EnterCriticalRegion()`
- `pthread_mutex_lock()`

If the execution of one call to:

```
EnterCriticalRegion()
```

is allowed to overlap with a call from:

```
LeaveCriticalRegion()
```

there could be race conditions internal to these functions.

Identifying Files 1

UNIX files can often be identified by other attributes in addition to the file name, such as the file serial number (i-node) together with the device.

You can store information on a file that you have created and closed, and then use this information to validate the identity of the file when you reopen it.

Comparing multiple attributes of the file improves the probability that you have correctly identified the appropriate file.

Identifying Files 2

The POSIX `fstat()` function can be used to read information about the file into the `stat` structure and compare it with existing information about the file to improve identification.

The structure members `st_mode`, `st_ino`, `st_dev`, `st_uid`, `st_gid`, `st_atime`, `st_ctime`, and `st_mtime` should all have meaningful values for all file types on POSIX compliant systems.

The `st_ino` and `st_dev` structure members, taken together, uniquely identify the file.

- `st_ino` field contains the file serial number (i-node)
- `st_dev` field identifies the device containing the file

Identifying Files 3

```
struct stat st;
dev_t dev; /* device */
ino_t ino; /* file serial number */
int fd = open(filename, O_RDWR);
if ( (fd != -1) && (fstat(fd, &st) != -1) &&
     (st.st_ino == ino) && (st.st_dev == dev)
   ) {
     ...
}
```

NOTE: The `st_dev` value is not necessarily consistent across reboots or system crashes.

Checking File Properties Securely

TOCTOU conditions result from a need to check file properties.

Linux mitigation is to use `open()` followed by `fstat()`.

A Windows mitigation is to use

`GetFileInformationByHandle()`

rather than

`FindFirstFile()` or `FindFirstFileEx()`

The `lstat()` Function

The only Linux stat function to stat a symbolic link rather than its target

Must be applied to a filename with no file descriptor alternative

Is useful in guarding against symlink vulnerabilities

Symlink Check with TOCTOU

```
struct stat lstat_info;
int fd;
if (lstat(fn, &lstat_info) == -1) {
  err(1, "lstat");
}
if (!S_ISLNK(lstat_info.st_mode)) {
   if ((fd = open(fn, O_EXCL|O_RDWR, 0600)) == -1)
     err(2, fn);
}
// process the file
...
```

This code stats `fn`, then opens this file if it is not a symbolic link.

The `lstat()` followed by the `open()` form a TOCTOU condition.

A symbolic link could have been created between the calls to `lstat()` and `open()`

Symlink Check w/o TOCTOU

One mitigation strategy for calling `lstat()` securely is a four-step algorithm:

1. `lstat()` the file name
2. `open()` the file
3. `fstat()` the file descriptor from step 2
4. compare the status from steps 1 and 3 to ensure that the files are the same

Secure Symlink Check

```
struct stat lstat_info, fstat_info;
int fd;
if (lstat(fn, &lstat_info) == -1)
  err(1, "lstat");
if ((fd = open(fn, O_EXCL|O_RDWR, 0600)) == -1)
  err(2, fn);
if (fstat(fd, &fstat_info) == -1) err(3, "fstat");
if (lstat_info.st_mode == fstat_info.st_mode &&
    lstat_info.st_ino == fstat_info.st_ino &&
    lstat_info.st_dev == fstat_info.st_dev)
      // process the file
  ...
```

Ensures that the file stat-ed is the same file as the file that is opened

Why this Works

`fstat()` is applied to file descriptors, not file names, so the file fstat-ed must be identical to the file that was opened.

`lstat()` does not follow symbolic links, but `fstat()` does.

The mode (`st_mode`) defines the file type

- comparing modes is sufficient to check for a symbolic link as `lstat()` will return the link file while will `fstat()` will return the referenced file

Comparing i-nodes (`st_ino`) ensures that the lstat-ed file has the identical i-node as the fstat-ed file.

Eliminating the Race Object

Race conditions exist because some object is shared by concurrent execution flows.

If the shared object(s) can be eliminated or its shared access removed, there cannot be a race vulnerability.

Resources that are capable of maintaining state are of concern with respect to race conditions.

Any two concurrent execution flows of the same computer will share access to that computer’s devices and various system-supplied resources.

Know What Is Shared 1

Among the most important, and most vulnerable, shared resources is the file system.

Windows systems have another key shared resource in the registry.

System-supplied sharing is easy to overlook because it is seemingly distant from the domain of the software.

A program creating a file in one directory may be impacted by a race condition in a directory several levels closer to the root.

Know What Is Shared 2

A malicious change to a registry key may remove a privilege required by the software.

Often the best mitigation strategies for system-shared resources have more to do with system administration than software engineering.

- System resources should be secured with minimal access permissions.
- System security patches should be installed regularly.

Know What Is Shared 3

Programmers should minimize vulnerability exposure by removing unnecessary use of system-supplied resources.

For example:

- The Windows `ShellExecute()` relies on the registry to select an application to apply to the file.
- It is preferable to call `CreateProcess()`, explicitly specifying the application, than to rely on the registry.

Process-level concurrency increases the number of shared objects slightly.

Concurrent processes inherit global variables and system memory, including settings such as the current directory and process permissions, at the time of child process creation.

Descriptor Tables

A vulnerability associated with inheriting open files is that this may unnecessarily populate the file descriptor table of a child process.

The unnecessary entries could cause the child's file descriptor table to fill, resulting in a denial of service.

It is best to close all open files, excepting perhaps `stdin`, `stdout` and `stderr`, before forking child processes.

`ptrace()` Function

A process that executes `ptrace()` essentially has unlimited access to the resources of the trace target.

This includes access to all memory and register values.

Avoid the use of `ptrace()` except for applications like debugging, in which complete control over the memory of another process is essential.

Peer Threads 1

Concurrency at the thread level leads to the greatest amount of sharing and correspondingly the most opportunity for race objects.

Peer threads share all

- system-supplied shared objects
- process-supplied shared objects

Peer Threads 2

Peer threads also share all

- global variables
- dynamic memory
- system environment variables

To minimize exposure to potential race objects in threads, limit the use of

- global variables
- static variables
- system environment variables

File Descriptors 1

The race object in a file-related race condition is often not the file but the file’s directory.

A symlink exploit relies on changing the directory so that the target of a file name has been altered.

A file is not vulnerable to a symlink attack if it is accessed through its file descriptor and not the file name’s directory that is the object of the race.

File Descriptors 2

Many file-related race conditions can be eliminated by using:

- **`fchown()`** rather than **`chown()`**
- **`fstat()`** rather than **`stat()`**
- **`fchmod()`** rather than **`chmod()`**

POSIX functions that have no file descriptor counterpart should be used with caution.

- **`link()`** and **`unlink()`**
- **`mkdir()`** and **`rmdir()`**
- **`mount()`** and **`unmount()`**
- **`lstat()`**
- **`mknod()`**
- **`symlink()`**
- **`utime()`**

Shared Directories

When a group of users have write permission to a directory, the potential for sharing and deception is greater than it is for shared access to a few files.

The vulnerabilities that result from malicious restructuring via hard and symbolic links suggest avoiding shared directories.

Principle of Least Privilege 1

The principle of least privilege is a wise strategy for mitigating race conditions as well as other vulnerabilities.

Race condition attacks involve a strategy where the attacker performs a function without permission.

If the process executing a race window is not more privileged than the attacker, then there is little to be gained by an exploit.

Principle of Least Privilege 2

There are several ways the principle of least privilege can be applied to mitigate file I/O race conditions:

- Avoid running processes with elevated privileges.
- When a process must use elevated privileges, drop privileges before accessing shared resources.

Static Analysis Tools 1

A static analysis tool analyzes software for race conditions without actually executing the software.

The tool parses software, sometimes relying on user-supplied search information.

Static analysis tools report apparent race conditions, sometimes ranking each reported item according to perceived risk.

Static Analysis Tools 2

Warlock is a static tool for analyzing C programs that relies on extensive programmer annotations to drive the race condition identification.

ITS4 is an alternative that uses a database of known vulnerabilities, including race conditions.

Perceived vulnerabilities are flagged and a severity level is reported.

RacerX performs control-flow-sensitive inter-procedural analysis that provides coarse-grained detection best suited for large systems.

Among the best known public domain tools are

- Flawfinder http://www.dwheeler.com/flawfinder/
- RATS http://www.securesw.com/rats

Static Analysis Tools 3

Extended static checking (ESC) tools perform a static analysis based on theorem-proving technology, rather than the compiler-like parsing of most static tools.

All static analysis algorithms are prone to some false negatives (vulnerabilities not identified) and frequent false positives (incorrectly identified vulnerabilities).

Dynamic Analysis 1

Dynamic race detection tools overcome some of the problems with static tools by integrating detection with the actual program’s execution.

Analyzing only the actual execution flow has the additional benefit of producing fewer false positives that the programmer must consider.

The main disadvantages of dynamic detection are:

- fails to consider execution paths not taken
- significant runtime overhead

Eraser

Eraser is a dynamic tool that intercepts operations held on runtime locks.

Eraser reports alarms based on the analysis of a widely accepted algorithm for examining sets of held locks.

Program annotations are supported to prevent false positives from recurring in future program runs.

MultiRace - Alcatraz

MultiRace uses an improved version of the lockset algorithm used by Eraser together with an algorithm derived from a static detection technique.

This combination is claimed to reduce false positives. MultiRace also improves on the runtime overhead of Eraser.

The Alcatraz tool maintains a file modification cache that isolates the actual file system from unsafe access.

User input is required to commit unsafe file changes.

Thread Checker - RaceGuard

Thread Checker performs dynamic analysis for thread races and deadlocks on both Linux and Windows C++ code.

RaceGuard is a UNIX kernel extension designed to provide for secure use of temporary files.

RaceGuard maintains its own cache of processes and their temporary files.

Execution-time file probes and opens are intercepted by RaceGuard and aborted if an attack is detected.

Mitigation Strategies

If you are operating:

- In someone else's directory, relinquish elevated privileges
  - If you are root (or administrator), set your effective user ID to that of the directory's owner for file operations in that directory
    - assuming that the directory is secured for that user; otherwise, you may endanger that user's files
  - If you are not root, you may be at risk of attacks against files you own elsewhere
    - don't operate on files in other user's directories
- In a shared directory such as **`/tmp`**, consider using
  - a temporary directory inside your home directory
  - a secured directory for root or administrator temporary files

Agenda

Concurrency

Time of Check, Time of Use

Files as Locks and File Locking

File System Exploits

Mitigation Strategies

Summary

Summary 1

Race conditions are subtle, difficult to discover, and frequently exploitable. Their problem is concurrency.

Concurrent code is more complex than sequential code.

- It is more difficult to write,
- more difficult to comprehend,
- more difficult to test.

There are no “silver bullets” when it comes to avoiding race conditions.

Summary 2

The vulnerabilities of race conditions can be divided into two major groups:

- those that are caused by the interactions of the threads within a multithreaded process
- vulnerabilities from concurrent execution outside the vulnerable software

The primary mitigation strategy for vulnerability to trusted threads is to eliminate race conditions using synchronization primitives.

Summary 3

Race conditions from untrusted processes are the source of many well-known file-related vulnerabilities:

- symlink vulnerability
- various vulnerabilities related to temporary files

Synchronization primitives are of little value for race vulnerabilities from untrusted processes.

Mitigation requires various strategies designed to eliminate the presence of shared race objects and/or carefully restrict access to those race objects.

Summary 4

Many tools have been developed for locating race conditions either statically or dynamically.

Most static tools produce significant numbers of false positives and false negatives.

Dynamic tools have a large execution-time cost and are incapable of discovering race conditions outside the actual execution flow.

Race detection tools have proven their ability to uncover race conditions even in heavily tested code.

Questions about Race Conditions

Backup

File Descriptors

A per-process unique, non-negative integer used to identify an open file for the purpose of file access.

The value of a file descriptor is from zero to **`OPEN_MAX`**.

A process can have no more than **`OPEN_MAX`** file descriptors open simultaneously.

File descriptors may also be used to implement message catalog descriptors and directory streams.

Open File Description

An open file description is a record of how a process or group of processes is accessing a file.

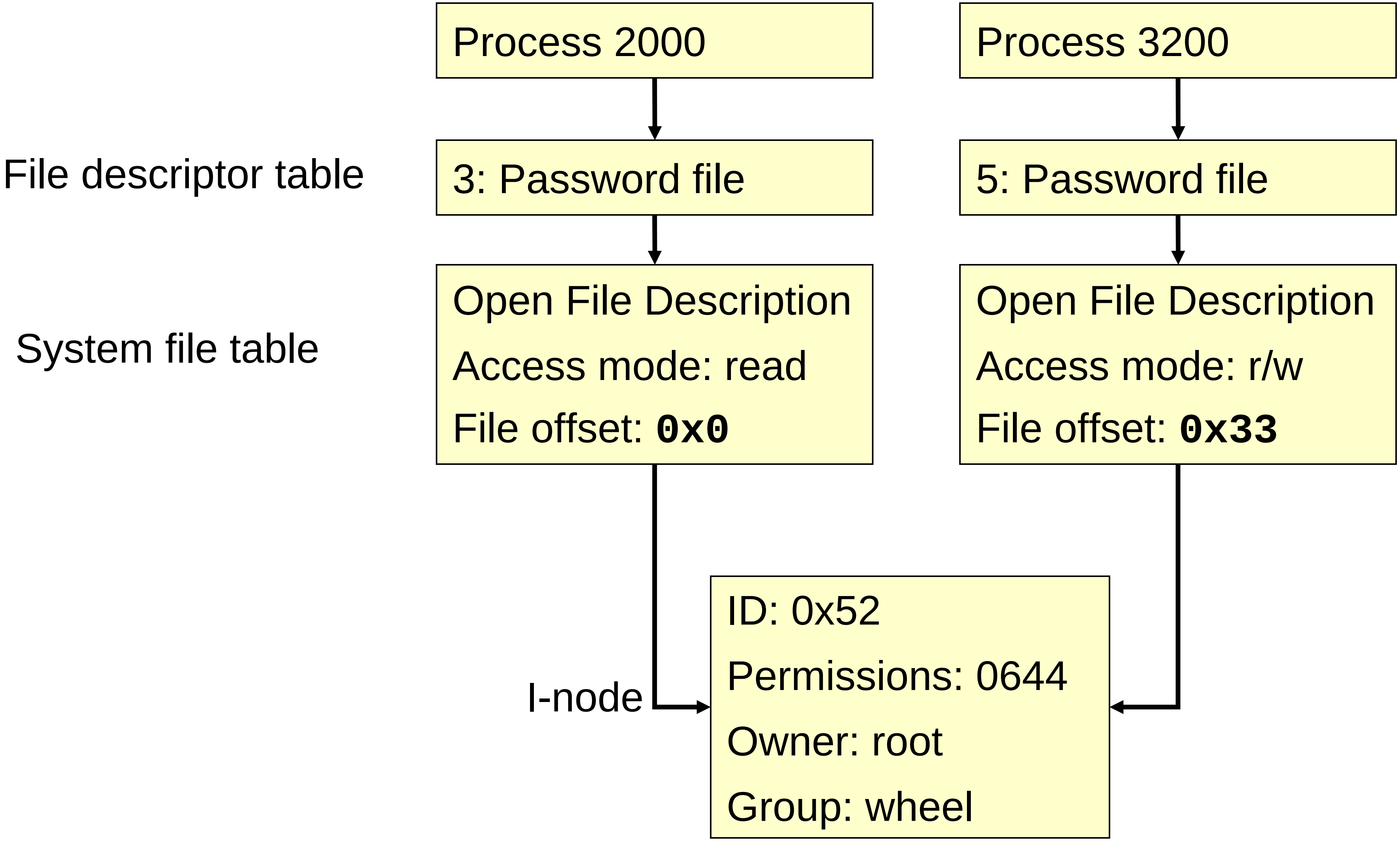
A file descriptor is actually just an identifier or handle; it does not actually describe anything.

Each file descriptor refers to exactly one open file description, but an open file description can be referred to by more than one file descriptor.

Attributes of an open file description include

- file offset
- file status
- file access modes

Independent Opens of the Same File



File descriptor table

System file table

I-node

Process 2000

3: Password file

Open File Description

Access mode: read

File offset: **`0x0`**

Process 3200

5: Password file

Open File Description

Access mode: r/w

File offset: **`0x33`**

ID: 0x52

Permissions: 0644

Owner: root

Group: wheel