Secure Programming (06-20010)
Chapter 6: Race Conditions

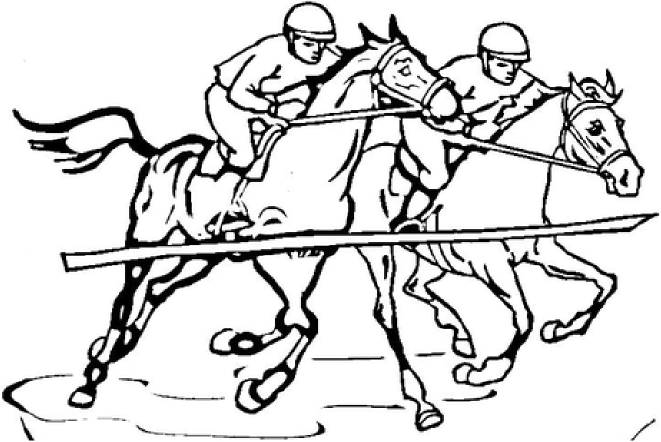
Christophe Petit

University of Birmingham

Lectures Content (tentative)

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

Race Conditions



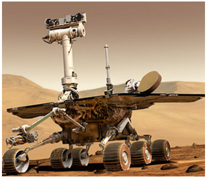
Picture source : `www.supercoloring.com/coloring-pages/race-horse`

Race conditions

- We tend to think of programs as executing in a linear way, without interruption
- However process scheduling can affect execution at any time, for any amount of time
- Attacker may affect scheduling by exhausting CPU
- Attacker may modify filesystem and environment during program execution

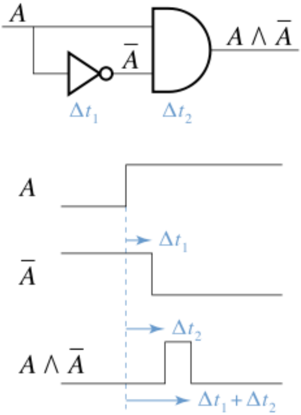
Race conditions : Impact

- Serious functionality bugs



- Serious security vulnerabilities, such as privilege escalation

Race condition : Electronics



Picture source : Wikipedia

Race Conditions (2)

- "Anomalous behaviour due to unexpected critical dependence on the relative timings of events"
- Can be created by an adversarial process, or simply result from synchronization failure in your code
- Typical adversarial examples : TOCTOU races
- Typical synchronization failure examples : deadlocks, database synchronization failure

Outline

Secure file opening

Locking bugs

Outline

Secure file opening

- Vulnerability description
- Protection mechanisms

Locking bugs

Outline

Secure file opening

- Vulnerability description
- Protection mechanisms

Locking bugs

open

```
OPEN(2)                 Linux Programmer's Manual                OPEN(2)

NAME    top

       open, openat, creat - open and possibly create a file

SYNOPSIS    top

       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);

       int openat(int dirfd, const char *pathname, int flags);
       int openat(int dirfd, const char *pathname, int flags, mode_t mode);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       openat():
           Since glibc 2.10:
               _POSIX_C_SOURCE >= 200809L
           Before glibc 2.10:
               _ATFILE_SOURCE

DESCRIPTION    top

       Given a pathname for a file, open() returns a file descriptor, a
       small, nonnegative integer for use in subsequent system calls
       (read(2), write(2), lseek(2), fcntl(2), etc.).  The file descriptor
       returned by a successful call will be the lowest-numbered file
       descriptor not currently open for the process.
```

- Permissions checked/granted based on **effective** UID

access : checks real UID

ACCESS(2) Linux Programmer's Manual ACCESS(2)

NAME top

access, faccessat - check user's permissions for a file

SYNOPSIS top

```
#include <unistd.h>

int access(const char *pathname, int mode);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <unistd.h>

int faccessat(int dirfd, const char *pathname, int mode, int flags);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
faccessat():
    Since glibc 2.10:
        _POSIX_C_SOURCE >= 200809L
    Before glibc 2.10:
        _ATFILE_SOURCE
```

DESCRIPTION top

access() checks whether the calling process can access the file *pathname*. If *pathname* is a symbolic link, it is dereferenced.

The *mode* specifies the accessibility check(s) to be performed, and is either the value **F_OK**, or a mask consisting of the bitwise OR of one or more of **R_OK**, **W_OK**, and **X_OK**. **F_OK** tests for the existence of the file. **R_OK**, **W_OK**, and **X_OK** test whether the file exists and grants read, write, and execute permissions, respectively.

The check is done using the calling process's *real* UID and GID, rather than the effective IDs as is done when actually attempting an operation (e.g., open(2)) on the file. Similarly, for the root user, the check uses the set of permitted capabilities rather than the set of effective capabilities; and for non-root users, the check uses an empty set of capabilities.

- Access returns 0 if **real** user ID has required permissions

Checking permissions with access

- Suppose you want to check permissions of **real** UID
- Is the following C code secure ?

```
if (access("filename", W_OK) != 0) {
   exit(1);
}
fd = open("filename", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- W_OK : test for write permissions
  O_WRONLY : open only for writing

A typical race condition

- Is the following C code secure ?

```c
if (access("filename", W_OK) != 0) {
   exit(1);
}
fd = open("filename", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- Something unexpected may happen between access check and file opening

Time Of Check to Time Of Use (TOCTOU)

- Typical race condition : something unexpected may happen between access check and when the file is used
- Adversary might be able to replace file called "filename" by another one after the access check
- Using symlinks, an adversary might redirect "filename" to a file with root privileges such as etc/shadow
- These attacks require local access to the system and precise timings, but are possible

Remember : symbolic links (symlinks)

- Symlinks are references to other files
- Automatically resolved by the operating system
- Every user on the local system can create symlinks
  - Link target does not need to be owned by user
  - User needs write permission on the directory where they create the symlink

symlink

symlink(3) - Linux man page

Prolog

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

Name

symlink - make a symbolic link to a file

Synopsis

#include <unistd.h>

int symlink(const char **path1*, **const char** ***path2*);**

Description

The *symlink*() function shall create a symbolic link called *path2* that contains the string pointed to by *path1* (*path2* is the name of the symbolic link created, *path1* is the string contained in the symbolic link).

The string pointed to by *path1* shall be treated only as a character string and shall not be validated as a pathname.

If the *symlink*() function fails for any reason other than [EIO], any file named by *path2* shall be unaffected.

Return Value

Upon successful completion, *symlink*() shall return 0; otherwise, it shall return -1 and set *errno* to indicate the error.

Overwriting etc/passwd

Victim	Attacker
`if (access("file", W_OK) != 0) { exit(1); }` `fd = open("file", O_WRONLY);` `// Actually writing over /etc/passwd` `write(fd, buffer, sizeof(buffer));`	`//` `//` `// After the access check` `symlink("/etc/passwd", "file");` `// Before the open, "file" points to the password database` `//` `//`

Victim:

```
if (access("file", W_OK)
!= 0) {
   exit(1);
}

fd = open("file",
O_WRONLY);
// Actually writing over
/etc/passwd
write(fd, buffer,
sizeof(buffer));
```

Attacker:

```
//
//
// After the access check
symlink("/etc/passwd", "file");
// Before the open, "file" points to the
password database
//
//
```

Picture source : Wikipedia

Explanation

- access checks permissions for real user ID
- open only checks permissions for effective user ID
- symlink creates a symbolic link from "file" to etc/passwd
- root has access rights on etc/passwd
- Impact ?
  - Privilege escalation : normal user acquires root rights
  - Deny-of-service

Success requires precise timing

- Attack successful if attacker's code executed during TOCTOU window (= time between check and use)
- To improve attack success probability
  - Slow down computer with CPU-expensive programs
  - Run many attack processes in parallel

Outline

Secure file opening

- Vulnerability description
- Protection mechanisms

Locking bugs

Countermeasures

- Use atomic operations
- Decrease success probability : check-use-check again
- Drop permissions : let operating system make the checks
- Use unpredictable file names

Use atomic operations

- Check and use permission within single system call
- Secure code to create a new file

```
fd = open("filename", O_CREAT|O_EXCL|O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- Opening will fail if a file with that name already exists

If O_CREAT and O_EXCL are set, *open()* will fail if the file exists. The check for the existence of the file and the creation of the file if it does not exist will be atomic with respect to other processes executing *open()* naming the same filename in the same directory with O_EXCL and O_CREAT set.

Safe opening in other languages

- Other languages support similar APIs for file handling

C#	Look for the System.IO.FileMode parameter
Java	Look for the OpenOptions parameter
Python	os.open

Check-use-check-again approach

- Idea : detect file modifications using stat, lstat, fstat
  1. Get file information before opening
  2. Open the file
  3. Get file information after opening
  4. Compare file information and abort if it changed
- Attacker can defeat this by restoring the original file, but this now requires to succeed in two races
- Increase number of checks to reduce success probability

stat

STAT(2)   Linux Programmer's Manual   STAT(2)

NAME top

stat, fstat, lstat, fstatat - get file status

SYNOPSIS top

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);

#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>

int fstatat(int dirfd, const char *pathname, struct stat *statbuf,
            int flags);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
lstat():
    /* glibc 2.19 and earlier */ _BSD_SOURCE
        || /* Since glibc 2.20 */ _DEFAULT_SOURCE
        || _XOPEN_SOURCE >= 500
        || /* Since glibc 2.10: */ _POSIX_C_SOURCE >= 200112L

fstatat():
    Since glibc 2.10:
        _POSIX_C_SOURCE >= 200809L
    Before glibc 2.10:
        _ATFILE_SOURCE
```

DESCRIPTION top

These functions return information about a file, in the buffer pointed to by *statbuf*. No permissions are required on the file itself, but—in the case of **stat**(), **fstatat**(), and **lstat**()—execute (search) permission is required on all of the directories in *pathname* that lead to the file.

Stat structure

```
The stat structure
    All of these system calls return a stat structure, which contains the
    following fields:

        struct stat {
            dev_t     st_dev;         /* ID of device containing file */
            ino_t     st_ino;         /* Inode number */
            mode_t    st_mode;        /* File type and mode */
            nlink_t   st_nlink;       /* Number of hard links */
            uid_t     st_uid;         /* User ID of owner */
            gid_t     st_gid;         /* Group ID of owner */
            dev_t     st_rdev;        /* Device ID (if special file) */
            off_t     st_size;        /* Total size, in bytes */
            blksize_t st_blksize;     /* Block size for filesystem I/O */
            blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

            /* Since Linux 2.6, the kernel supports nanosecond
               precision for the following timestamp fields.
               For the details before Linux 2.6, see NOTES. */

            struct timespec st_atim;  /* Time of last access */
            struct timespec st_mtim;  /* Time of last modification */
            struct timespec st_ctim;  /* Time of last status change */

        #define st_atime st_atim.tv_sec      /* Backward compatibility */
        #define st_mtime st_mtim.tv_sec
        #define st_ctime st_ctim.tv_sec
        };
```

Check-use-check-again approach (2)

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int main()
{
        struct stat statBefore, statAfter;
        lstat("/tmp/X", &statBefore);
        if (!access("/tmp/X", O_RDWR)) {
                /* the real UID has access right */
                int f = open("/tmp/X", O_RDWR);
                fstat(f, &statAfter);
                if (statAfter.st_ino == statBefore.st_ino)
                { /* the I-node is still the same */
                        write_to_file(f);
                }
                else perror("Race Condition Attacks!");
        }
        else fprintf(stderr, "Permission denied\n");
}
```

Code source : www.cis.syr.edu/~wedu/Teaching/CompSec/LectureNotes_New/Race_Condition.pdf

Drop permissions with seteuid

- Let the operating system handle permissions for you
- Use seteuid to temporarily drop to real UID privileges

```
#include <unistd.h>
#include <sys/types.h>
uid_t real_uid = getuid();
uid_t effective_uid = geteuid();
seteuid (real_uid);
f = open("/tmp/X", O_WRITE);
if (f != -1)
        write_to_file(f);
else
        fprintf(stderr, "Permission denied\n");
seteuid (effective_uid);
```

Code source : www.cis.syr.edu/~wedu/Teaching/CompSec/LectureNotes_New/Race_Condition.pdf

Use unpredictable file names

- Idea : if attacker cannot guess the filename you use, they cannot build the proper symlink
- tempnam_r : replace filename by "unpredictable" name, such that a file with this name does not exist

```
if (tmpnam_r(filename)){
        FILE* tmp = fopen(filename,"wb+");
        ...
}
```

- However : race condition still exists, and filename not totally unpredictable
- Better to use mkstemp, which returns a file descriptor

mkstemp

```
MKSTEMP(3)                Linux Programmer's Manual                MKSTEMP(3)
```

NAME top

mkstemp, mkostemp, mkstemps, mkostemps - create a unique temporary file

SYNOPSIS top

```
#include <stdlib.h>

int mkstemp(char *template);

int mkostemp(char *template, int flags);

int mkstemps(char *template, int suffixlen);

int mkostemps(char *template, int suffixlen, int flags);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
mkstemp():
    _XOPEN_SOURCE >= 500
        || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
        || /* Glibc versions <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE

mkostemp(): _GNU_SOURCE
mkstemps():
    /* Glibc since 2.19: */ _DEFAULT_SOURCE
        || /* Glibc versions <= 2.19: */ _SVID_SOURCE || _BSD_SOURCE
mkostemps(): _GNU_SOURCE
```

DESCRIPTION top

The **mkstemp**() function generates a unique temporary filename from *template*, creates and opens the file, and returns an open file descriptor for the file.

The last six characters of *template* must be "XXXXXX" and these are replaced with a string that makes the filename unique. Since it will be modified, *template* must not be a string constant, but should be declared as a character array.

The file is created with permissions 0600, that is, read plus write for owner only. The returned file descriptor provides both read and write access to the file. The file is opened with the open(2) **O_EXCL** flag, guaranteeing that the caller is the process that creates the file.

The **mkostemp**() function is like **mkstemp**(), with the difference that the following bits—with the same meaning as for open(2)—may be specified in *flags*: **O_APPEND**, **O_CLOEXEC**, and **O_SYNC**. Note that when creating the file, **mkostemp**() includes the values **O_RDWR**, **O_CREAT**, and **O_EXCL** in the *flags* argument given to open(2); including these values in the *flags* argument given to **mkostemp**() is unnecessary, and produces errors on some systems.

The **mkstemps**() function is like **mkstemp**(), except that the string in *template* contains a suffix of *suffixlen* characters. Thus, *template* is of the form *prefixXXXXXXsuffix*, and the string XXXXXX is modified as for **mkstemp**().

The **mkostemps**() function is to **mkstemps**() as **mkostemp**() is to **mkstemp**().

RETURN VALUE top

On success, these functions return the file descriptor of the temporary file. On error, -1 is returned, and *errno* is set appropriately.

Outline

Secure file opening

Locking bugs

Remember : Race Conditions

- "Anomalous behaviour due to unexpected critical dependence on the relative timings of events"
- Can be created by an adversarial process, or simply result from synchronization failure in your code
- Typical adversarial examples : TOCTOU races
- Typical synchronization failure examples : deadlocks, database synchronization failure

Example : incrementing a global value

- Suppose two threads want to increment a global variable

- Intended execution

Thread 1	Thread 2		Value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	0
	increase value		0
	write back	→	2

- Possible race condition

Thread 1	Thread 2		Value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

- Need synchronization mechanism between threads to enforce atomicity

Databases

- A database server handles simultaneous queries from multiple users
- Need synchronization mechanism to ensure
  - Each request is executed as intended
  - All users have the same view of the database

Static methods

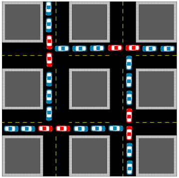
- Static methods/variables are methods/variables that belong to the class (as opposed to the object)
- Race conditions may occur when static variables accessed simultaneously by various threads

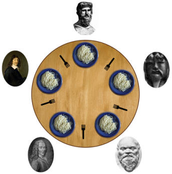
Common issue : non atomic operations

- Need synchronization mechanism between threads to enforce atomicity

Locks

- Idea : when resource (register, file, database, variable. . .) is used by a process, lock it to prevent further use
- Locks can be implemented with files containing locking status (simple to use, easy to unlock manually)
- Beware of classical locking issues : deadlocks & lifelocks





Pictures source : Wikipedia

Locks : Windows and Unix

- Unix
  - Both shared ("reading") and exclusive ("writing") locks
  - Not mandatory by default (can be ignored)
  - see fcntl, flock, lockf
- Windows
  - Windows file system prevents write or delete access on executing files
  - Share-access controls for whole-file access-sharing for read, write, or delete
  - Byte-range locks to arbitrate read and write access to regions within a single file

Record locking in Unix

Advisory record locking

Linux implements traditional ("process-associated") UNIX record locks, as standardized by POSIX. For a Linux-specific alternative with better semantics, see the discussion of open file description locks below.

F_SETLK, **F_SETLKW**, and **F_GETLK** are used to acquire, release, and test for the existence of record locks (also known as byte-range, file-segment, or file-region locks). The third argument, *lock*, is a pointer to a structure that has at least the following fields (in unspecified order).

```
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK,
                        F_WRLCK, F_UNLCK */
    short l_whence;  /* How to interpret l_start:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock
                        (set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

The *l_whence*, *l_start*, and *l_len* fields of this structure specify the range of bytes we wish to lock. Bytes past the end of the file may be locked, but not bytes before the start of the file.

l_start is the starting offset for the lock, and is interpreted relative to either: the start of the file (if *l_whence* is **SEEK_SET**); the current file offset (if *l_whence* is **SEEK_CUR**); or the end of the file (if *l_whence* is **SEEK_END**). In the final two cases, *l_start* can be a negative number provided the offset does not lie before the start of the file.

l_len specifies the number of bytes to be locked. If *l_len* is positive, then the range to be locked covers bytes *l_start* up to and including *l_start*+*l_len*-1. Specifying 0 for *l_len* has the special meaning: lock all bytes starting at the location specified by *l_whence* and *l_start* through to the end of file, no matter how large the file grows.

Locks : C#

```
class Account {  // this is a monitor of an account

    long val = 0;
    object thisLock = new object();

    public void deposit(const long x) {
        // only one thread at a time may execute next statement
        lock(thisLock) {
            val += x;
        }
    }

    public void withdraw(const long x) {
        // only one thread at a time may execute next statement
        lock(thisLock) {
            val -= x;
        }
    }

}
```

Code source : Wikipedia

Locks : other languages

C	POSIX Threads
Objective-C	@synchronized
VB.NET	SyncLock
Java	synchronized
Python	mutex mechanism
Ruby	mutex object
x86 assembly	LOCK prefix
PHP	Mutex class

Databases : transactions

- Sequence of operations that can be perceived as a single logical operation on the data
- Implemented by locking resources and keeping a (partial) copy until the transaction completes
- Satisfy ACID properties :
  - Atomicity : "either all or nothing"
  - Consistency : any transaction brings the database from one valid state to another valid state
  - Isolation : result as if transactions executed sequentially
  - Durability : transaction remains effective once committed

Transactions in MySqli

mysqli::begin_transaction

mysqli_begin_transaction

(PHP 5 >= 5.5.0, PHP 7)

mysqli::begin_transaction -- mysqli_begin_transaction — Starts a transaction

Description

Object oriented style (method):

```
public bool mysqli::begin_transaction ([ int $flags [, string $name ]] )
```

Procedural style:

```
bool mysqli_begin_transaction ( mysqli $link [, int $flags [, string $name ]] )
```

Transactions in MySqli (2)

mysqli::commit

mysqli_commit

(PHP 5, PHP 7)

mysqli::commit -- mysqli_commit — Commits the current transaction

Description

Object oriented style

```
bool mysqli::commit ([ int $flags [, string $name ]] )
```

Procedural style

```
bool mysqli_commit ( mysqli $link [, int $flags [, string $name ]] )
```

Commits the current transaction for the database connection.

Transactions in MySqli (3)

mysqli::rollback

mysqli_rollback

(PHP 5, PHP 7)

mysqli::rollback -- mysqli_rollback — Rolls back current transaction

Description

Object oriented style

```
bool mysqli::rollback ([ int $flags [, string $name ]] )
```

Procedural style

```
bool mysqli_rollback ( mysqli $link [, int $flags [, string $name ]] )
```

Rollbacks the current transaction for the database.

Transactions in MySqliL : example

```
<?php
$all_query_ok=true; // our control variable

//we make 4 inserts, the last one generates an error
//if at least one query returns an error we change our control variable
$mysqli->query("INSERT INTO myCity (id) VALUES (100)") ? null : $all_query_ok=false;
$mysqli->query("INSERT INTO myCity (id) VALUES (200)") ? null : $all_query_ok=false;
$mysqli->query("INSERT INTO myCity (id) VALUES (300)") ? null : $all_query_ok=false;
$mysqli->query("INSERT INTO myCity (id) VALUES (100)") ? null : $all_query_ok=false; //duplicated PRIMARY KEY VALUE

//now let's test our control variable
$all_query_ok ? $mysqli->commit() : $mysqli->rollback();

$mysqli->close();
?>
```

Code source : php.net/manual/en/mysqli.rollback.php

Race Condition : Detection

- Race conditions are difficult to reproduce and debug : result is probabilistic and depends on relative timing
- Static analysis tool : Thread Safety Analysis, used in gcc and Clang
- Dynamic analysis tools : Thread Safety Analysis, Intel Inspector, Intel Advisor, Helgrind
- See `www.owasp.org/index.php/Testing_for_Race_Conditions_(OWASP-AT-010)`

Helgrind

<< Home Page

Information Source Code **Documentation** Contact How to Help Gallery

| Table of Contents | Quick Start | FAQ | User Manual | Download Manual | Research Papers | Books |

Valgrind User Manual

7. Helgrind: a thread error detector

Table of Contents

7.1. Overview
7.2. Detected errors: Misuses of the POSIX pthreads API
7.3. Detected errors: Inconsistent Lock Orderings
7.4. Detected errors: Data Races

- 7.4.1. A Simple Data Race
- 7.4.2. Helgrind's Race Detection Algorithm
- 7.4.3. Interpreting Race Error Messages

7.5. Hints and Tips for Effective Use of Helgrind
7.6. Helgrind Command-line Options
7.7. Helgrind Monitor Commands
7.8. Helgrind Client Requests
7.9. A To-Do List for Helgrind

To use this tool, you must specify `--tool=helgrind` on the Valgrind command line.

7.1. Overview

Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives.

The main abstractions in POSIX pthreads are: a set of threads sharing a common address space, thread creation, thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers.

Summary

- Race conditions produce anomalous behaviour due to unexpected critical dependence on events' relative timings

- TOCTOU = time of check to time of use
- Secure file opening

```
fd = open("filename", O_CREAT|O_EXCL|O_WRONLY);
```

- Use synchronization mechanisms such as locks to protect against non adversarial race conditions

References

- David Wheeler, Secure Programming How To, Chapter 7.11
- Viega-MacGraw, Building Secure Software, Chapter 9
- `www.cis.syr.edu/~wedu/Teaching/CompSec/LectureNotes_New/Race_Condition.pdf`

Acknowledgements

- While preparing this course I used teaching material developed by Erik Tew at the University of Birmingham (kindly provided to me) and Meelis Roos at Tartu University (available on the web)
- Some of my slides are heavily inspired from theirs (but blame me for any errors !)