



RACES

How race conditions happen



Requirements for race condition

- Two or more processes have access to the same object
- Algorithm used by processes does not properly enforce an access order
- At least one process modifies the object



Simple Java servlet

```
import          java.io.*;
import          javax.servlet.*;
import          javax.servlet.http.*;
public class Counter extends HttpServlet {
    int    count = 0;
    public void  doGet(HttpServletRequest in, HttpServletResponse out)
                throws ServletException, IOException {
        out.setContentType("text/plain");
        PrintWriter    p = out.getWriter();
        count++;
        p.println(count + " hits so far!");
    }
}
```



Modification to Java servlet

```
import          java.io.*;
import          javax.servlet.*;
import          javax.servlet.http.*;
public class Counter extends HttpServlet {
    int    count = 0;
    public void  doGet(HttpServletRequest in, HttpServletResponse out)
                throws ServletException, IOException {
        out.setContentType("text/plain");
        PrintWriter    p = out.getWriter();
        p.println(++count + " hits so far!"); ←
    }
}
```



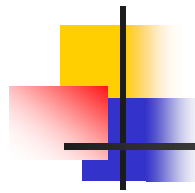
Race Conditions

- In a pre-emptively multi-tasked environment, anything can happen in-between the execution of two statements
 - Check if something is OK to do
 - Do it (perhaps the conditions have changed?)
- Semaphores and locks are mechanisms that prevent concurrent access to, or modification of, an object by different processes



To fix race conditions

- Race condition occurs when a certain condition assumed true does not hold
- *Window of vulnerability*: interval of time when violation of assumption leads to incorrect behavior
- Reduce window to zero: make relevant code atomic
- An operation that cannot be interrupted with regards to an object is called "atomic"



Java synchronized

- The synchronized keyword ensures that only a single thread will execute a statement or block at a time
 - Prevents thread from observing object in inconsistent state
 - Enforces appropriate sequencing of state transitions
- The JVM implementation is responsible for enforcing it
- It can have a significant impact on efficiency




Revised Java servlet

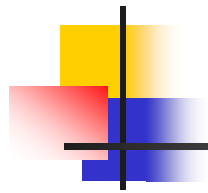
```
import          java.io.*;
import          javax.servlet.*;
import          javax.servlet.http.*;
public class Counter extends HttpServlet {
    int    count = 0;
    public synchronized void doGet(HttpServletRequest in, HttpServletResponse out)
        throws ServletException, IOException {
        out.setContentType("text/plain");
        PrintWriter    p = out.getWriter();
        count++;
        p.println(count + " hits so far!");
    }
}
```




Improved Java servlet

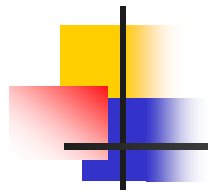
```
public class Counter extends HttpServlet {
    int          count = 0;
    public void  doGet(HttpServletRequest in, HttpServletResponse out)
                    throws ServletException, IOException {
        int my_count;
        out.setContentType("text/plain");
        PrintWriter p = out.getWriter();
        synchronized(this) {
            my_count = ++count;
        }
        p.println(my_count + " hits so far!");
    }
}
```





Other Example Race Condition

- User 1 creates a file with world-writable permissions
- User 1 wants to change the permissions to exclude others with `"chmod 700 filename"`
- User 2 tries to overwrite the file in-between
- Will user 1 or user 2 succeed?
 - User 1 should have set the umask correctly!



Database Race Condition

- If (condition for field 1)
 then do something to field 2
- However process 2 changes field 1 in-between...
- Result: invalid combination of values
(e.g., bank account balance)



Example Race Condition

- Two processes: red and blue
- Red: Check that user 1 has enough money to pay check #y
- Blue: Check that user 1 has enough money to pay check #x
- Red: Pay check #y
- Is there really enough money to pay check #x?



Effects of Race Conditions

- Normally:
 - race conditions show up as periodic errors
 - frequency of the error will depend upon how likely the 'bad' order is to occur
 - it is often hard to get race condition errors to repeat
- When exploited:
 - crackers can attempt to force the particular conditions that will produce a flaw
 - depending upon the exact form of the flaw, it may be produced with high probability
 - most common (mis)use: modify the value of some shared object



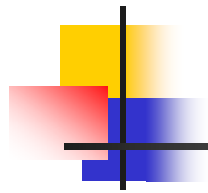
Example: `mkdir`.

- `mkdir` is actually a fairly complex sequence of actions. Here are some of the actions triggered by `mkdir dir` under an early implementation for a Unix system:
 - Superuser creates a directory object `dir`
 - Permissions are set to `777`.
 - Superuser does a `chown` to change the ownership of the directory to the calling user.
 - Modify permissions via `umask(2)` to match the environment `umask` value.
- Can you spot the problem??



One difficulty ...

- From the man pages:
 - *The mkdir command creates specified directories in mode 777. The directories are then modified by umask(2), according to how you have set up umask.*
- Patient crackers can automate a process to exploit this race condition to obtain ownership of file. Here is a well-known method:
 - Find a writable directory
 - Start a scanning program that will look for creation of `/tmp/junk`
 - Start up `mkdir /tmp/junk` and use `nice` to cause it to run slowly in the background. Move scanner to foreground.



Flaw continued ...

- When the scanner spots the new directory:
 1. Remove the original `/tmp/junk`
 2. Create a link from the secret file you want to `/tmp/junk`
- Suppose the scanner's link in (2) beat the `mkdir`'s `chown`. Now `mkdir` will change the ownership of the secret file to cracker.



Why do these problems arise?

- Problem: there are many of these race conditions in operating systems. Many occur in common system utilities.
 - It is a lot of trouble to identify and then fix them. Often the fix will cause systems to run slower, since it is necessary to coordinate access. Most sites do not have anyone with the time or ability to do so properly.
- The above explains why it is important that all users of a system be trustworthy.



Yet another example

- Using same buffer for plaintext and ciphertext
 - Load buffer with plaintext
 - Encrypt buffer
 - Send buffer contents to recipient
- Looks harmless, until in multithreaded application, last two steps swapped and plaintext is sent
- Impossible? Not for Internet Information Server 4 when using SSL (occasional unencrypted packet sent)
 - www.microsoft.com/technet/security/bulletin/MS99-053.asp
- Use two buffers, zeroing out the ciphertext buffer across calls

Another classic source of (security) problems

- **race condition** aka **data race** is a common type of bug in concurrent programs
 - basically: two execution threads mess with the same variable at the same time
 - not necessarily a *security* bug
- **Non-atomic check and use** aka **TOCTOU (Time Of Check, Time of Use)** a closely related type of security flaw

Problem: some precondition required for an action is invalidated between the time it is checked and the time the action is performed

 - typically, this precondition is access control condition
 - typically, it involves some concurrency

Race condition

- Two concurrent execution threads both execute the statement
 $x = x+1;$
where x initially has the value 0.
- What is the value of x in the end?
- Answer: x can have the value 2 or 1
- The root cause of the problem is that $x = x+1$ is not an **atomic operation**, but happens in two steps, reading x and assigning the new value, which may be **interleaved** in unexpected ways
- Why can this lead to security problems?
- Think of internet banking, and running two simultaneous sessions with the same bank account... *Do try this at home!* 😊

Classic example race condition: `mkdir` on Unix

- `mkdir` is setuid root, ie. executes as root
- It creates new directory *non-atomically*, in two steps:
 1. creates the directory, with owner is root
 2. sets the owner, to whoever invoked `mkdir`
- Attack: by creating a *symbolic link* between steps 1 and 2, attacker can own any file

Other classic UNIX race conditions

- **lpr**
 - print utility with option to remove file after printing
 - could be used to delete arbitrary files
- but this still happens
 - **CVE-2003-1073**
A race condition in the at command for Solaris 2.6 through 9 allows local users to delete arbitrary files via the -r argument with .. sequences in the job name, then modifying the directory structure after at checks permissions to delete the file and before the deletion actually takes place

Combination of race condition with failure to check that file names do not contain ..

Example race condition

```
const char *filename="/tmp/erik";  
if (access(filename, R_OK) != 0) {  
    ... // handle error and exit;  
}  
// file exists and we have access  
int fd open (filename, O_RDONLY);  
...
```

Between calls to **access** and **open** the file might be removed!

Race condition & file systems

Signs of trouble:

- Access to files using **filenames** rather than **file handles or file descriptors**
 - filenames may point to different files at different moments in time
- Creating files or directories in publicly accessible places, for instance `/tmp`
 - especially if these have predictable file names

Promising future for data races?

- Trend: **more multi-CPU machines**
 - to keep improving computer power in accordance with Moore's Law, despite physical constraints
- Hence: **more multi-threaded software**
 - to take advantage of max. available computing power
- Hence: **many problems with data races in code**
 - as programmers cannot cope with concurrency
 - writing correct concurrent programs is **hard**

[Interesting article to read:

"The free lunch is over :a fundamental turn toward concurrency in software", by Herb Sutter]