



## RACES and LINKS

---



## Simple Race Condition

---

	<b>Process 1</b>	<b>Process 2</b>
1	X:= 0 ;	X:= 0 ;
2	Do something...	Do something...
3	X:= X + 1 ;	X:= X - 1 ;
4	Print x	Print x



## 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!");
    }
}
```

The keyword synchronized prevents multiple threads from running code in the same object.



## 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!");
    }
}
```

The keyword synchronized is limited to as small a block of code as possible.



## 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)



## 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



## Types of modifications:

- Change owner or other attribute of a file
- Change file read/written to
- Modify data in file
- Add information to file



## 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.



## Another example

- Aside: not all race conditions are in software. Here is an old hardware problem mentioned on alt.hacker:

*The vending machines at school allows you to charge snacks to your student ID. First, you run your ID through a magnetic scanner and then the machine connects to a server (through a phone jack in the back) and checks to see if you have any credit. Once your ID clears, you simply unplug the jack, click on Snickers, and un/re-plug the machine.*

Obviously, this just another way to shoplift :(



## 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](http://www.microsoft.com/technet/security/bulletin/MS99-053.asp)
- Use two buffers, zeroing out the ciphertext buffer across calls



## File System Vulnerabilities

- Most common attack vectors:
  - Symlink attacks (234 entries as of April 2004)
  - Directory traversal attacks (252 entries)
- Other attack vectors:
  - Information leakage
    - Recycled disk space and buffers
    - File descriptors
  - Insecure file permissions (configuration issue)
  - File system "mounting" issues (OS issue)



## Symlink Attacks: Outline

- Symbolic links and hard links
- Basic symlink attack
  - Known or predictable file name
  - Defense: Randomness
- Symlink attacks on insecure temporary files
  - Race conditions (148 entries in ICAT)
  - Defense: Atomic operations



## File System Links

---

To create a link:

- In [-fhns] source\_file [target\_file]

Two types of links:

- Hard links
- Symbolic links

What is the difference?



## File System Links

---


- Hard links
  - Windows: CreateHardLink
  - UNIX: ln
- Symbolic links
  - UNIX:
    - ln -s
  - Windows:
    - a.k.a. "directory junctions" in NTFS



## Hard Links

---

- Indistinguishable from original entry
- May not refer to directories or span file systems
- Created link is subject to the same, normal file access permissions.
- Deleting a hard link does not delete the file unless all references to the file have been deleted
- A reference is either a hard link or an open file descriptor



## Example

---

- `% ls -al .localized`  
`-rw-r--r-- root meuser .localized`
- `% ln .localized pascal/hard.loc`
- `% ls -al pascal/hard.loc`  
`-rw-r--r-- root meuser hard.loc`
- `% rm pascal/hard.loc`  
override `rw-r--r-- root/meuser` for `pascal/hard.loc`? Yes
- `% ls -al .localized`  
`-rw-r--r-- root meuser .localized`
- Note that the hard link showed the same permissions
- Note that deleting the hard link did not delete the file (the reference count was not zero)



## Symbolic Links

- Windows:
  - Directory junctions apply to directories only
    - Can refer to directories on different computers
  - Jargon: "File system reparse points"
  - Contain parameters resolved at access time
  - Several operations, complex setup (see <http://www.sysinternals.com/ntw2k/source/misc.shtml#junction>)
- UNIX:
  - Contain a path, which is resolved at access time
  - May refer to directories and files
  - May span file systems
  - Permissions appear different from the original



## Symbolic Link Example

- Using the same starting file as for the hard link example:
- ```
% ln -s .localized pascal/sym.loc
```
- ```
% ll pascal/sym.loc
```

```
lrwxr-xr-x  1 pascal  staff  pascal/sym.loc
```

```
-> .localized
```
- Note:
  - The "->"
  - The permissions (see the "l"?)
  - The owner and group are different (they were root/meuser for ".localized")
  - Deleting the symlink doesn't delete the file



## Power of Symbolic Links

- You can create links to files that don't exist yet
- Symlinks continue to exist after the files they point to have been renamed, moved or deleted
  - They are not updated
- You can create links to arbitrary files, even in file systems you cannot see
- Symlinks can link to files located across partition and disk boundaries
- Example:
  - You can change the version of an application in use, or even an entire web site, just by changing a symlink



## Basic Attack

- Trick a process (with higher privileges) to operate on a file different from the one it thinks it is.
- Example:
  - Create the link "temp -> /etc/password"
  - A privileged process executes
    - truncate("temp", 0)
      - The "truncate" call follows symlinks
      - Changes the length of the file "temp" to 0
    - But truncated /etc/password instead!
      - Note that the *contents* are deleted, not the file
  - Can be used for write or read operations
    - Or deletion if the symlink is in the path and not the end point



## Conditions of Vulnerability

- If you are operating in a secured directory, you do not need to worry about symlink attacks
- A secured directory is one with permissions of all the directories, from the root of the file system to your directory, set so that only you (or root) can make changes in your secured directory
  - Example: /home/me (user home directories are usually set by default with secure permissions)
- You are at risk if you operate
  - In a shared directory such as /tmp
  - In someone else's directory, especially with elevated privileges
    - Example: an anti-virus program running as administrator



## Suggested Workarounds

- 1) Store the file in a secured directory
    - It was stored in a shared directory
  - 2) Relinquish root privileges before doing file operations (if not needed)
  - 3) Use a random name (!)
  - 4) Create files with "umask 077"
    - New files with no permissions to groups and others
- What about third-party components that you utilize?





## Best Defenses

---

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
    - Do not operate on files in other user's directories
- In a shared directory such as /tmp, consider using instead
  - A temporary directory inside your home directory
  - A secured directory for root or administrator temporary files



## UNIX Filenames vs File Descriptors

---

- Filenames and directory structure are changeable
- Open file descriptors are fixed
  - System calls that use a file descriptor are to be used whenever possible instead of the equivalent functionality using paths
    - `fchmod(int fd, mode_t mode);`
    - `fchown(int fd, uid_t owner, gid_t group);`
    - `fchdir(int fd);`
    - `fstat(int fd, struct stat *sb);`
  - File descriptors specify an inode (see next slide)



## inodes

---

- An inode is a data structure containing user, group and access control information (and more)
- The inode specifies the location of the file on the disk
- Hard links associate a name to an inode
  - Several hard links can point to a single inode
    - There is no difference between the "first" hard link and other ones
- Inodes are deleted only when all references have been deleted
  - Open file descriptors and hard links count as references
  - Directories also have inodes



## Hard Links in UNIX

---

- By creating hard links, an attacker could make you:
  - Change the permissions of an unintended file
  - Change the contents of an unintended file
- Defenses:
  - Manipulate files inside safe directories (with correctly set permissions)
  - Do not open and manipulate files as root if you don't need to
  - Do not re-open temporary files in shared directories (more on this later)



## Deleting a UNIX File

```
unlink(char *path);
```

- Deletes a hard link
- File (and inode) is actually deleted when the link count reaches zero
- Unlink (the file deletion call) follows symlinks!
  - Safely deleting a file is difficult if not done in a secured directory
- Deletes a symbolic link if the link is the last component of the path
  - Does not affect intermediate symlinks!



## Safely Deleting a File

- Issue: A file you want to delete is not deleted if someone else made a hard link to it
- Scenario:
  - You have learned that a setuid program is vulnerable, so you want to delete it (and perhaps install a new version)
  - An attacker could still exploit it by making a hard link to it
- Safe Delete:
  - Remove the setuid/setgid bits
  - Unlink it (rm)
  - Reboot



## Istat

---

- `lstat(char *path, struct stat *sb);`
- Istat returns information on symbolic links
  - Istat takes a path, which may contain symbolic links before the last item!
  - The struct contains file information
    - inode
    - file type "S\_IFLNK" indicates that the last part of the path is a symlink
- Using Istat before deleting a file sounds interesting



## Deleting a File

---

- Given the following pseudo-code:

```
lstat (intermediate_directory, sb);  
  
if (sb says it's not a symlink) {  
    unlink("intermediate_directory/myfile");  
}
```
- What can happen?



## Answer

---

- Race conditions between lstat and other operations are possible
- Someone may replace a file (or directory) with a symlink in between (depending on access permissions)
  - If you are root and manipulating someone else's files, they may do this!
- Chroot may help limit which files may be affected, but does not prevent the problem (there may still be interesting files to target inside the chroot jail)



## To open arbitrary files

---

- lstat() the file before opening it, saving the stat structure
- Perform open()
- fstat() the file descriptor returned by open()
- Compare fields in two stat structures
  - st\_mode
  - st\_ino
  - st\_dev
- If comparisons successful, lstat() called on the opened file