

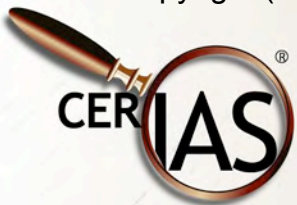
Course 2: Programming Issues, Section 5

Pascal Meunier, Ph.D., M.Sc., CISSP

Updated April 4, 2006

Developed thanks to the support of Symantec Corporation,
NSF SFS Capacity Building Program (Award Number 0113725)
and the Purdue e-Enterprise Center

Copyright (2004) Purdue Research Foundation. All rights reserved.



Course 2 Learning Plan

- Buffer Overflows
- Format String Vulnerabilities
- Code Injection and Input Validation
- Cross-site Scripting Vulnerabilities
- **Links and Race Conditions**
- Temporary Files and Randomness
- Canonicalization and Directory Traversal

Learning objectives

- Learn how symlinks work
- Learn how symlinks may fool a program
- Learn how race conditions happen
- Learn how symlinks and race conditions work together
- Learn how to defend against symlink attacks and how to avoid race conditions

Material Not Covered Here

- The following has been covered in other units:
 - File permissions
 - Umask
 - Setuid files
 - Relative and absolute paths
 - Environment variables
 - ❖ Path

Introduction to 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)

Mounting File Systems (Side Note)

- Some file systems can be mounted with a "nosuid" option
- Safer if you don't trust setuid or setgid programs on that file system -- the setuid and setgid bits are ignored

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

- Hard links
 - Windows: CreateHardLink
 - UNIX: ln
- Symbolic links
 - UNIX:
 - ❖ ln -s
 - Windows:
 - ❖ a.k.a. "directory junctions" in NTFS
 - ❖ Manually: use Linkd.exe (Win 2K resource kit) or "junction" freeware
- Virtual Drives
 - "subst" command

Virtual Drives (Windows)

- Similar to symbolic links but limited functionality
- Effect
 - A drive letter ("X:") actually points to a directory on a physical drive
 - Overcomes limits on path length
 - ❖ NTFS has limit of 32000 characters
 - ❖ Windows has a limit of 256 characters
 - NT, 2000, XP
 - "Path too long" error
 - ❖ Malicious software can hide inside very long paths
 - Setup using a "subst" virtual drive
 - Remove drive, and virus scanner can't find it!
 - <http://www.securiteam.com/windowsntfocus/5QP08156AQ.html>

Subst Vulnerability

- Virtual drives persist after a user logs off (NT)
- If next user tries to use that same letter drive, they may use a folder of the attacker's choosing
 - Store confidential files
 - Run trojans
- Example: Network-mapped home drive
 - Mounting operation used to fail silently if the drive letter was already in use
 - CVE-1999-0824

Hard Links

- Indistinguishable from original entry (peer)
- May not refer to directories or span file systems
- Created link is subject to the same, normal file access permissions.
- Deleting a hard link doesn't delete the file unless all references to the file have been deleted
- A reference is either a hard link or an open file descriptor
- In Windows, a hard link exists at the NTFS file system level and is not supported by FAT32
 - Note: Different from a Windows shortcut

Example

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

## Hard Link Vulnerability Example

- Fool audit logging programs by using a hard link to a sensitive file
- Audit trails record benign name instead of sensitive file access
- CVE-2002-0725 (NTFS)

## Hard Links – Windows Lab

- Create a file “original.txt” and put some text in it
- Create a link to “original.txt” and call it “secondary.txt”
  - Use `fsutil hardlink create secondary.txt original.txt`
  - Or write a program to create a hard link using `CreateHardLink()`  
[http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/hard\\_links\\_and\\_junctions.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/hard_links_and_junctions.asp)
- Show that you can edit “original.txt”, save, and when you open “secondary.txt”, it has the same changes.

## Hard Links – UNIX/Cygwin Lab

- Create a file “original.txt” and put some text in it.
- Create a link to “original.txt” and call it “secondary.txt”
  - Use `ln original.txt secondary.txt`
- Show that you can edit “original.txt”, save, and when you open “secondary.txt”, it has the same changes.

# 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/wheel 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 can't 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
    - ❖ Very convenient!

## Basic Attack

- Trick a process (with higher privileges) to operate on another file than 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 don't 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 such 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

## Example: CUPS Vulnerability

- CVE-2002-1366
- Common Unix Printing System (CUPS) 1.1.14 through 1.1.17 allows local users with lp privileges to create or overwrite arbitrary files via file race conditions, as demonstrated by "ice-cream".
- Predictable file name:
  - '/etc/cups/certs/<pid>'
  - Shared directory with users that have "lp" privilege
    - ❖ "lp" privilege could be gained through another exploit
- File manipulated using root privileges
  - ❖ Symlink redirected operations anywhere and allowed gaining root privileges

## 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 will give no permissions to groups and others
- What about third-party components that you utilize?

## Best Defenses (Both Windows and UNIX)

- 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 instead
    - ❖ A temporary directory inside your home directory
    - ❖ A secured directory for root or administrator temporary files

## Windows Example

- Recursive deletion utility: "rd"
- Scenario:
  - Attacker makes a link from c:\temp\tempdir to c:\windows\system32 or any other sensitive directory
  - Administrator does "rd /s c:\temp"
  - Sensitive directory is erased!
    - ❖ (Example from Howard and Leblanc 2003)
- Note: Unlike the Windows "rd" utility, the UNIX command "`rm -rf`" does not traverse symlinked directories when deleting
  - But what happens when you do "`rm -rf`" on a Windows directory junction from within a Cygwin shell?

## Symlink File Write Example: Netscape 6.01

- The Netscape installer (running as root) creates a file in /tmp using a predictable name
- Attack: create a number of symlinks to cover the probable names, all pointing to a file to be deleted
  - The installer will delete the target!
    - ❖ CVE-2001-1066
- Which installers are you using that might have similar vulnerabilities?

## Question

- Let's say you have a setuid program "myprogram" that uses a temporary file, /tmp/mytemp
- What happens when attacker does this?
  - In -s /somedir/somefile /tmp/mytemp
  - /usr/bin/myprogram &
- a) Nothing
- b) There's a symlink pointing to "/somedir/somefile", and "myprogram" runs
- c) "myprogram" may perform operations on "/somedir/somefile" instead of the intended temporary file

## Other Example: XFree86 startx

- CVE-1999-0433 Symlink vulnerability
- Xfree86 runs as root, creates a temporary file
- What will this do?
  - `ln -s /dev/hd0 /tmp/.tX0-lock`
  - `startx`
- Note: /dev/hd0 refers to hard disk

## Answer

- XFree86 will write its temporary file to the raw device, messing up the file system...

## Symlink File Reading Example

- Digital Unix “msgchk” (setuid) checks to see if you have new messages, using your file \$HOME/.mh\_profile
- Predictable file name: .mh\_profile
- Attacker does:
  - `ln -s target_to_read .mh_profile`
- msgchk displays part of target if privileges of process allow it, as an error message
  - CERT/CC VU#440539

## Exercise

- Examine the installer script named "find\_java.sh"
  - Location provided by instructor
- What does it do?
  - What should the variable \$jvm contain?
    - ❖ What is the source of its value?
  - What should the variable \$VAR contain?
    - ❖ What is the source of its value?
- Which program does the script really run?
  - How can this be exploited?

## Exercise Answers

- The script executes a program named "java"
- Through links, another program may be invoked, masquerading as the real java
  - This program could check if it has root privileges and do nasty things when it does
    - ❖ During the install \*or\* later
    - ❖ Pass control to the real java when done, so as not to raise suspicions

# Windows Defenses

- Check directory attributes
  - `DWORD GetFileAttributes ( LPCTSTR lpFileName );`
  - `FindFirstFile`
- The return value contains ORed attributes
- The "FILE\_REPARSE\_POINT" attribute indicates a directory junction
  - Is it OK to traverse it?
  - Where does it point to?
  - Should the program operate there?

# Problem

- What happens between the attribute check, and the subsequent actions on the file?
- Similar problem for UNIX

## 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
- An operation that can't be interrupted with regards to an object is called "atomic"

## 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?

## Mini-lab: Database Locks

- Objective: Demonstrate a contention mechanism preventing race conditions
- Outline
  - You will create a test database and table
  - You will lock the table from one shell
  - Notice that operations on the same table from another shell block until the original lock is released

# MySQL

- Start a root shell
  - Click on the Knoppix CD icon (penguin, second to right)
- Type "mysqld\_safe &"
- Type "mysql -u root"
- Create the database
  - Type "create database ci;"
  - Type "use ci;"
- Note: This is **not** a recommended setup for a real MySQL installation. However, the MySQL user tables are read-only on the Knoppix CD, so we'll use the MySQL user "root" without a password (!)

## Mini-lab: Database Locks

- Create a table
  - Create table users (

```
uid VARCHAR (20) NOT NULL,
PRIMARY KEY (uid)
);
```

)
- Type "lock table users write;"
- Now open another terminal, login again to MySQL
- Try locking the table. What happens?

## Mini-lab: Database Locks

- Type "unlock tables;" in the first window
- What happens in the second window?
- Type "lock table <tablename> write;" again in the first window. What happens?
- Type "unlock tables;" in the second window
- What happens in the first window?
- This mini-lab should have demonstrated how table locks prevent race conditions
- Can you apply this to file locks, in case two instances of your program are running?
  - How do you create a file without race conditions?

## Race Conditions in Windows

- Calling "CreateFile" with the
  - "CREATE\_NEW" flag will create a file, and fail if it exists
    - ❖ Good for file locks
    - ❖ But where did you just create it?
      - In /tmp/adir/?
        - » What if adir was replaced with a directory junction?
  - "OPEN\_EXISTING" flag will open an existing file, and fail if the file doesn't exist
    - ❖ But which file did you really open?
- Other users can create (given permissions)
  - Directory junctions
  - Hard links

## Deletion in Windows

- All file deletion functions in Windows use a path, and there's no flag to forbid following directory junctions
- You think you are deleting  
"C:\My Documents\MyJunction\important.file"  
but in fact you may be deleting  
"C:\WINDOWS\SYSTEM32\IMPORTANT.FILE"
- Note: Junctions are deleted by calling  
RemoveDirectory(dir\_path)

## Windows: Changing Attributes

- The call uses a file name, which may point to something unexpected through directory junctions or hard links
- ```
BOOL SetFileAttributes(  
    LPCTSTR lpFileName,  
    DWORD dwFileAttributes  
);
```
- lpFileName is a path

Windows: Deleting a File

- `BOOL DeleteFile(
 LPCTSTR lpFileName
);`
- Argument is a path
- Will delete a hard link and not the file pointed to
 - Good
- May delete something unexpected by following directory junctions
 - Bad, but note that there are legitimate uses for junctions
- Make sure that there are no unexpected directory junctions in the path
 - How do you do that? (see next slides!)

Windows Information by Handle

- `BOOL GetFileInformationByHandle (`
 `HANDLE hFile,`
 `LPBY_HANDLE_FILE_INFORMATION`
 `lpFileInformation`
 `);`
- The handle is reliable (not interpreted, unchanging)
- "FILE_ATTRIBUTE_REPARSE_POINT" attribute will tell you if you are dealing with a reparse point
- How do you get a handle for the reparse point itself (and not for the directory it points to)?
 - (see next slide)

Opening a File: CreateFile

- It's like using the "start" button to log out
- ```
HANDLE CreateFile(
 LPCTSTR lpFileName,
 DWORD dwDesiredAccess,
 DWORD dwShareMode,
 LPSECURITY_ATTRIBUTES
lpSecurityAttributes,
 DWORD dwCreationDisposition,
 DWORD dwFlagsAndAttributes,
 HANDLE hTemplateFile
);
```
- The "FILE\_FLAG\_OPEN\_REPARSE\_POINT" allows you to open a reparse point *without interpretation*

## What About Race Conditions?

- So you checked the handle for an intermediate directory, and it's not a reparse point
- What prevents someone from replacing it with a reparse point **after** you've checked?
- MSDN says: "The RemoveDirectory function marks a directory for deletion on close. Therefore, the directory is not removed until the last handle to the directory is closed."
  - "Subsequent calls to CreateFile fail with ERROR\_ACCESS\_DENIED."

# The Windows Directory Crawl

- Principle:
  - As long as you keep the intermediate directory handles open, you can secure the path to the file you want

## Hard Links in Windows

- MSDN says:
  - Once two files are linked together, you cannot determine which is the original file and which is the copy.
- By creating hard links, an attacker could make you:
  - Change the attributes of an unintended file
  - Change the contents of an unintended file
- Defenses:
  - Manipulate files inside safe directories (with correctly set acls)
  - Don't manipulate files as administrator if you don't need to
  - Don't re-open temporary files in shared directories
    - ❖ Not to be confused with the ReOpenCall, which is safe because it uses a handle instead of a path

## Conclusions

- If someone else can modify (delete and recreate as directory junctions) directories on the path you are using, you are in trouble
  - You are safe only if the directory is secure, as per the earlier definition (slide 14)
- You can secure the path against directory junctions by opening handles to each intermediate directory and checking that they are not directory junctions

# 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's no difference between the "first" hard link and others
- 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)
  - Don't open and manipulate files as root if you don't need to
  - Don't re-open temporary files in shared directories (more on this later)

## Question

- In Unix, which one of these allows an attacker to fool you into deleting a different file than you intended?
- a) Hard links
- b) Symbolic links in the path, excluding the file itself
- c) Directory junctions

## Question

- Which one of these allows an attacker to fool you into deleting a file in UNIX?
- a) Hard links
  - No, if an attacker creates additional hard links, the file won't be deleted unless all hard links are deleted
- **b) Symbolic links (symlinks)**
  - Yes. Symbolic links can make you delete another file, especially if they happen in the middle of a path specification
- c) Directory junctions
  - Directory junctions are a Windows technology (NTFS), not UNIX (UFS). The possibility of mounting of NTFS file systems in Linux/UNIX is not considered here.

## 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 (side topic)

- 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, or executing it and stopping it with a SIGSTOP signal
  - Attacker can't copy it because copy removes the setuid bit
- Safe Delete:
  - Remove the setuid/setgid bits
  - Unlink it (rm)
  - Reboot

# Istat

- Istat(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 Istat 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 (there may still be interesting files to target inside the chroot jail)

UNIX Shared Directories

- The permissions of shared directories should have the "sticky" bit set
- Effect of the sticky bit:
 - Only the file owner, or root, can delete a file, regardless of the permissions
 - It makes it safer if you need to reopen a file because the file can't be replaced (easily) by another hard link or a symlink (more on this later)
- Note that the file contents can still be changed by others if the file is world-writable
 - Still need to set the correct permissions at creation time
 - ❖ e.g., umask

The Problem of Temporary File Sweepers

- Goal of sweepers: delete old temporary files that are not needed and are taking space in shared directories
- Run as root
 - Can't be run safely for deleting temporary files owned by root
- Risk deleting files that are still needed
- All have race conditions
- Cause additional security problems for programs using temporary files
- See Zalewski M. (2002),
<http://www.packetstormsecurity.org/papers/unix/tmpwatch.txt>

Doing Without File Sweepers

- Unlink temporary files immediately after creating them
 - If your process is terminated, the files go away
 - Not perfect, a race condition is still present

When Is It Safe to Delete?

- 1) When a file is inside a secured directory (see slide 15)
- 2) When the file is inside a "sticky" directory and was created securely
 - Directories with the "sticky" bit only allow the owner and root to delete the file
 - **Condition:** the administrator must not use temporary directory sweepers like "tmpwatch"
 - For more details see David Wheeler's race conditions chapter at <http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/avoid-race.html>

UNIX Defenses: Creating a file

- `int open(const char *path, int oflag, mode_t mode);`
- The flags should be `O_CREAT | O_EXCL`
 - If the last path item is a symbolic link and `O_CREAT` and `O_EXCL` are set, the link is **not** followed.
 - The check for symlinks and the file creation with the correct permissions (mode) are not subject to race conditions (it's an atomic operation)
- The mode should be as narrow as possible
- *Problem:* You know the last part isn't a symlink, but what about symlinks in-between?
 - If you are creating a file directly below `/tmp`, it is usually safe to assume that `/tmp` is not a malicious symlink

Mini-Lab: File Locks

- Download the file lock demo program
- Look at it; notice the "sleep" instruction
- Compile it (with gcc)
- Open another terminal
- Try to run it in both terminals simultaneously. What happens?
- Delete the file lock. What happens if you replace it with a symlink?
 - `cd /tmp`
 - `rm mylock; ln -s sometestfile mylock`

Mini-Lab: File Locks

- This lab should have demonstrated:
 - How to safely create a file lock
 - How a file lock can prevent two copies of the same program from running at the same time, or doing the same thing

UNIX Directories

- The current directory of a process is specified by the directory's inode
- If you carefully set and control the current process directory, it is safer to refer to it than using absolute paths
 - something in the absolute path could be or become a symlink
- `fchdir(int fd);`
 - Changes the current working directory to the directory referenced by the file descriptor `fd`
- The starting point for *some* relative path searches is the *process* current directory (refer to the man pages).

The UNIX Directory Crawl

- Algorithm:
 - Start at the filesystem root
 - Loop – Get the file descriptor of a directory just one level below
 - Do an "lstat" on that same name
 - ❖ lstat returns file information without following symlinks
 - Do an "fstat" on the file descriptor
 - ❖ Get the file information for the directory you just opened
 - Check that the inodes and devices match, and that lstat indicates that the file is not a symlink
 - Change the current directory with "fchdir", using the file descriptor
 - LOOP until you reach the file or directory you want

Cryogenic Sleep

- Setuid programs can receive signals if either the effective or real uid matches that of the sender
- SIGSTOP can put a process to "sleep"
 - Use SIGSTOP to change file system as needed for passing tests
 - ❖ e.g., until a particular inode gets reused
 - ❖ Opening the directory first guarantees that the inode can't be reused
 - If the `lstat` was done first, there would be a race condition between the `lstat` and `fopen` and `fchdir`
 - ❖ The inode and device check makes it difficult to exploit but not completely impossible
 - See "Symlinks and Cryogenic Sleep" by Olaf Kirch

A Better Directory Crawl

- For OSes that support the `O_NOFOLLOW` flag in `open` (do not follow a symlink)
 - OpenBSD, FreeBSD, Linux
- Algorithm:
 - Start at the filesystem root
 - Loop – Get the file descriptor of a directory just one level below, using the `O_NOFOLLOW` flag
 - Change the current directory with `"fchdir"`, using the file descriptor
 - LOOP until you reach the file or directory you want

Conclusions:

- Note that hard links can still possibly be used to fool your program
 - Checking the link count on the inode can help
 - ❖ e.g., `statbuf.st_nlink`
 - ❖ Could your program be "attacked" by creating hard links specifically for this check to fail?
 - e.g., web site "DoS" on a shared server
- Safer to never reopen a file in a temporary directory
 - Create it securely and unlink it as soon as possible
 - ❖ You still have access to it through the file descriptor
- Don't "dig" more than one level down an unsecured or shared directory (without `O_NOFOLLOW`)

Summary of UNIX Dangers

- Unlink follows a path
 - There needs to be a version where a file descriptor can be passed in addition, to verify the inode in an atomic operation
- Open doesn't have a flag to not follow symlinks inside a path (instead of just the end)
 - Need to do the directory crawl
- Safe directory crawl
 - Mount the filesystem in read only mode if possible
 - Use O_NOFOLLOW
- There is no way to tell if it is safe to delete another process' temporary files (tmp sweepers)

Questions or Comments?

§

About These Slides

- You are free to copy, distribute, display, and perform the work; and to make derivative works, under the following conditions.
 - You must give the original author and other contributors credit
 - The work will be used for personal or non-commercial educational uses only, and not for commercial activities and purposes
 - For any reuse or distribution, you must make clear to others the terms of use for this work
 - Derivative works must retain and be subject to the same conditions, and contain a note identifying the new contributor(s) and date of modification
 - For other uses please contact the Purdue Office of Technology Commercialization.
- Developed thanks to the support of Symantec Corporation



Pascal Meunier **pmeunier@purdue.edu**

Contributors:

Jared Robinson, Alan Krassowski, Craig Ozancin, Tim Brown, Wes Higaki, Melissa Dark, Chris Clifton, Gustavo Rodriguez-Rivera

