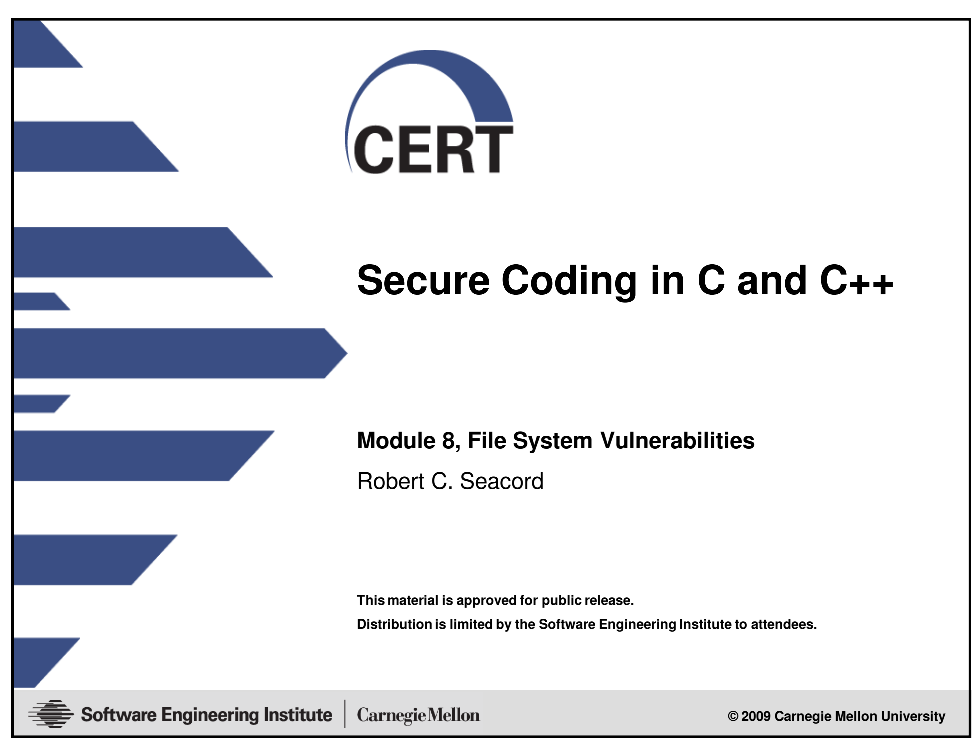
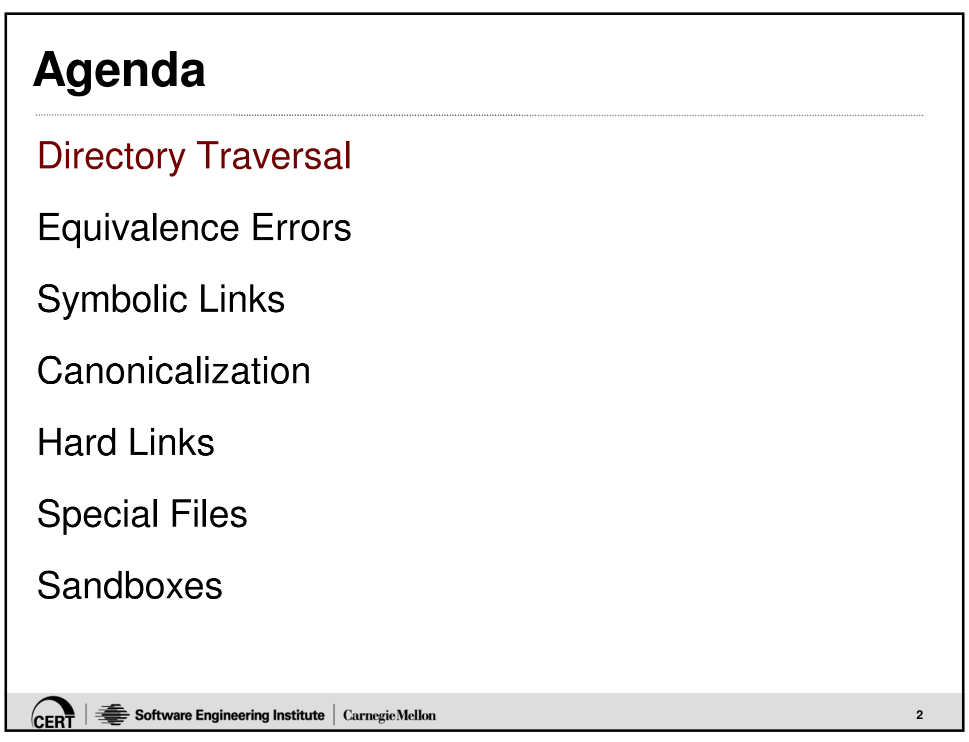

CERT
Secure Coding in C and C++
Module 8, File System Vulnerabilities
Robert C. Seacord
This material is approved for public release.
Distribution is limited by the Software Engineering Institute to attendees.
Software Engineering Institute
CarnegieMellon
© 2009 Carnegie Mellon University


Agenda
Directory Traversal
Equivalence Errors
Symbolic Links
Canonicalization
Hard Links
Special Files
Sandboxes
CERT
Software Engineering Institute
CarnegieMellon
2

Path Names

Absolute path names: If the path name begins with a slash, the predecessor of the first file name in the path name is the root directory of the process.

Relative path names: If the path name does not begin with a slash, the predecessor of the first file name of the path name is the current working directory of the process.

Multiple path names may resolve to the same file.

Path Name Resolution

Path name resolution is performed for a process to resolve a path name to a particular file in a file hierarchy.

Each file name in the path name is located in the directory specified by its predecessor.

- For example, in the path name fragment **`a/b`**, file **`b`** is located in directory **`a`**.
- Path name resolution fails if a specifically named file cannot be found in the indicated directory.

Special File Names

Inside a directory, the special file name "`.`" refers to the directory itself.

Inside a directory, the special file name "`..`" refers to the directory's parent directory.

As a special case, in the root directory, "`..`" may refer to the root directory itself.

Path Name Resolution: Symbolic Links

If a symbolic link is encountered during path name resolution, the contents of the symbolic link replace the name of the link.

```
/usr/tmp -> ../var/tmp
```

Evaluates to: `/usr/../var/tmp`

Which evaluates to: `/var/tmp`

Operations on symbolic links behave like operations on regular files unless all of the following are true:

- the link is the last component of the path name
- the path name has no trailing slash
- the function is required to act on the symbolic link itself

Path Names in URLs

A URL may contain a host and a path name:

`http://host.name/path/name/file`

Many web servers use the operating system to resolve the path name.

- "`.`" and "`..`" can be embedded in a URL
- relative paths work
- hard links and symbolic links work

Directory Traversal Vulnerability

A directory traversal vulnerability arises when a program operates on a path name, usually supplied by the user, without sufficient validation.

For example, a program might require all operated-on files to live only in `/home`, but validating that a path name resolves to a file within `/home` is trickier than it looks.

../pathname

Accepting input in the form of `../` without appropriate validation can allow an attacker to traverse the file system to access an arbitrary file.

For example, the following path:

```
/home/../etc/passwd
```

resolves to:

```
/etc/passwd
```

Note that `..` is ignored if the current working directory is the root directory.

Example Directory Traversal Vulnerability

VU#210409 describes a directory traversal vulnerability in FTP clients.

- An attacker can trick users of affected FTP clients into creating or overwriting files on the client's file system.
- To exploit these vulnerabilities, an attacker must convince the FTP client user to access a specific FTP server containing files with crafted file names.
- When an affected FTP client attempts to download one of these files, the crafted file name causes the client to write the downloaded files to the location specified by the file name, not by the victim user.

VU#210409: Demonstration Session

```
CLIENT> CONNECT server
220 FTP4ALL FTP server ready. Time is Tue Oct 01, 2002 20:59.
Name (server:username): test
331 Password required for test.
Password:
230-Welcome, test - Last logged in Tue Oct 01, 2002 20:15 !

CLIENT> pwd
257 "/" is current directory.

CLIENT> ls -l
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 1
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ...\FAKEME5.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ../../FAKEME2.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ../FAKEME1.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ..\..\FAKEME4.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 ..\FAKEME3.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11
/tmp/ftptest/FAKEME6.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 C:\temp\FAKEME7.txt
-rw-r----- 0 nobody nogroup 54 Oct 01 20:10 FAKEFILE.txt
-rw-r----- 0 nobody nogroup 0 Oct 01 20:11 misc.txt
226 Directory listing completed.
```


VU#210409: Demonstration Session 2

```
CLIENT> GET *.txt

Opening ASCII data connection for FAKEFILE.txt...
Saving as "FAKEFILE.txt"

Opening ASCII data connection for
../../FAKEME2.txt...
Saving as "../../FAKEME2.txt"

Opening ASCII data connection for
/tmp/ftptest/FAKEME6.txt...
Saving as "/tmp/ftptest/FAKEME6.txt"
…
```

If a client is vulnerable, it saves files outside of the user’s current working directory.

VU#210409: Vulnerable Products

Product	../	..\	C:	/path	…
wget 1.8.1	💣	☺	☺	💣 3	☺
wget 1.7.1	💣	☺	☺	☺ 2	☺
OpenBSD 3.0 FTP	💣	☺ 1	☺ 1	💣	☺
Solaris 2.6, 2.7 FTP	💣	☺	☺	💣	☺

1. installed the file in the current directory
2. created subdirectories within the current directory
3. only with the `-nH` option ("Disable host-prefixed directories")

Inadequate File Name Validation

Many privileged applications construct path names dynamically incorporating user supplied data.

For example, the following privileged program can be used to parse files in a specific directory:

```
const char *safepath = "/usr/lib/safefile/";
size_t spl = strlen(safe_path);
if (!strncmp(fn, safe_path, spl) {
  process_libfile(fn);
}
else abort();
```

Relative Path Names

If this program takes the file name argument `fn` from an untrusted source (e.g., a user), an attacker can bypass these checks by supplying a file name such as

`/usr/lib/safefiles/../../../etc/shadow`

Data Sanitization

A sanitizing mechanism can remove characters such as `.` and `../` that may be required for some exploits.

An attacker can try to fool the sanitizing mechanism into cleaning data into a dangerous form.

Suppose the attacker injects a `.` inside a file name (e.g., `sensi.tiveFile`) and the sanitizing mechanism removes the character, resulting in the valid file name, `sensitiveFile`.

If the input data are now assumed to be safe, then the file may be compromised.

Poor Data Sanitization

Examples of poor data sanitation techniques for eliminating directory traversal vulnerabilities:

Strip out `../`

- `path = replace(path, "../", "");`
- Input of the form "`....//`" results in "`../`"

Strip out `../ and ./`

- ```
  path = replace(path, "../", "");
  path = replace(path, "./", "");
  ```
- Input of the form `.../....///` results in `../`

Use canonicalization to properly sanitize an untrusted file name.

Agenda

Directory Traversal

Equivalence Errors

Symbolic Links

Canonicalization

Hard Links

Special Files

Sandboxes

Summary

  ```

## Path Equivalence Vulnerabilities

Path equivalence vulnerabilities occur when an attacker provides a different but equivalent name for a resource to bypass security checks.

This is a type of canonicalization error.



## Equivalence Errors

Trailing characters

Single dot directory: `/./`

Case sensitivity

Forks

## Trailing Characters

A trailing / on a file name could bypass access rules that don’t expect a trailing /, causing a server to provide the file when it normally would not.

## Single Dot Directory: /./

EServ Password-Protected File Access Vulnerability

It is possible to construct a web request that is capable of accessing the contents of a protected directory on the web server.

The following example gives an attacker access to a password protected directory:

`http://host/./admin/`

That URL is functionally equivalent to

`http://host/admin/`

but may circumvent validation.

## Case Sensitivity

The Macintosh Hierarchical File System (HFS+) is case insensitive, so

`/home/PRIVATE == /home/private`

Apache directory access control is case sensitive, as it is designed for UFS (CAN-2001-0766).

- `/home/PRIVATE != /home/private`

This creates a directory traversal vulnerability.

For more info:
http://www.securityfocus.com/bid/2852/

## Apple File System Forks

HFS and HFS+ are the traditional file systems on Apple computers

- In HFS, data and resource forks are used to store information about a file.
- The data fork provides the contents of the file, while the resource fork stores metadata such as file type.

Resource forks are accessed in the file system as

- `sample.txt/..namedfork/rsrc`

Data forks are accessed in the file system as

- `sample.txt/..namedfork/data` same as `sample.txt`

## Vulnerabilities in Resource Forks

Applications may be vulnerable to information disclosure due to resource forks.

For example, CVE-2004-1084 describes a vulnerability for Apache running on an Apple HFS+ file system.

- A remote malicious user may be able to directly access file data or resource fork contents.
- Attackers can read the source code of PHP, Perl, and other server-side scripting languages.

## Other Equivalence Issues

Leading or trailing white space

Leading or trailing slash(es)

Internal space: `file`(SPACE)`name`

Asterisk wildcard: `pathname*`

Equivalence vulnerabilities can be mitigated by canonicalization.

## Agenda

Directory Traversal

Equivalence Errors

Symbolic Links

Canonicalization

Hard Links

Special Files

Sandboxes

Summary

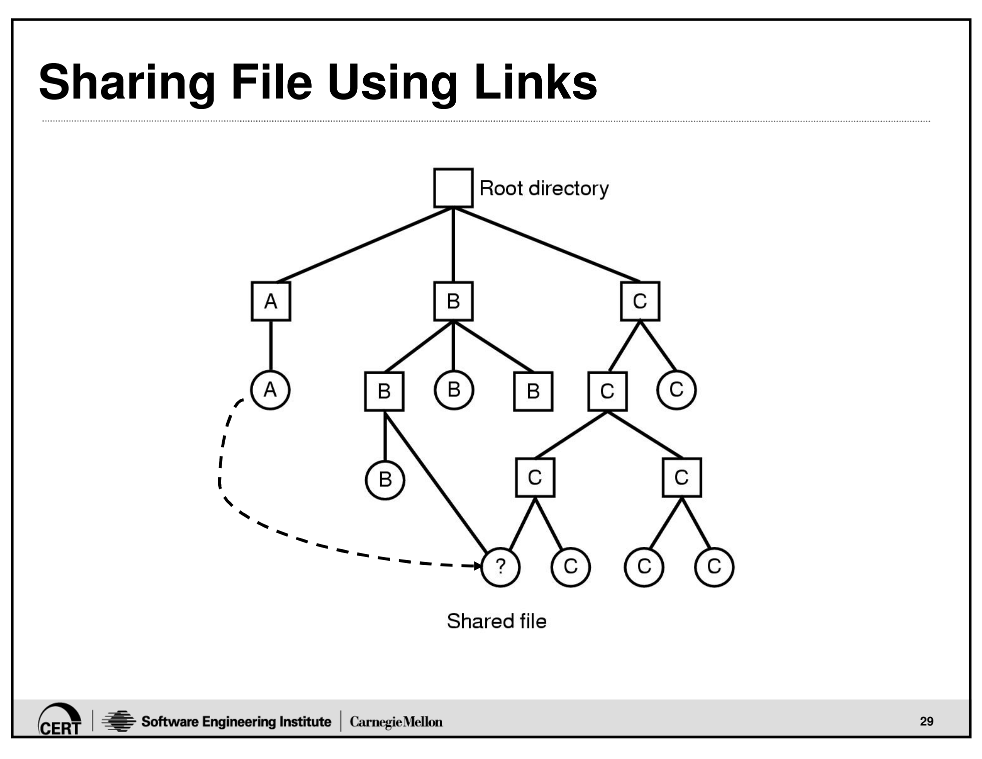
## Symbolic Links

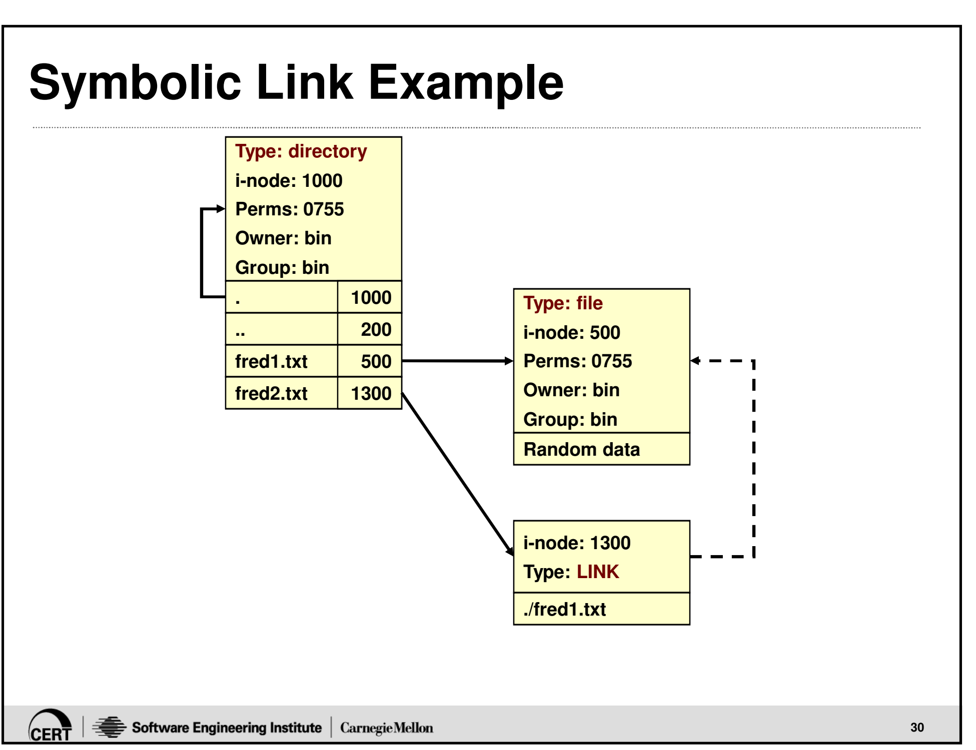
Convenient solution to file sharing.

Often referred to as “symlinks” after the `symlink()` system call.

Creating a symlink creates a new file with a unique i-node.

Symlinks are special files that contain the path name to the actual file.


Sharing File Using Links
Root directory
A
B
C
A
B
B
B
C
C
B
C
C
?
C
C
C
Shared file
CERT
Software Engineering Institute
Carnegie Mellon
29


Symbolic Link Example
Type: directory
i-node: 1000
Perms: 0755
Owner: bin
Group: bin
. 1000
.. 200
fred1.txt 500
fred2.txt 1300
Type: file
i-node: 500
Perms: 0755
Owner: bin
Group: bin
Random data
i-node: 1300
Type: LINK
./fred1.txt
CERT
Software Engineering Institute
Carnegie Mellon
30

## Symbolic Link Vulnerability Example

Assume the following code runs as a setuid root application with effective root privileges:

```
fd = open("/home/usr1/.conf", O_RDWR);

if (fd < 0) abort();

write(fd, userbuf, userlen);
```

Absolute path not supplied by user

Assume also that the attacker can control the data (`userbuf`) that is written in the call to `write()`.

Simple Exploit

An attacker creates a symbolic link from `.conf` to the `/etc/passwd` authentication file

```
% cd  /home/usr1
% ln -s /etc/passwd .conf
```

and then runs the vulnerable program, which

- opens the file for writing as root
- writes attacker controlled information to the password file

```
% runprog
```

This attack can be used, for example, to create a new root account with no password.

The attacker then uses the `su` command to switch to the root account for root access:

```
% su
#
```

Symlink Aware Functions

The following functions operate on the symbolic link file itself, and not on the file it references:

`unlink()` deletes the symbolic link file

`lstat()` returns information about the symbolic link file

`lchown()` changes the user and group of the symbolic link file

`readlink()` reads the contents of the specified symbolic link file

`rename()` renames a symlink specified as the *from* argument or overwrites a symlink file specified as the *to* argument

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.

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.

For example, you can change the version of an application in use, or even an entire website, by changing a symlink.

Conditions of Vulnerability

Symlink attacks are not a concern within a secure directory.

You are at risk if you operate

- in a shared directory such as `/tmp`
- in someone else's directory with elevated privileges (running an antivirus program as administrator, for example)

Use canonicalization to avoid symbolic link vulnerabilities.

Agenda

Directory Traversal

Equivalence Errors

Symbolic Links

Canonicalization

Hard Links

Special Files

Sandboxes

Summary

Canonicalization

Path names, directory names, and file names may contain characters that make validation difficult and inaccurate.

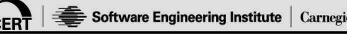
Furthermore, any path name component can be a symbolic link, which further obscures the actual location or identity of a file.

To simplify file name validation, it is recommended that names be translated into their *canonical* form.

Canonicalizing file names makes it much easier to verify a path, directory, or file name by making it easier to compare names.

Because the canonical form can vary between operating systems and file systems, it is best to use operating-system-specific mechanisms for canonicalization.

[FIO02-C. Canonicalize path names originating from untrusted sources]



Canonical Form

Canonical form is the standard form or representation for something.

Canonicalization is the process by which various equivalent forms of a name can be resolved to a single, standard name.

Canonicalization provides a solution for

- directory traversal
- equivalence errors
- symlink issues

The canonical form should not include symlinks.

`/usr/../home/rcs` is equivalent to `/home/rcs`

`/home/rcs` is the canonical path

Canonicalization Question

Given that there is a symbolic link:

`/home/alfred/sss -> /home/myhomebiz/accounting/spreadsheets/`

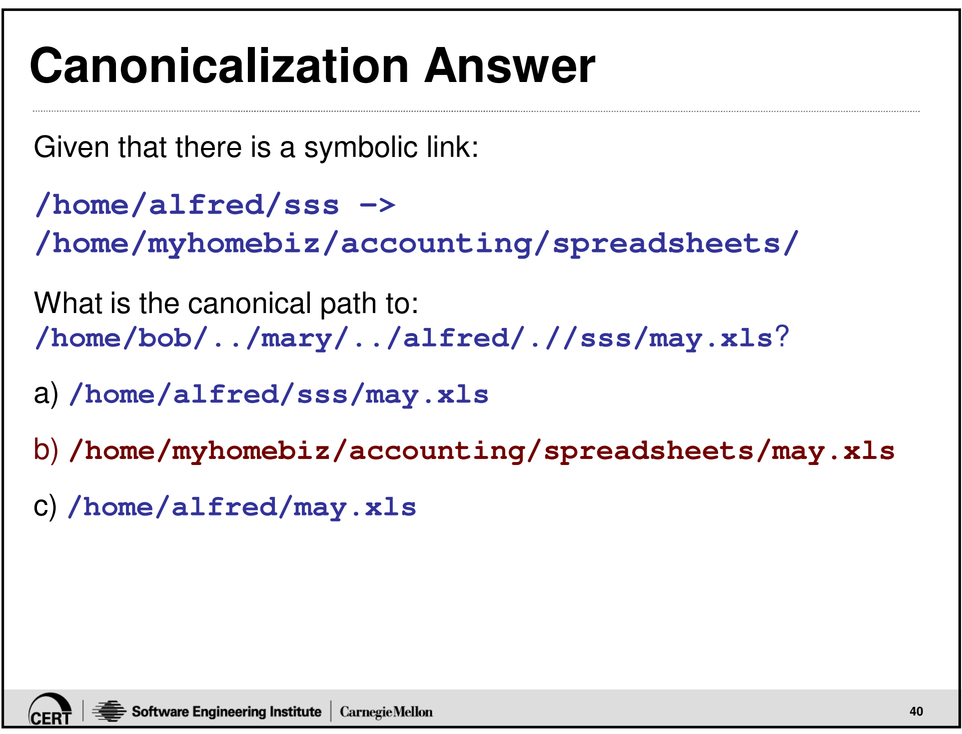
What is the canonical path to:
`/home/bob/../mary/../alfred/.//sss/may.xls`?

a) `/home/alfred/sss/may.xls`

b) `/home/myhomebiz/accounting/spreadsheets/may.xls`

c) `/home/alfred/may.xls`

CERT | Software Engineering Institute | Carnegie Mellon 39



UNIX Canonicalization

The POSIX `realpath()` function returns the canonical path name for a file.

The GNU libc4, libc5, and BSD implementations contain a buffer overflow [VU#743092].

- fixed in libc-5.4.13 (the vulnerability was latent for over a decade)

Consequently, programs must use versions of this function where this issue is known to be resolved.

The `realpath()` Function

The `realpath()` function is specified as

```
char *realpath(
  /* file path to resolve */
  const char *restrict file_name,
  /* location to save canonical path */
  char *restrict resolved_name
);
```

If `resolved_name` is a null pointer, the behavior of `realpath()` is implementation-defined.

Revised `realpath()` Implementation

The `realpath()` function has changed as of POSIX.1-2008

This revision, and many current implementations (led by glibc and Linux), allocate memory to hold the resolved name if a null pointer is passed.

- memory is allocated as if by `malloc()`
- the application should relesea such memory when no longer required by a call to `free()`

The following statement can be used to conditionally include code that depends on this form of the `realpath()` function

```
#if _POSIX_VERSION >= 200809L || defined (linux)
```


Revised `realpath()` Example

```
char *realpath_res = NULL;
/* Verify argv[1] is supplied */
realpath_res = realpath(argv[1], NULL);
/* Verify file name */
fopen(realpath_res, "w");
/* ... */
free(realpath_res);
realpath_res = NULL;
```

PATH_MAX Version

Older versions of the `realpath()` function expect `resolved_name` to refer to a character array large enough to hold the canonicalized path.

A buffer of at least size `PATH_MAX` is adequate, but `PATH_MAX` is not guaranteed to be defined.

If `PATH_MAX` is defined, allocate a buffer of size `PATH_MAX` to hold the result of `realpath()`.

PATH_MAX Version Example

```
char *realpath_res = NULL;
char *canonical_file = NULL;
canonical_file = malloc(PATH_MAX);
realpath_res = realpath(
  argv[1], canonical_file
);
/* Verify file name */
fopen(realpath_res, "w");
/* ... */
free(canonicalized_file);
```

canonicalize_file_name()

GCC users can also use the GNU extension:

```
char *canonicalize_file_name(const char *)
```

- returns the canonical name containing no `.`, `..` components, repeated path separators (`/`), or symlinks.
- result is passed back as the return value of the function in a block of allocated memory
- when done, memory should be freed by calling `free()`

Canonicalization in Windows

Canonicalization issues are more complex in Windows, due to the many ways of naming a file:

- universal naming convention (UNC) shares
- drive mappings
- short (8.3) name
- long name
- Unicode name
- special files
- trailing dots, forward slashes, and backslashes
- shortcuts
- etc.

Avoid Decisions Based on Names

Avoid making decisions based on a path, directory, or file name; there is a very loose correlation between file names and files.

- Don't trust the properties of a resource because of its name.
- Don't use the name of a resource for access control.

There is often more than one valid way to represent the name of the object.

Instead of file names, use operating system-based mechanisms.

- access control lists (ACLs)
- other authorization techniques
- file type
- other metadata (number of hard links, etc.)

This is particularly true of Windows operating systems, where canonicalization is a nightmare [Howard 02].

Agenda

Directory Traversal

Equivalence Errors

Symbolic Links

Canonicalization

Hard Links

Special Files

Sandboxes

Summary

Hard Links

Hard links can be created using the `ln` command. For example, the command `ln /etc/passwd`

- increments the link counter in the i-node for the `passwd` file
- creates a new directory entry in the current directory

Hard links are indistinguishable from original directory entry.

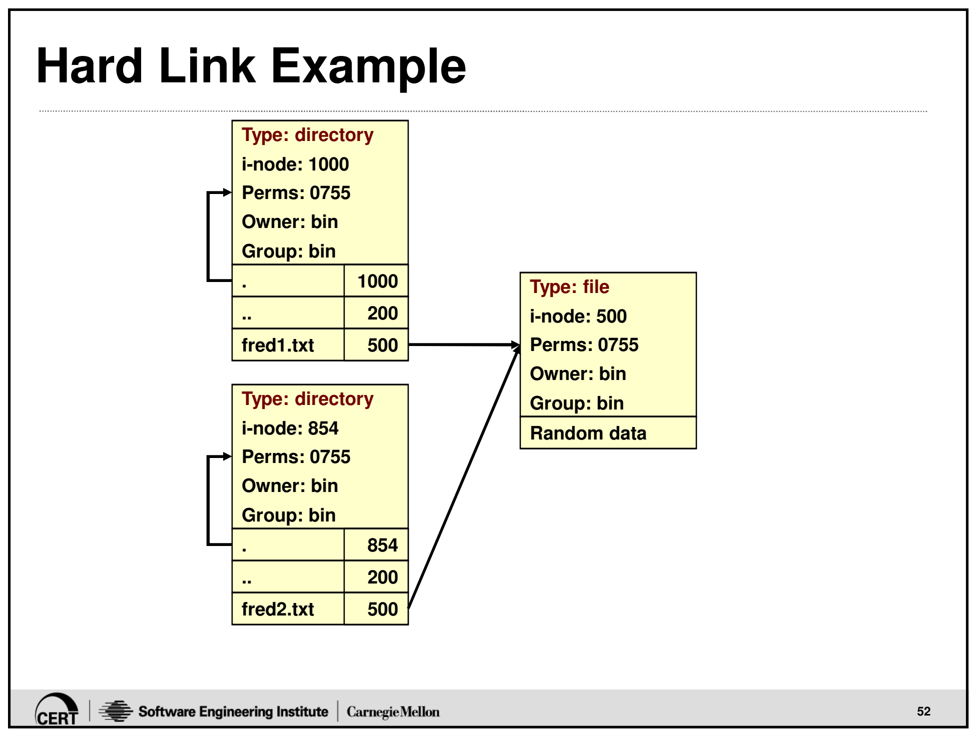
Hard links cannot refer to directories or span file systems.

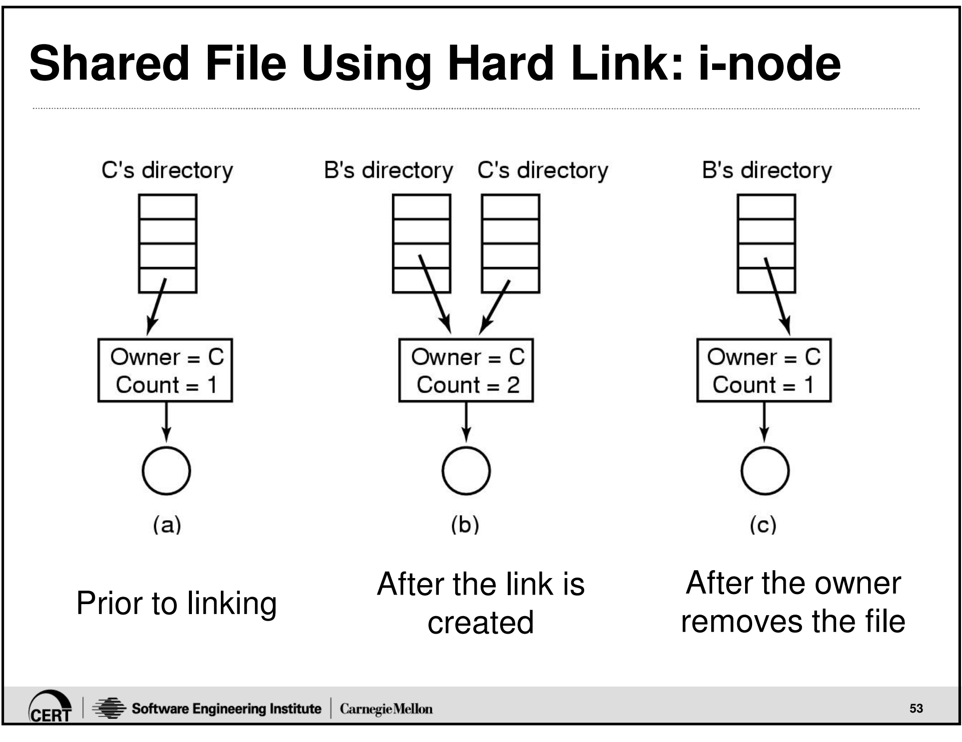
Ownership and permission reside with the i-node, so all hard links to the same i-node have the same ownership and permissions.

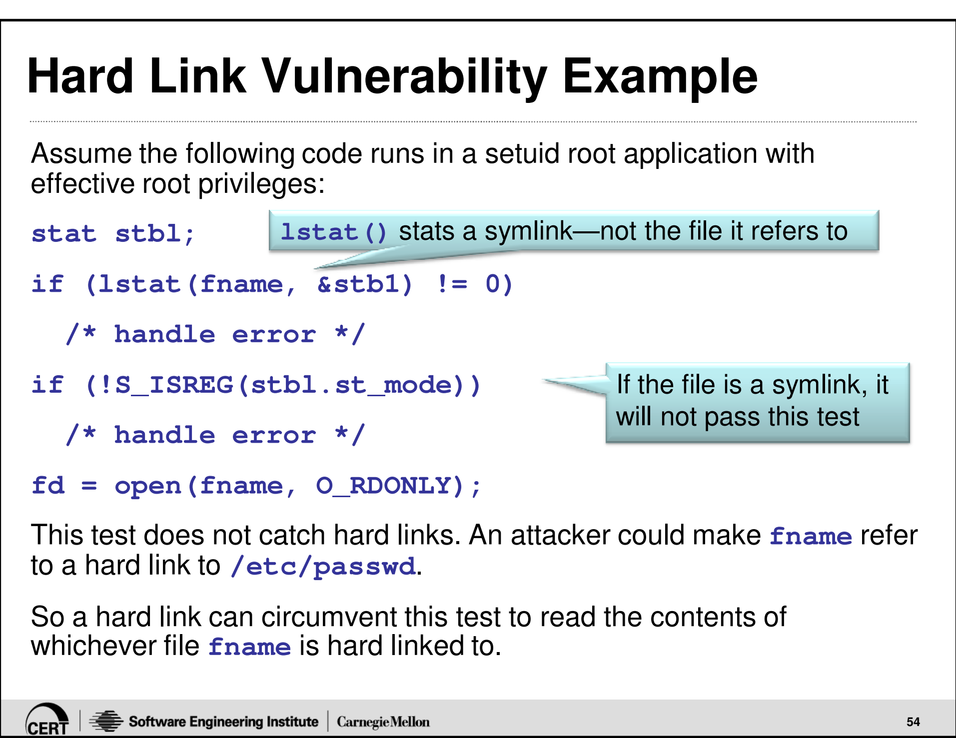
Deleting a hard link doesn't delete the file unless all references to the file have been deleted.

- a reference is either a hard link or an open file descriptor
- i-node can only be deleted (data addresses cleared) if link counter is 0
- original owner cannot free disk quota unless all hard links are deleted

Hard Link Example




Shared File Using Hard Link: i-node
C's directory
B's directory
C's directory
B's directory
Owner = C
Count = 1
Owner = C
Count = 2
Owner = C
Count = 1
(a)
(b)
(c)
Prior to linking
After the link is created
After the owner removes the file
Software Engineering Institute
CarnegieMellon
53


Hard Link Vulnerability Example
Assume the following code runs in a setuid root application with effective root privileges:
stat stbl;
lstat() stats a symlink—not the file it refers to
if (lstat(fname, &stbl) != 0)
/* handle error */
if (!S_ISREG(stbl.st_mode))
If the file is a symlink, it will not pass this test
/* handle error */
fd = open(fname, O_RDONLY);
This test does not catch hard links. An attacker could make fname refer to a hard link to /etc/passwd.
So a hard link can circumvent this test to read the contents of whichever file fname is hard linked to.
Software Engineering Institute
CarnegieMellon
54

There Can Be Only One

Check the link count to determine if there is more than one path to the file.

```
stat stbl;
if ( (lstat(fname, &stbl) == 0) && // file exists
     (!S_ISREG(stbl.st_mode)) && // regular file
     (stbl.st_nlink <= 1) ) { // no hard links
  fd = open(fname, O_RDONLY);
}
else {
  /* handle error */
}
```

> This code has another vulnerability, however, which we will look at in the next module.

Hard Link Mitigations

Create separate partitions for sensitive files and user files (hard links cannot span file systems)

- prevents hard-link exploits (such as linking to `/etc/passwd`)
- good advice for system administrators
- developers cannot assume that systems are configured in this manner

Hard Link vs. Symbolic Link

Hard Link	Symbolic Link
Shares an i-node with the file linked to	Is its own file (that is, has its own i-node)
Same owner and privileges as linked-to file	Has owner and privileges independent of linked-to file
Always links to an existing file	Can reference a non-existent file
Doesn't work across file systems or on directories	Works across file systems and on directories
Cannot distinguish between original and recent links to an i-node	Can easily distinguish symbolic links from other types of files

Checking for the Existence of Links 1

Case 1: A program with elevated privileges that wants to create a file that doesn't already exist.

Make sure the file doesn't exist, for example, by calling `open()` with the `O_CREAT | O_EXCL` flags.

```
int fd = open(
  file_name,
  O_CREAT | O_EXCL |O_WRONLY,
  new_file_mode
);
```

Checking for the Existence of Links 2

Case 2: A setuid program that wants to prevent users from overwriting protected files.

(Temporarily) drop privileges and perform the I/O with the real user ID.

If the user passes you a symbolic link or a hard link, who cares? as long as the user has permissions to modify the file.

Creating a hard link or a symbolic link will not alter the permissions on the i-node for the actual file.

Agenda

Directory Traversal

Equivalence Errors

Symbolic Links

Canonicalization

Hard Links

Special Files

Sandboxes

Summary

UNIX Special Files 1

Directory

- marked with a `d` as the first letter of the permissions field
  `drwxr-xr-x /`

Symbolic link is a reference to another file.

- stored as a textual representation of the file's path
- marked with an `l` in the permissions string:

```
lrwxrwxrwx termcap -> /usr/share/misc/termcap
```

Named pipes enable different processes to communicate and can exist anywhere in the file system.

- made with the command `mkfifo` as in `mkfifo mypipe`.
- marked with a `p` as the first letter of the permissions string:

```
prw-rw---- mypipe
```

UNIX Special Files 2

Socket

- allows communication between two processes running on the same machine
- marked with an `s` as the first letter of the permissions string

```
srwxrwxrwx X0
```

Device file

- used to apply access rights and to direct operations on the files to the appropriate device drivers
- distinction between character devices and block devices:
  - Character devices provide only a serial stream of input or output (marked with a `c` as the first letter of the permissions string).
  - Block devices are randomly accessible (marked with a `b`).

```
crw------- /dev/kbd
brw-rw---- /dev/hda
```

Linux Device Names

On Linux, it is possible to lock certain applications by attempting to open devices rather than files, such as

- `/dev/mouse`
- `/dev/console`
- `/dev/tty0`
- `/dev/zero`

A web browser, for example, that failed to check for these devices would allow an attacker to create a website with image tags such as

```
<img src="file:///dev/mouse"/>
```

that would lock the user’s mouse.

POSIX: Regular File?

The `stat()` function can be used in conjunction with the `S_ISREG()` macro to identify regular files.

```
struct stat s;
if (stat(filename, &s) == 0) {
  if (S_ISREG(s.st_mode)) {
    /* file is a regular file */
  }
}
```

Agenda

Directory Traversal

Equivalence Errors

Symbolic Links

Canonicalization

Hard Links

Special Files

Sandboxes

Summary

Sandboxes

These tools serve to isolate one or more programs into a safe subset of the file system.

- `chroot()`
- `jail()`

chroot() 1

Calling **chroot()** effectively establishes an isolated file directory with its own directory tree and root.

- confines a user process to a portion of the file system
- prevents unauthorized access to system resources

The new tree guards against "..", symlink, and other exploits applied to containing directories.

Calling **chroot()** requires superuser privileges, while the code executing within the jail cannot execute as root.

chroot() 2

In UNIX, **chroot()** changes root directory

- Originally used to test system code safely
- Confines code to limited portion of file system
- Sample use:

```
chdir /tmp/ghostview
chroot /tmp/ghostview
su tmpuser     (or su nobody)
```

Potential problems

- **chroot()** changes root directory, but not current dir
  If forget **chdir()**, program can escape from changed root
- If you forget to change UID, process could escape

`jail()`

First appeared in FreeBSD

In addition to file system restrictions imposed by `chroot()`

- each jail is bound to a single IP address, so processes within the jail cannot use other IP addresses for sending or receiving network communications
- only interact with other processes in the same jail

Still too coarse

- directory to which program is confined may not contain all utilities the program needs to call
- if copy utilities over, may provide dangerous weapons
- no control over network communications rCs9

Extra Programs Needed in Jail

Files needed for `/bin/sh`

- `/usr/ld.so.1` shared object libraries
- `/dev/zero` clear memory used by shared objs
- `/usr/lib/libc.so.1` general C library
- `/usr/lib/libdl.so.1` dynamic linking access library
- `/ usr/lib/libw.so.1` I18n library
- `/usr/lib/libintl.so.1` I18n library

Slide 69

---

rCs9 this statement appears inconsistent with the fact that you can restrict to single IP address
Robert C. Seacord, 8/28/2008

Agenda

Directory Traversal

Equivalence Errors

Symbolic Links

Canonicalization

Hard Links

Special Files

Sandboxes

Summary

Vulnerability and Mitigation Summary

Vulnerability	Mitigation
Directory Traversal	Canonicalization
Equivalence Errors	Canonicalization
Symbolic Links	Canonicalization
Hard Links	Check link count Separate partition
Special Files	Check file type

Summary

Avoid exposing your file system directory structure or file names through your user interface or other external APIs.

There is a very loose correlation between file names and files: Avoid making decisions based on a path, directory, or file name.

Use operating-system-specific canonicalization methods.

Don't make assumptions about the file system.

Secure Coding Guidelines

FIO02-C. Canonicalize path names originating from untrusted sources

References

[Meunier 04] Pascal Meunier. CS390S: Canonicalization and Directory Traversal, November 2004.

[MITRE 07] MITRE. Common Weakness Enumeration, Draft 7. October 2007. http://cwe.mitre.org

[Howard 02] Howard, Michael & LeBlanc, David C. *Writing Secure Code,* 2nd ed. Redmond, WA: Microsoft Press, 2002 (ISBN 0-7356-1722-8).

[Viega 03] Viega, John & Messier, Matt. *Secure Programming Cookbook for C and C++: Recipes for Cryptography, Authentication, Networking, Input Validation & More*. Sebastopol, CA: O'Reilly, 2003 (ISBN 0-596-00394-3).