

Symbolic Links Considered Harmful

SAMBA

Jeremy Allison

**Samba Team/Google Open Source
Programs Office**

**jra@samba.org
jra@google.com**

What are symbolic links ?

- Strange to have to explain this in a file system conference, but..
- Most computer users these days have no contact with a file system.
 - iPhone and Android users have no concept of a file system on their device. Each application only handles its own kind of data storage.
 - Possibly to enforce data “silos” to keep users tied to an application.
 - Students no longer know where a file is stored:
<https://www.theverge.com/22684730/students-file-folder-directory-structure-education-gen-z>
 - Users only search for “objects” by name.
- I have to help my family move “objects” around from phone to file server.

Step back – What is a file system ?

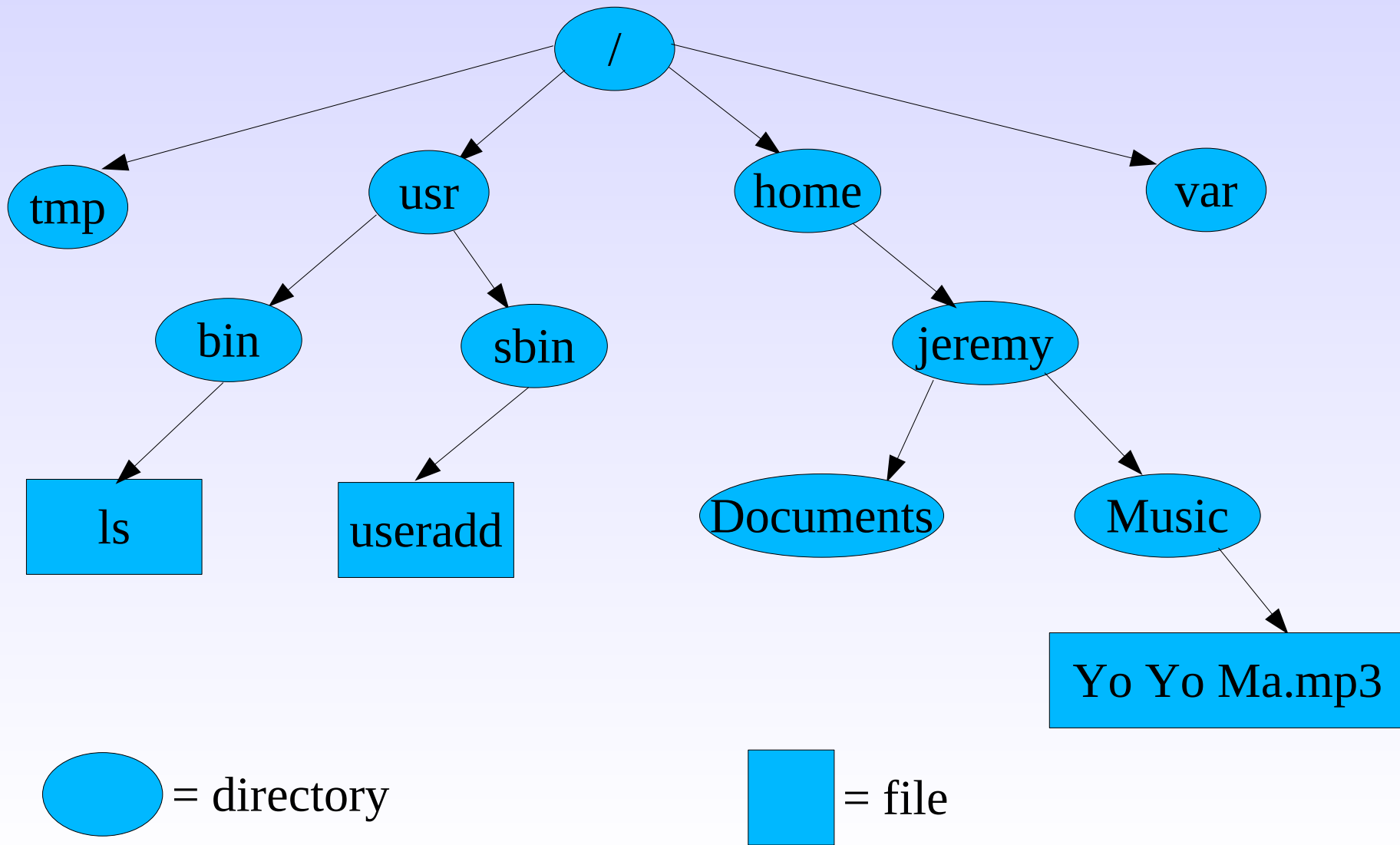


SAMBA

Opening Windows to a wider world

A file system visualized.

Path = /home/jeremy/Music/Yo Yo Ma.mp3



The original UNIX file system

C API dealing with paths

- `open("/home/jeremy/Music/Yo Yo Ma.mp3", int flags, mode_t mode)`
- `unlink("/path/to/file")`
- `mkdir("/new/directory/name")`
- `rmdir("/directory/name")`
- `stat("Yo Yo Ma.mp3", struct stat *st)` (`$cwd` is `"/home/jeremy/Music"`)
- `chmod("/path/to/file", int mode)`
- `chown("/path/to/file", uid_t owner, gid_t group)`
- `chdir("/path/to/new/working/directory")`
- etc..
- Note the "path" may be specified from the root (starts with '/') or relative to the current working directory (doesn't start with '/').

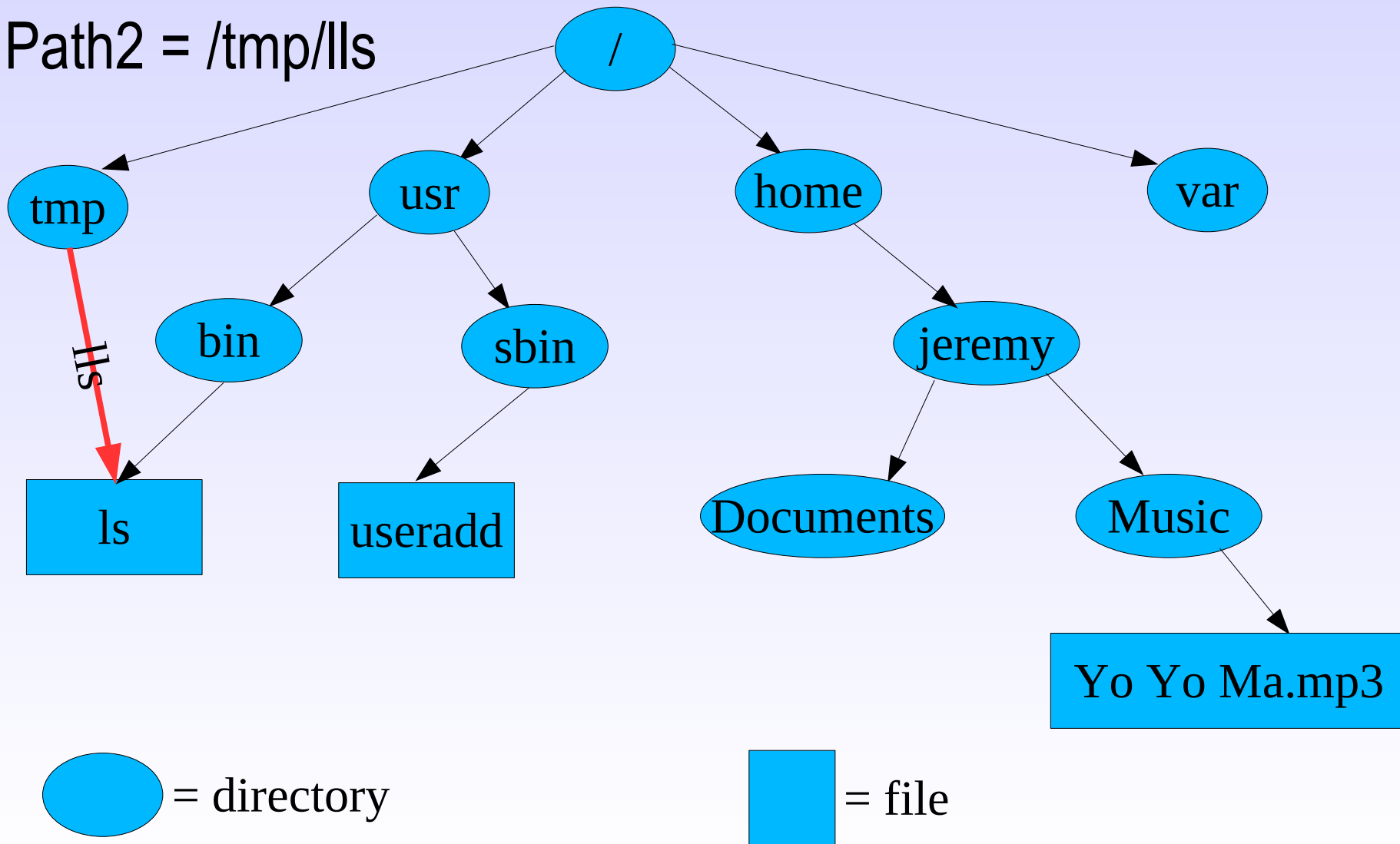
Hard links:

In `/usr/bin/ls` `/tmp/lls`

`link("/usr/bin/ls", "/tmp/lls")`

Path1 = `/usr/bin/ls`

Path2 = `/tmp/lls`



Hard link details

- Hard links create a new directory entry (name) which points to the *same* file data and metadata.
- Hard links cannot be made to directories.
- Hard links simply create a new absolute path to the *same* file.
- Useful to allow a single file to be referenced by many names.
 - Underlying data is only removed once last link has gone.
- First use of hard links seems to be in the Incompatible Timesharing System (ITS) in 1969.

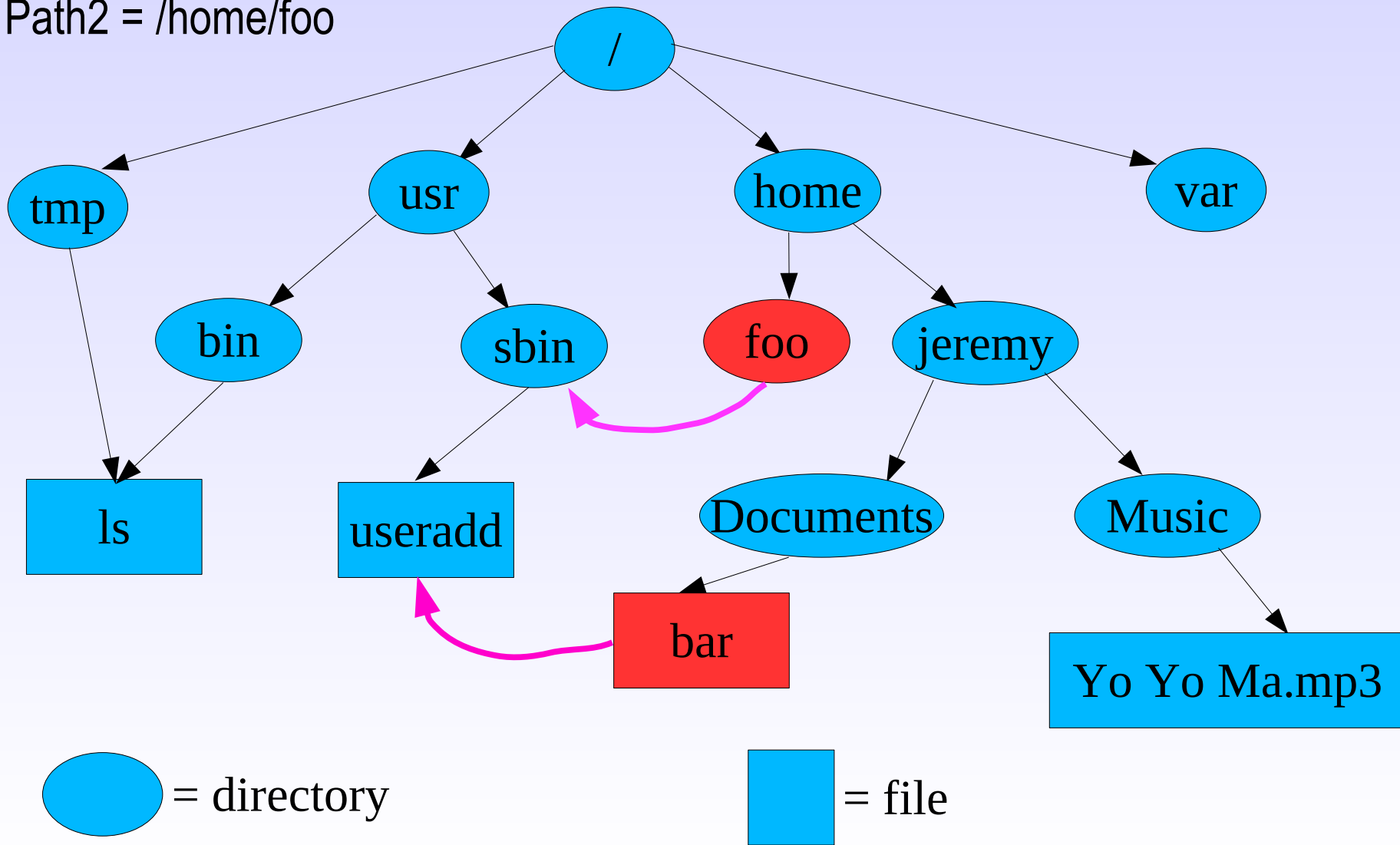
Symbolic links:

In -s /usr/sbin /home/foo

`symlink("/home/foo", "/usr/sbin")`

Path1 = /usr/sbin

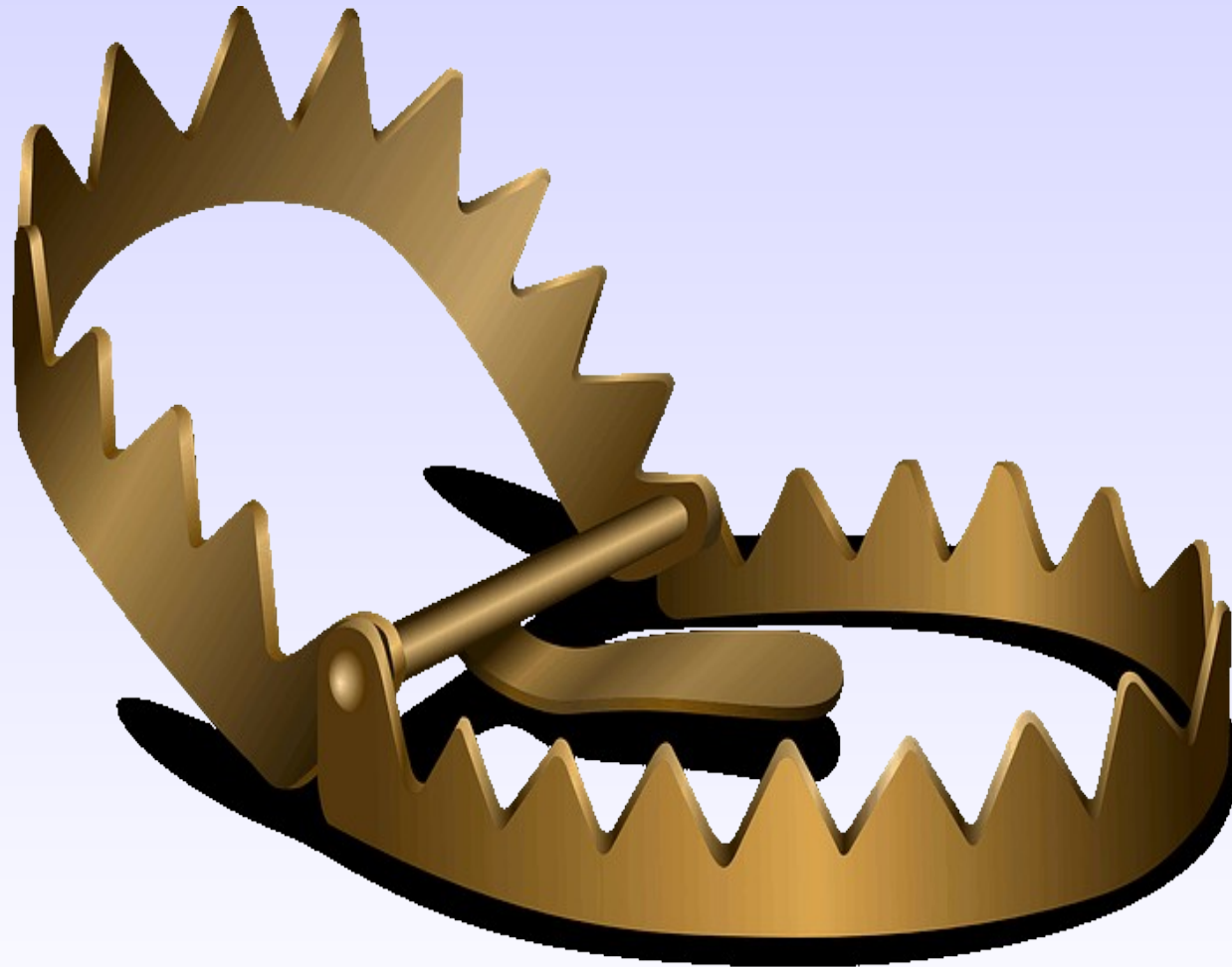
Path2 = /home/foo



Symbolic link details

- Symbolic links allow the creation of a new object in the file system that causes any process accessing it to follow it to an arbitrary target somewhere else on the file system.
 - Not only files, but directories too.
 - Loops can be created.
 - This should have been a warning sign to file system designers that they were doing something wrong.
- First reference to them is from MULTICS in 1965.
 - But added to 4.2 BSD Unix.
 - “..symbolic links have been added to release 4.2 of Berkeley Unix. This feature frees the user from the constraints of the strict hierarchy that a tree structure imposes. This flexibility is essential for good name space management.”

Symbolic links for application developers



Why are symlinks so bad ?

- Symlinks allow paths to change on the fly, creating a whole class of time-of-check, time-of-use (TOCTOU) race condition security problems.
- Symlinks are not restricted to privileged users, but can be created by anyone with write access anywhere in a path.
- Symlinks break the beautiful “tree” abstraction of a POSIX file system.
- Symlinks break the beautiful simplicity of the POSIX file system API.

The evolution of the API to deal with symlinks

- First change was introduction of lstat
 - lstat(const char *path, struct stat *st)
 - Original “stat()” API silently and transparently follows symlinks.
 - Allows calling application to detect if the terminal component of a path is a symlink.
 - Does not detect symlinks other than the terminal component.
- Ended up in unsafe code such as:

```
lstat(dangerous_path, &sbuf);  
if (!S_ISLNK(sbuf.st_mode)) {  
    do_dangerous_operation_on(dangerous_path);  
}
```

Go Speed Racer Go !

- Race condition exists between:

```
lstat(dangerous_path, &sbuf);  
-----race starts here-----  
if (!S_ISLNK(sbuf.st_mode)) {  
    -----race ends here-----  
    do_dangerous_operation_on(dangerous_path);  
}
```
- If an attacker can rename `dangerous_path`, and replace it with a symlink to somewhere else before `do_dangerous_operation_on(dangerous_path)` is called, the dangerous operation is applied to the attacker's chosen path.

Surely these races are too narrow for me to care about ?

- <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=symbolic+link>
 - There are 1361 CVE Records that match your search.
- This does not just include “old” applications that were written before symlink mitigation API’s were added to POSIX.
 - This includes symlink race condition security holes in the Rust language standard library (from 2022).
- The API’s added to mitigate symlink errors are impossible for application developers to use safely.
 - Similar to the care needed for “atomic” files data and meta-data updates, the POSIX API changes are too complex for safe use.

POSIX Symlink API mitigations

- First was an additional flag, `O_NOFOLLOW` to the `open()` system call.
 - Looks perfect, in practice doesn't do what applications need.
 - `O_NOFOLLOW` prevents the terminal component in a pathname passed to `open()` being a symlink.
 - If completely ignores symlinks in non-terminal components.
- Example of an exploit:
 - (Application running as root – checks `/data/mydir` is safe)
Attacker renames `"/data/mydir"` → `"/data/out-of-the-way"`
`symlink("/data/mydir", "/etc");`
`int fd = open("/data/mydir/passwd", O_WRONLY|O_NOFOLLOW..);`
`ret = write(fd, data, size);`
 - Application now writes into `/etc/passwd`

More POSIX API mitigations

- To solve the previous `O_NOFOLLOW` problem applications have to `chdir()` into the parent directory. Check it hasn't been symlink raced and then use `O_NOFOLLOW`, then `chdir` back.
 - Samba does this.
- `open()` → `openat(int dirfd, const char *path, int flags, mode_t mode)`
 - This actually works. The 'dirfd' parameter here is a handle of a containing (parent) directory.
 - So long as "path" has no "/" characters and flags contains `O_NOFOLLOW`, then this cannot be raced.
- Of course, getting the handle on the parent directory also has to be protected against symlink races.

XXXXat's for everyone !

- Based on `openat()`, **ALL** path-based operations must have an `XXXXat()` variant to avoid symlink races in the same way.
- Oh look, lots of new system calls.

`openat()`, `mkdirat()`, `unlinkat()`, `linkat()`, `renameat()`,
`symlinkat()`, `fstatat()`, `fchmodat()`, `fchownat()`, `futimesat()`,
`mknodat()`, `faccessat()`, `readlinkat()`, `utimensat()`, `scandirat()`,
`execveat()`

- The original clean and simple POSIX filesystem API doesn't look so clean and simple anymore.
 - And on Linux, one of these calls doesn't work – `fchmodat()` will still always follow symlinks in the target path.

Pathnames are now broken.

- Any application that allows more than one component in a path without splitting the last component off and using the XXXXat() functions can be symlink raced.
- Still not enough for a feature complete application.
 - Extended attribute calls are missing, no getxattrat(), setxattrat() etc.
 - For extended attribute pathname operations the chdir()/realpath()/getxattr()/chdir() dance must still be done.
- I know, let's add more open() flags !
 - Linux added O_PATH.
 - Allows a handle to be taken on a file or directory, usually meant to be passed as the file descriptor argument to the XXXXat() functions.
 - O_PATH handles cannot be used to read/write data.

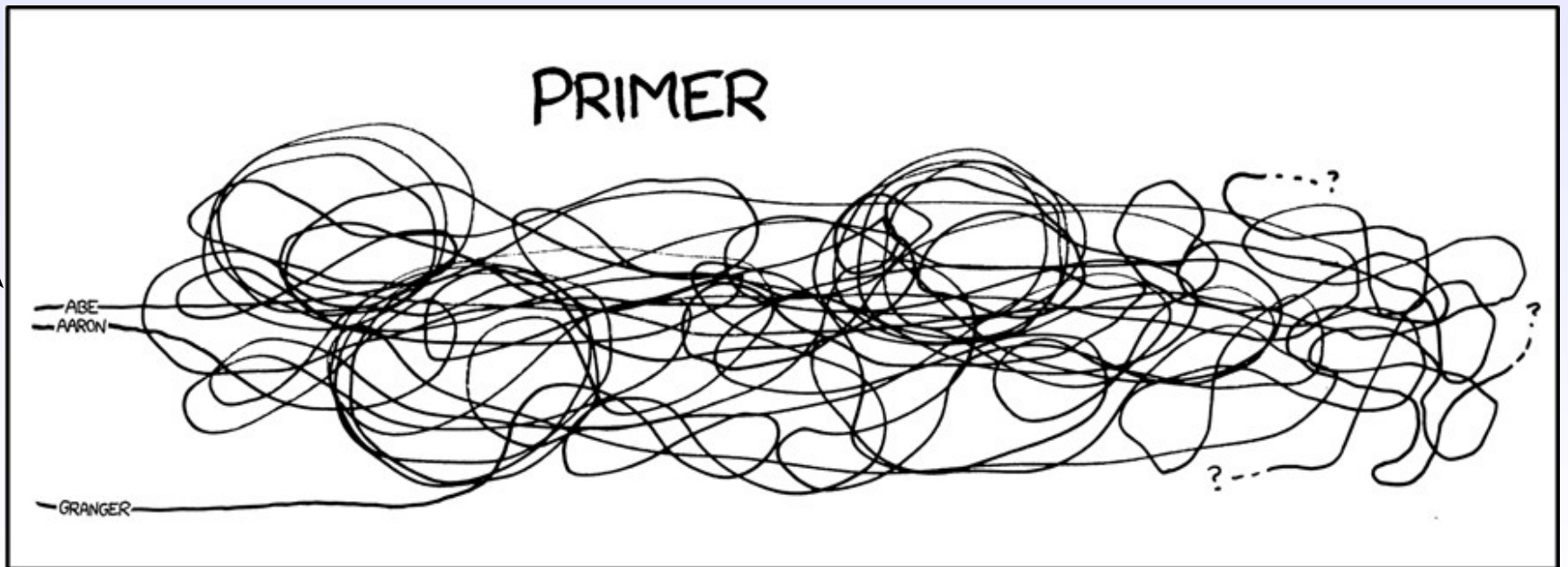
Extended attributes revisited

- Having an O_PATH handle would be a great solution for getting/setting extended attributes where you don't want to open the file for modification.
 - Unfortunately O_PATH prohibits reading or writing extended attributes.
- “Hack” solution, invented by a Red Hat engineer.
 - ```
int fd = openat(dirfd, "file", O_PATH|O_NOFOLLOW);
sprintf(buf, "/proc/self/fd/%d", fd);
getxattr(buf, ea_name, value, size);
```
- Depends on Linux-only semantics of /proc file system.
  - Insanity, pure insanity.

# Symlinks turn this:



Into this  
(“Primer” timeline credit xkcd):



# I assert that pathnames are now unusable for “mortal” application developers on POSIX

- I claim that for a non-trivial application, it is impossible for application developers to avoid symlink races.
- It's not just their own code – all library code they call that uses path names must be aware of multi-path-component symlink races.
- Spoiler alert – even security library code is not symlink race aware.

# Example #1

- Given a directory hierarchy:

```
foo/
 bar/
 baz/
 bibble
```

- `$ strace setfacl -R -m u:gdm:r foo`

```
setxattr("foo", "system.posix_acl_access", "...", 44, 0) = 0
```

```
getxattr("foo/bar", "system.posix_acl_access", 0x7ffc474a4c00, 132) = -1 ENODATA
```

```
setxattr("foo/bar", "system.posix_acl_access", "...", 44, 0) = 0
```

```
getxattr("foo/bar/baz", "system.posix_acl_access", 0x7ffc474a4b70, 132) = -1 ENODATA
```

```
setxattr("foo/bar/baz", "system.posix_acl_access", "...", 44, 0) = 0
```

```
getxattr("foo/bar/baz/bibble", "system.posix_acl_access", 0x7ffc474a4ae0, 132) = -1
ENODATA
```

```
setxattr("foo/bar/baz/bibble", "system.posix_acl_access", "...", 44, 0) = 0
```

## Example #2

- In one of the patches for git CVE-2022-24765

```
+#ifndef is_path_owned_by_current_user
+static inline int is_path_owned_by_current_uid(const char *path)
+{
+ struct stat st;
+ if (lstat(path, &st))
+ return 0;
+ return st.st_uid == geteuid();
+}
+
+#define is_path_owned_by_current_user is_path_owned_by_current_uid
+#endif
```

- Called from `ensure_valid_ownership(const char *path)`, also added for CVE-2022-24765.

# Example #3

- Rust language standard library CVE-2022-21658.

The Rust Security Response WG was notified that the `std::fs::remove_dir_all` standard library function is vulnerable to a race condition enabling symlink following (CWE-363). An attacker could use this security issue to trick a privileged program into deleting files and directories the attacker couldn't otherwise access or delete.

## Overview

Let's suppose an attacker obtained unprivileged access to a system and needed to delete a system directory called `sensitive/`, but they didn't have the permissions to do so. If `std::fs::remove_dir_all` followed symbolic links, they could find a privileged program that removes a directory they have access to (called `temp/`), create a symlink from `temp/foo` to `sensitive/`, and wait for the privileged program to delete `foo/`. The privileged program would follow the symlink from `temp/foo` to `sensitive/` while recursively deleting, resulting in `sensitive/` being deleted.

To prevent such attacks, `std::fs::remove_dir_all` already includes protection to avoid recursively deleting symlinks, as described in [its documentation](#):

This function does not follow symbolic links and it will simply remove the symbolic link itself.

Unfortunately that check was implemented incorrectly in the standard library, resulting in a TOCTOU (Time-of-check Time-of-use) race condition. Instead of telling the system not to follow symlinks, the standard library first checked whether the thing it was about to delete was a symlink, and otherwise it would proceed to recursively delete the directory.

This exposed a race condition: an attacker could create a directory and replace it with a symlink between the check and the actual deletion. While this attack likely won't work the first time it's attempted, in our experimentation we were able to reliably perform it within a couple of seconds.



# How can we fix this mess #1 ?

- Learn from Windows.
  - Yes, Windows implemented this **RIGHT**.
- The Windows NTFS file system has application followed symlinks, called reparse points.
- Symbolic links on NTFS by default can only be created by an Administrator (root).
  - This fixes the problem perfectly. No code is safe from root anyway.
- Unfortunately this will break many existing applications (systemd user services for one).

# How can we fix this mess #2 ?

- New system call (yes ! The Linux way).
- Linux system call `openat2()` has a flags field:
  - `RESOLVE_BENEATH`
  - `RESOLVE_IN_ROOT`
  - `RESOLVE_NO_SYMLINKS`
- All restrict symlink following in different ways (see the man page).
  - No glibc wrapper (yet).
  - Only fixes the problem for `open()`.
  - All applications need to be re-written.
  - Promising for the future though.

## How can we fix this mess #3 ?

- Suggested by lwn user “willy”.
- Add a prctl() (process control) option that causes any system call traversing a symlink to return ELOOP.
- This will break existing applications, but in the “right” way (i.e. they individually have to ask for it, and then cope “correctly”).
- No one is currently planning on implementing this.

# How can we fix this mess #4 ?

- Suggested by lwn user “nix”
- Change symlink semantics such that symlinks owned by non-root are only followed by a process with a token containing the uid that created them.
  - More subtle protection, but would still break existing applications.
  - Probably too confusing for administrators, symlinks “randomly” breaking.
  - Still doesn’t fix the “restricted share” problem when exporting a file system (may be a Samba / NFS specific problem).
- Again, no one is currently planning on implementing this.

# How can we fix this mess #5 ?

- Linux has a little known mount option `MNT_NOSYMFOLLOW`.
  - This does **exactly** what we need !
  - Allows symlinks to be created and read on a mounted filesystem, but any attempt to traverse a path containing a symlink returns `ELOOP`.
- Breaks applications in the “right” way.
- Allows application vendors to declare – “This application is only secure if run on a file system mounted with `MNT_NOSYMFOLLOW`.”
- Doesn’t seem to be available as a “mount” command option.

# Conclusion

- Short term (on Linux) `MNT_NOSYMFOLLOW` is my preferred choice.
- Keeps existing symlink requirements for normal apps (systemd, kernel name links etc.)
- Allows specific applications to opt out of symlink insanity.
  - Still allows symlinks to be stored and followed manually if the application is coded that way.
  - Turns symlinks into Windows “shortcuts”.
- Proselytize the “no more symlinks” creed !
- Let’s eliminate symlink race CVEs by 2032 !

# Questions and Comments ?

Email: [jra@samba.org](mailto:jra@samba.org)  
[jra@google.com](mailto:jra@google.com)

Slides available at: