Safe Open

Safely Opening a File

Problem: **Being sure that the file you try to open is the one that you actually open.**

Common solution: **Using the standard system calls or library routines, e.g., `open` or `fopen`.**

Risk: **An attacker, without root privilege, running on the same host manipulates the file or its path. The attack is more serious if *your program* is running as root.**

Solution: **Let's look at some background material and examples, then try to construct a solution…**

(More details:

http://www.cs.wisc.edu/mist/safefile

James A. Kupsch and Barton P. Miller, "How to Open a File and Not Get Hacked," *2008 Third International Conference on Availability, Reliability and Security* (ARES), Barcelona, Spain, March 2008.)

Building a Safe Open Facility

First, some background:

- Directories and paths
- The Posix `open` call, up close

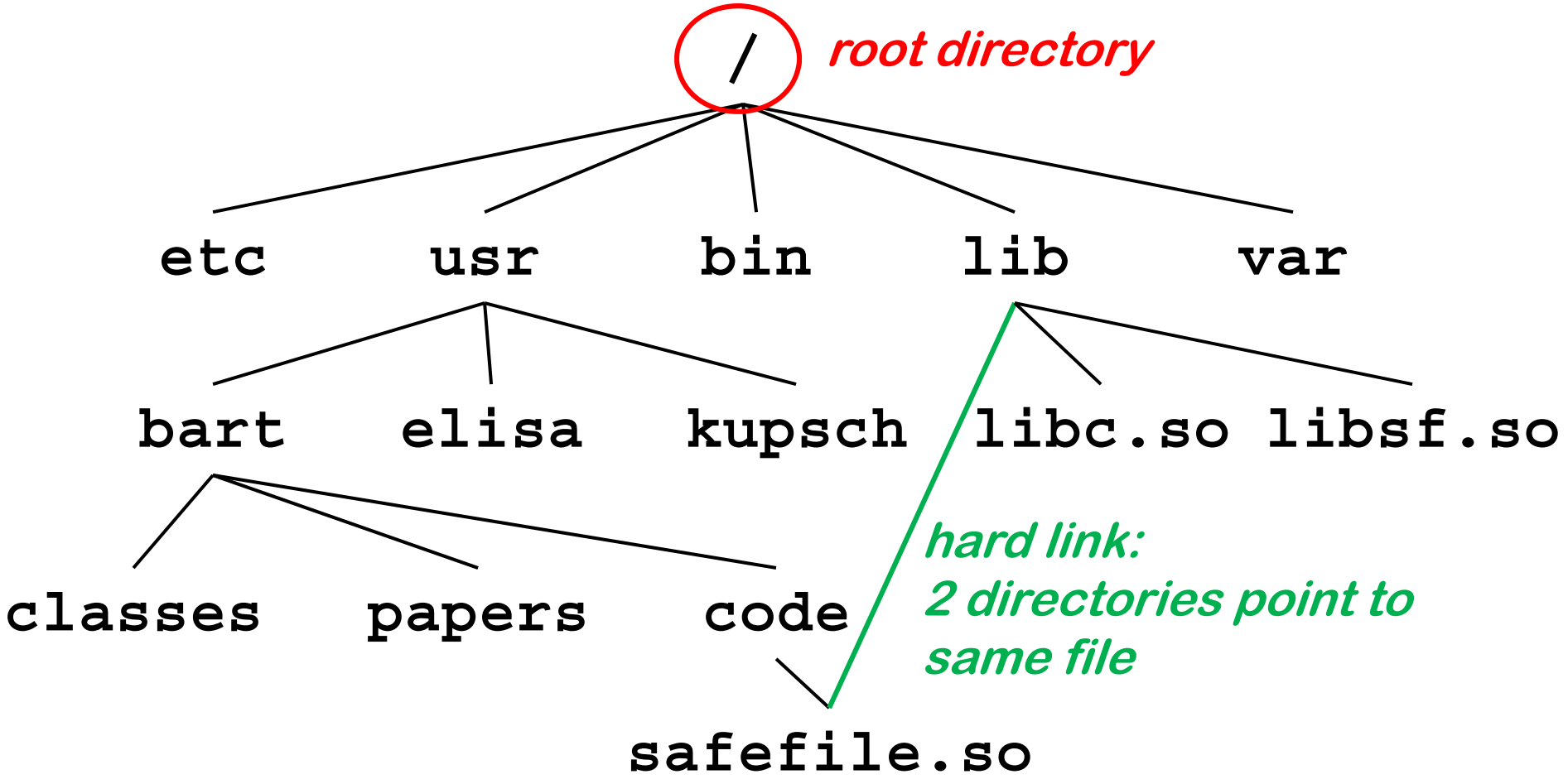
Next, specify the desired properties of a safe `open`

Last, build the suite of open calls.

Directories and Paths

/usr/bart/code/safefile.so

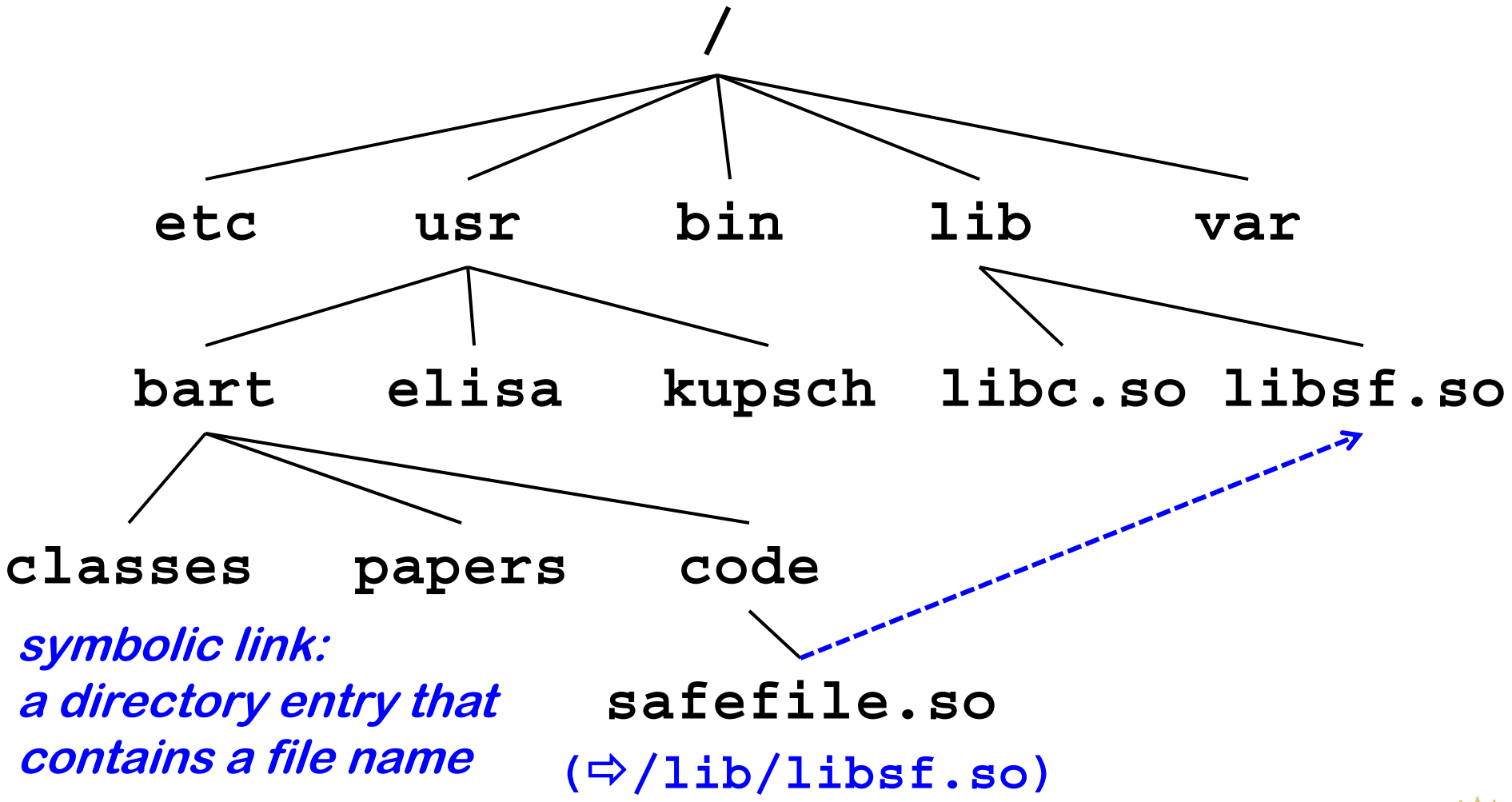
/lib/safefile.so



Directories and Paths

/usr/bart/code/safefile.so

points to: /lib/libsf.so



symbolic link:
a directory entry that contains a file name

Directories and Paths

How to create these links in Linux:

Hard link:

```
cd /lib
ln /usr/bart/code/safefile.so
```

Symbolic link:

```
cd /usr/bart/code
ln -s safefile.so /lib/libsf.so
```

Directories and Paths

Some path and directory terminology:

Absolute path: A path starting at the root directory, i.e., one that starts with a "/".

Relative path: A path that starts with the current working directory.

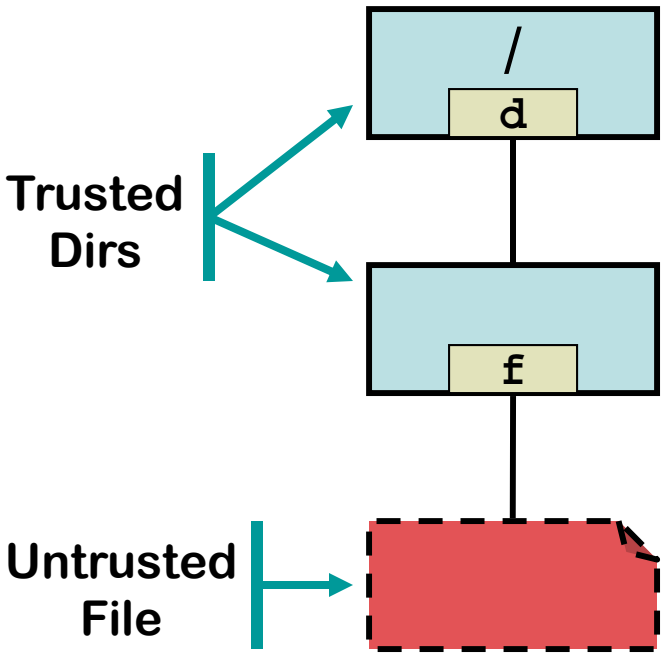
Static path: An absolute path without any symbolic links.

Canonical path: A static path that doesn't contain "." or "..".

Sticky bit directory: A remove only succeeds if the uid of the process is either (1) root, (2) owner of the directory, or (3) owner of the file.

Attack: Untrusted File

Program's Goal: open file `/d/f`, assume only trusted users can change the file



Program (P) / Attacker (A)

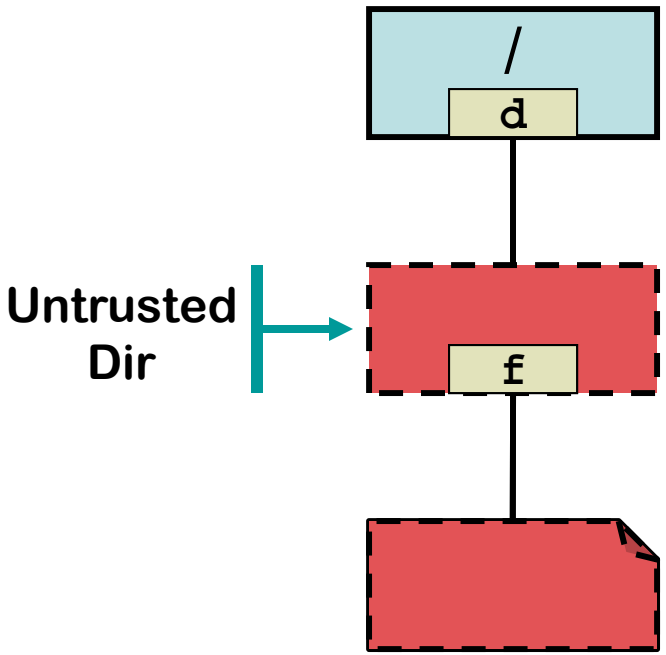
```
P: fopen("/d/f", "r")
A: fopen("/d/f", "w")
A: write bad data
P: read data
P: use data
```

Problem: untrusted users can modify `/d/f`.

Uses untrusted data.

Attack: Untrusted Directory

Program's Goal: check trust of `/d/f`, assume only trusted users can control contents of `/d/f`



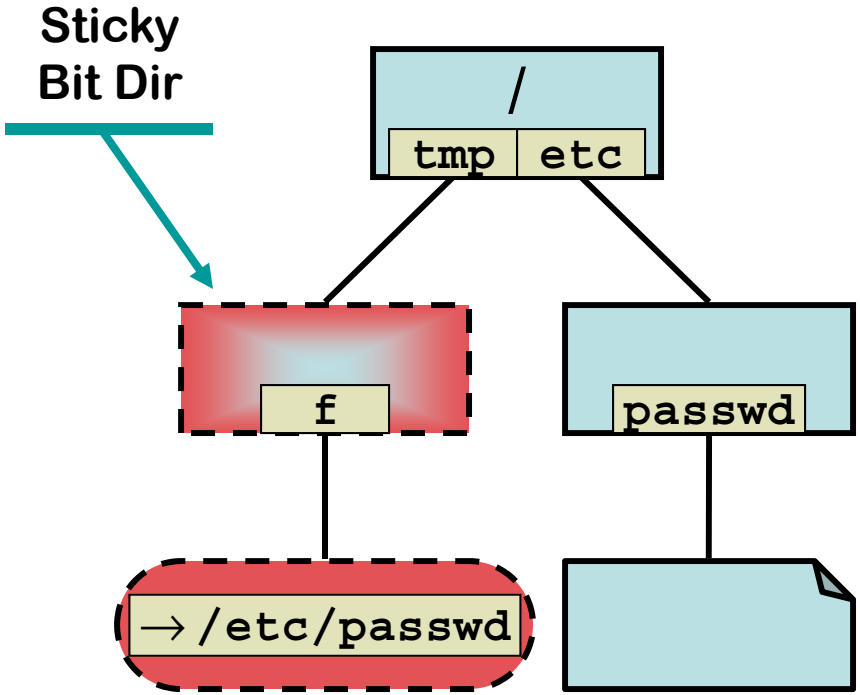
Program (P) / Attacker (A)

```
P: stat("/d/f")
P: check trust using stat
A: unlink("/d/f")
A: creat("/d/f")
A: write bad data
P: fopen("/d/f", "r")
P: read data
P: use data
```

Problem: untrusted users can remove and create files in `/d`. Denial of Service after unlink, Uses untrusted data after create.

Attack: Symbolic link

Program's Goal: create file `f` in directory `/tmp`



Program (P) / Attacker (A)

A: `symlink("/etc/passwd", "/tmp/f")`

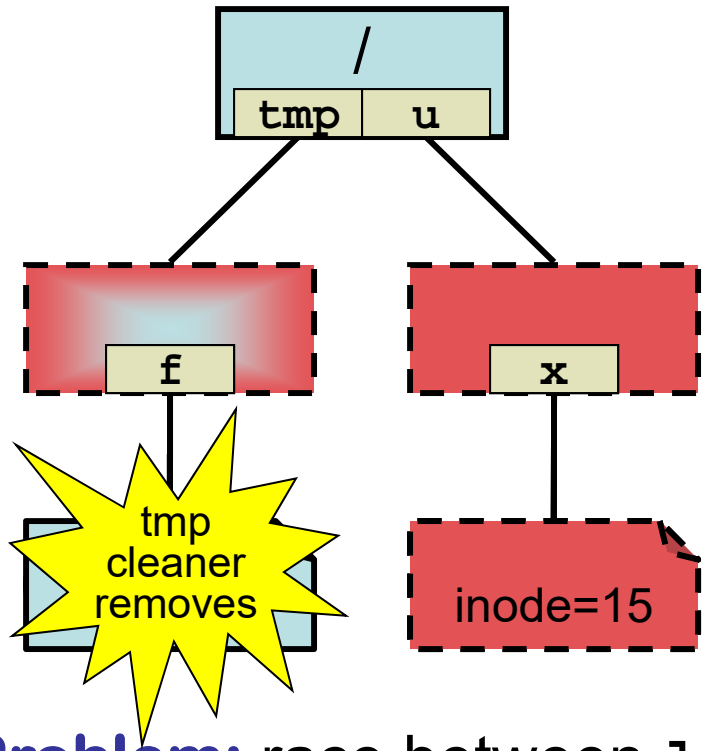
P: `fopen("/tmp/f", "w")`

P: `write data`

Problem: untrusted user can pick file that is opened/created as trusted user. Attacker causes wrong file to be opened/created.

Attack: Cryogenic Sleep

Program's Goal: use `lstat` to check trust of `/tmp/f`. If good, open `/tmp/f`. If same object, trust content and location.



Program (P) / Attacker (A)

```
P: lstat("/tmp/f")
P: check trust from lstat
A: sleep/delay program
A: wait until /tmp/f removed
A: symlink("/u/x", "/tmp/f")
A: unlink("/u/x");creat("/u/x")
     until dev/inode match
A: resume program
P: open("/tmp/f", O_RDONLY)
P: fstat
P: check dev/inode match
P: use data
```

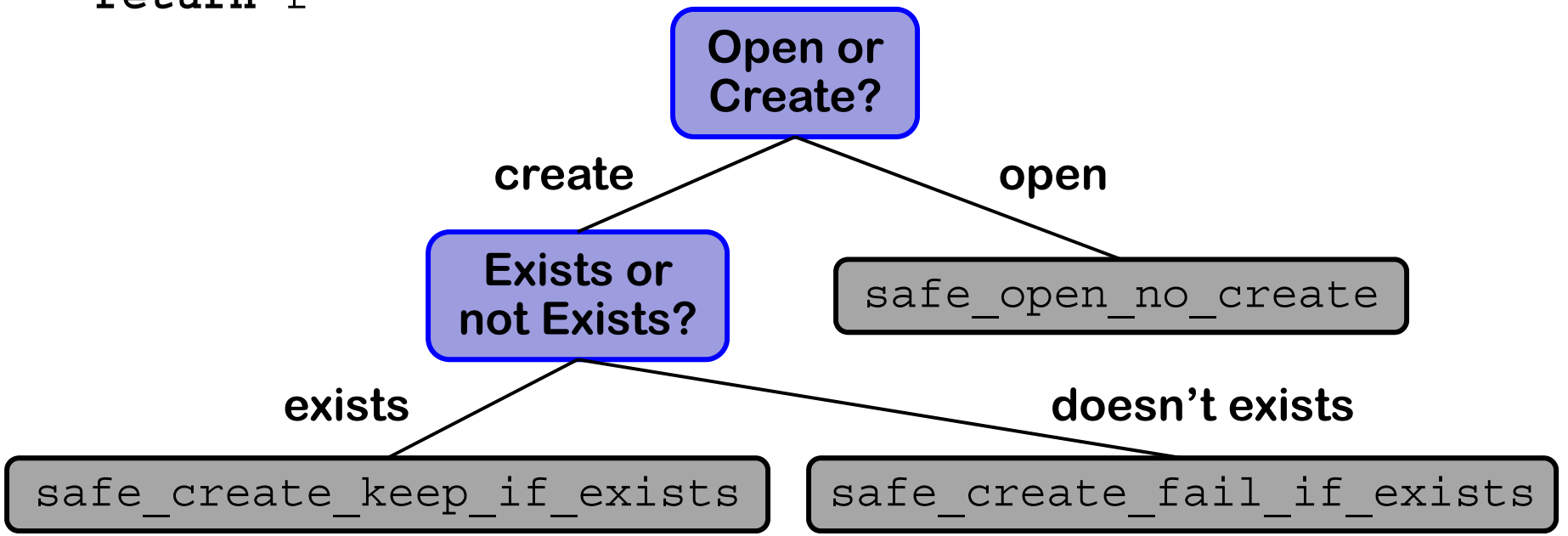
Problem: race between `lstat` and `open`. Tmp file cleaner removes `/tmp/f`. Attacker gets program to open untrusted file.

Desired Properties for Safe Open

- Never use O_CREAT without O_EXCL
- When opening an existing file, don't allow last component of path to be symbol link.
- For file create operations, insure that explicit permissions are always specified.
- Function as if the open/create operation was atomic so that no race (TOCTOU) attacks are possible.
- Work as drop-in replacements for existing open system calls.

The Safe Open Wrapper

```
function safe_open_wrapper(fn, flags, perms)
    if (O_CREAT ∈ flags)
        if (O_EXCL ∈ flags)
            f ← safe_create_fail_if_exists(fn, flags, perms)
        else
            f ← safe_create_keep_if_exists(fn, flags, perms)
    else
        f ← safe_open_no_create(fn, flags)
    return f
```



```
function safe_open_no_create(fn, flags)
    if ((O_CREAT or O_EXCL) ∈ flags) return EINVAL
    want_trunc ← (O_TRUNC ∈ flags)
    if (want_trunc) flags ← flags - {O_TRUNC}
  TRY_AGAIN:
    f ← open(fn, flags)
    entryStat ← lstat(fn)
    if (entryStat < 0 and f < 0) return error from lstat
    if (entryStat < 0 and f ≥ 0)
        close(f)
        goto TRY_AGAIN
    if (S_ISLNK(entryStat))
        if (f ≠ -1) close(f)
        return EEXIST
    if (open failed with ENOENT) goto TRY_AGAIN
    if (f < 0) return error from open
    fdStat ← fstat(f)
    if (entryStat and fdStat refer to different files)
        close(f)
        goto TRY_AGAIN
    if (want_trunc and fdStat.st_size ≠ 0) ftruncate(f,0)
    return f
```

Insure that we use a safe parameter set.

Insure we don't truncate file until we know it's the right file.

Simple: normal error if open fails.

Did the file disappear and another appear in its place? (Cryo Sleep Attack)

Are we in the middle of a path manipulation?

Return error if last element of path name is a symbolic link.

```
function safe_create_fail_if_exists(fn, flags, perms)
   flags ← flags + {O_CREAT,O_EXCL}
   return open(fn, flags, perms)

function safe_create_keep_if_exists(fn, flags, perms)
   flags ← flags - {O_CREAT,O_EXCL}
   loop forever
      f ← safe_create_fail_if_exists(fn, flags, perms)
      if (f ≠ -1 or errno ≠ EEXIST)
         return f
      f ← safe_open_no_create(fn, flags)
      if (f ≠ -1 or errno ≠ ENOENT)
         return f

function safe_create_replace_if_exists(fn, flags, perms)
   loop forever
      unlink(fn)
      if (unlink failed and errno ≠ ENOENT)
         return -1
      f ← safe_create_fail_if_exists(fn, flags, perms)
      if (f ≠ -1 or errno ≠ EEXIST)
         return f
```