How to Open a File and Not Get Hacked

James A. Kupsch
and Barton P. Miller
University of Wisconsin

SecSE 2008, Barcelona, Spain
March 5th, 2008

SDSC

Background

- Vulnerability assessment project focusing on distributed systems software
- As part of this effort we observed security problems involving
  - opening of files
  - path vulnerabilities

  in all software examined
- Major goal to prevent security problems and educate developers on secure coding

Problem of Safely Opening a File

Almost all programs open a file. Insecure permissions anywhere in a path can be a disaster.

Opening safely should be simple, but is not:

- Standard APIs are not secure by default, much easier to use in an insecure fashion
- No standard API that provides secure semantics
- Subtle semantics for checking trust, due to symbolic links, hard links, sticky bit semantics
- Safe open requires many non-atomic operations and dealing with the concurrency problems that arise

Others have thought about this problem, but they haven't gotten it right.

Threats

Security of a system depends on the security of its files. If a user on the system can attack them, user accounts, applications or the system can easily be compromised.

- **Reveal secrets**
  - `/etc/shadow`
- **Allow unauthorized access**
  - `/etc/passwd` `~/.ssh/authorized_keys`
- **Execute programs**
  - `/etc/rc` `~/.bashrc` `~/.vimrc`
- **Prevent operation**
  - overwrite file contents

Trust

- Trusted users - trust not to do anything bad
- Trusted path - safe from attack
  - only trusted users can modify
    - which file
    - file contents
  - less precautions needed, most attacks are prevented
  - most applications incorrectly assume paths are trusted

Strategies for Safe Open

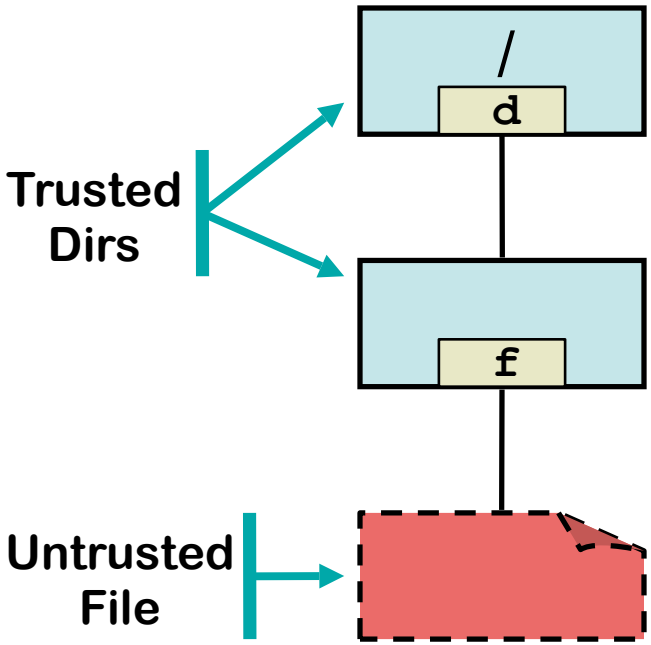
- Verify Path is Trusted
  - Do not use if not trusted
- Safely Open an untrusted file
  - Prevents common security problems with
    - symbolic links
    - misuse of the API leading to weak permissions
  - Detects attacks of the path

Safe Coding Practices

- Active field
  - Many books
  - Prior work on this problem
    - Viega & McGraw - Building Secure Software
    - Bishop - SANS 2002 Tutorial
  - US CERT Secure Coding Standards http://www.securecoding.cert.org
  - ISO/IEC TR 24731: C library extensions: Bounds checking of string values and I/O safety
- None correctly describe
  - checking the trust of a path
  - complete safer `open` and `fopen` replacements

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`

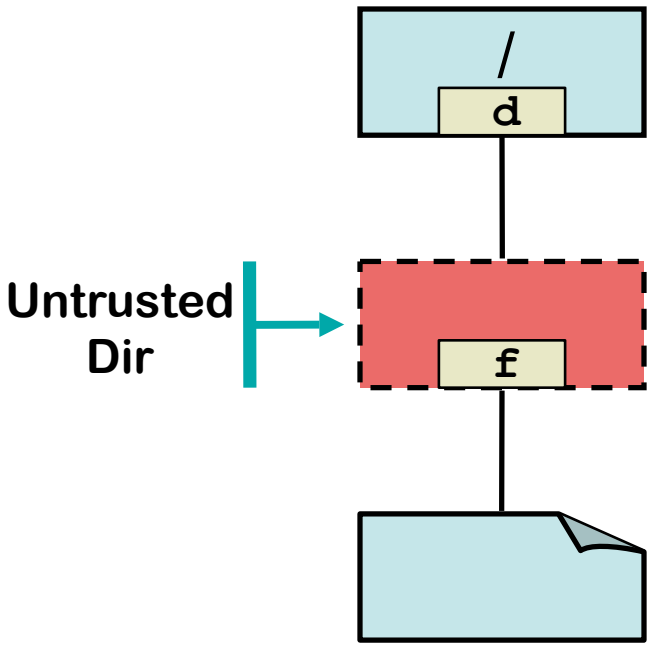
P: `use data`

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`**



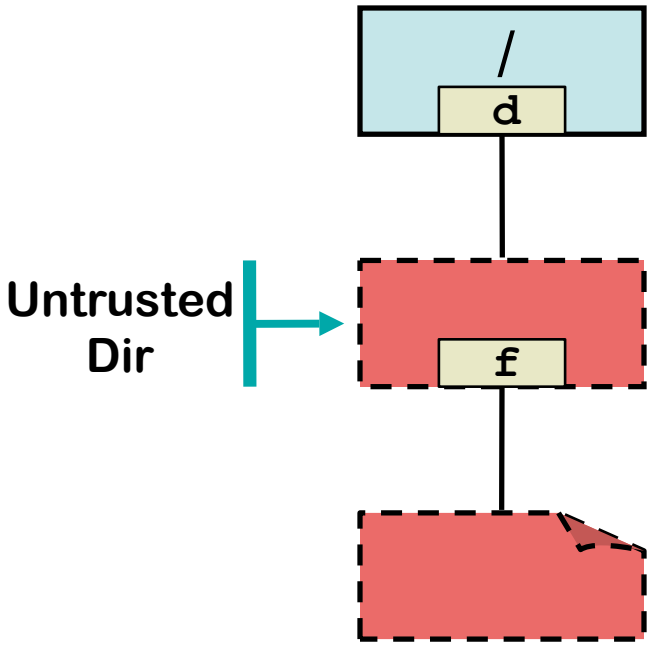
Program (P) / Attacker (A)

```
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: Untrusted Directory

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



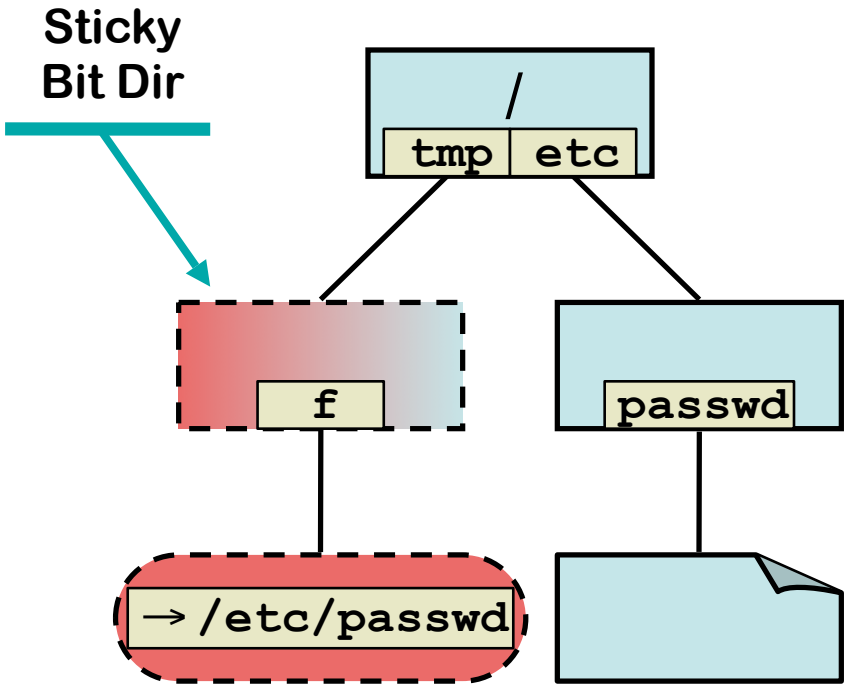
Program (P) / Attacker (A)

```
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 `/d`



Program (P) / Attacker (A)

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

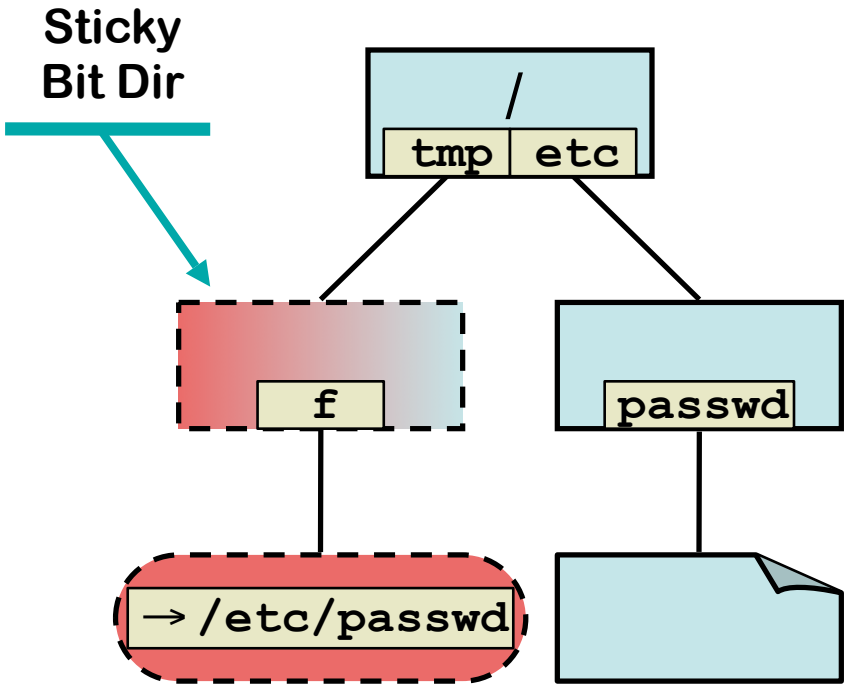
P: `fopen("/tmp/f", "w")`

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: Symbolic link

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



Program (P) / Attacker (A)

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

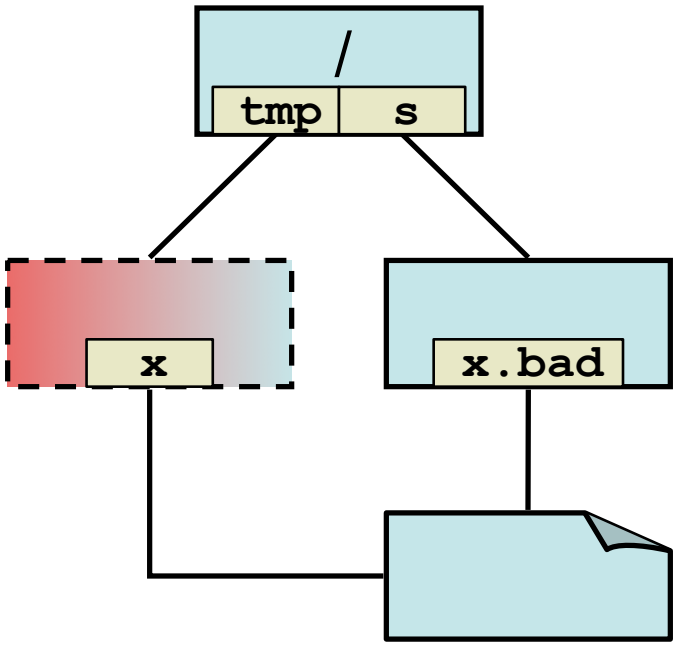
P: `fopen("/tmp/f", "w")`

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: Hard links

Program's Goal: assume that trusted user created file **`f`** in the **`/tmp`** (sticky bit set) directory if perms of **`f`** are trusted.



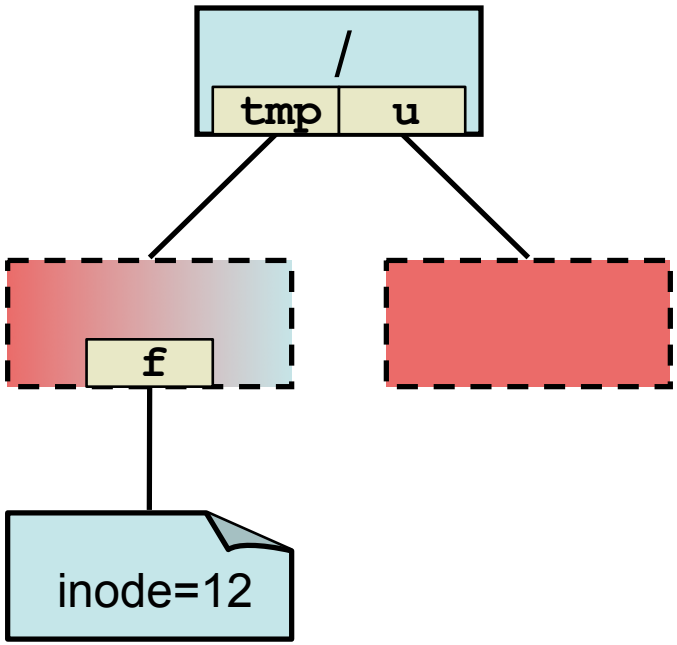
Program (P) / Attacker (A)

```
A: link("/s/x.bad",
                "/tmp/x")
P: fopen("/tmp/x", "r")
P: fstat
P: check trust using fstat
P: use data
```

Problem: any user can create a hard link to any other user's file. Application thinks it created a directory entry it didn't.

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.



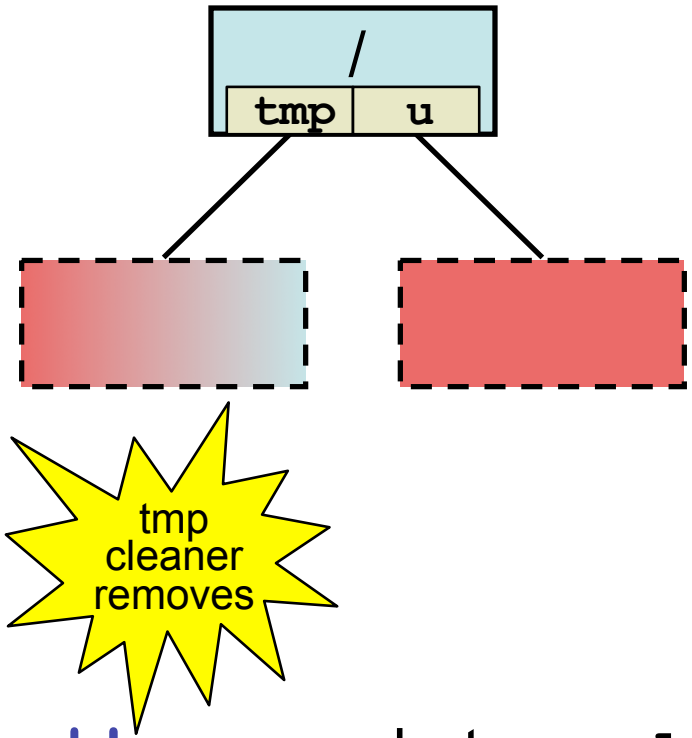
Program (P) / Attacker (A)

```
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.

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.



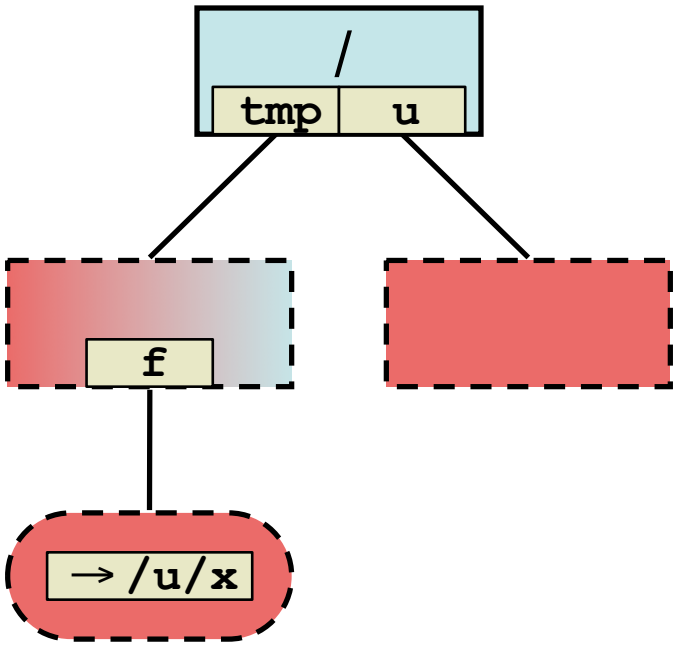
Program (P) / Attacker (A)

```
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.

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.



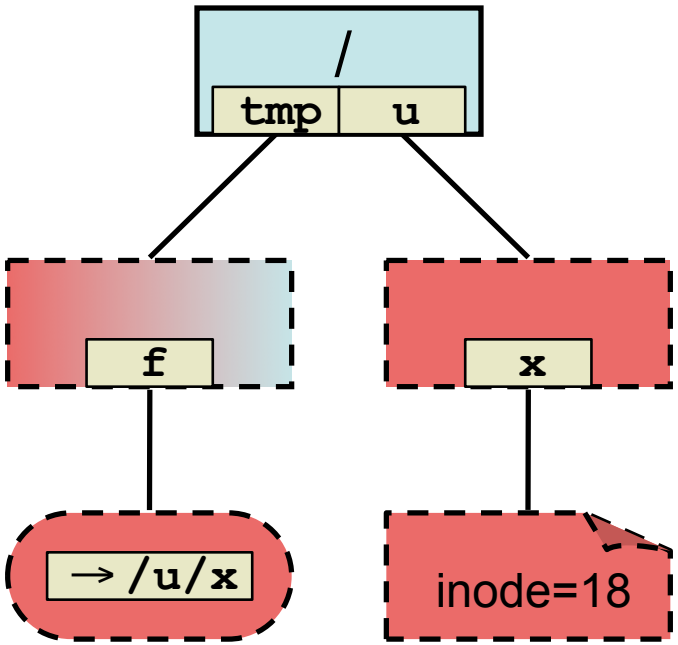
Program (P) / Attacker (A)

```
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.

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.



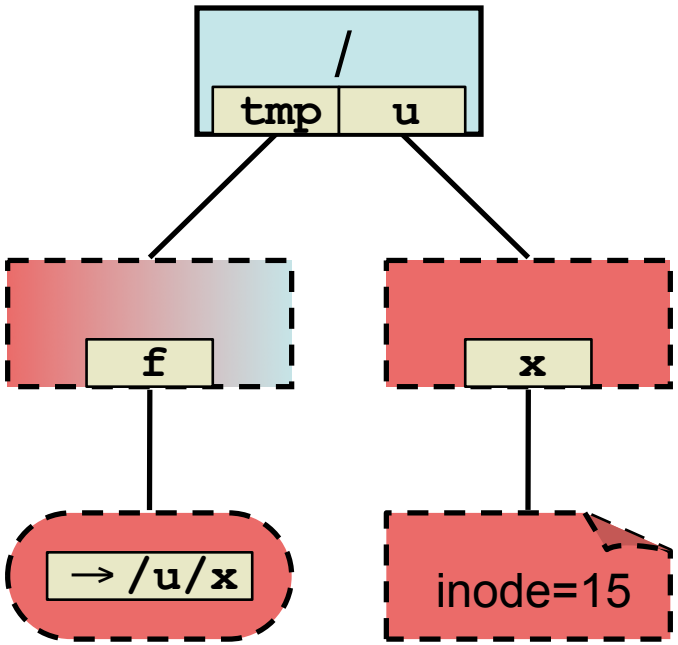
Program (P) / Attacker (A)

```
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.

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.



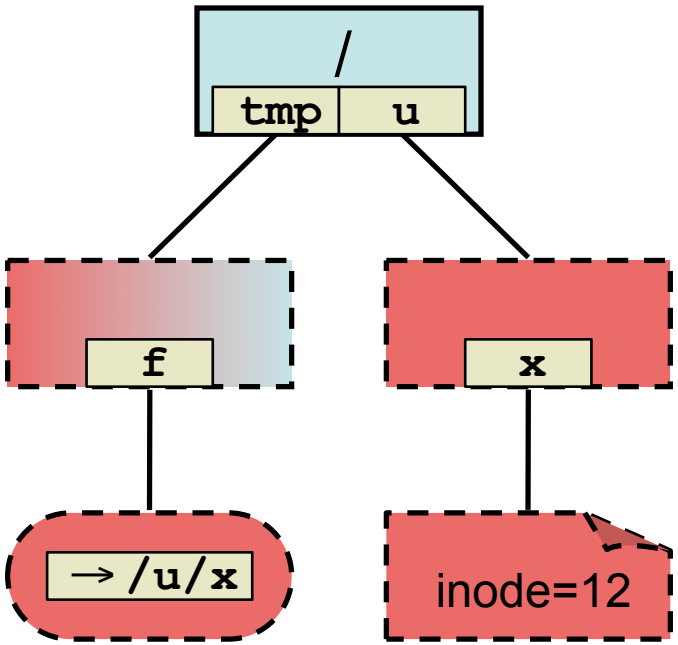
Program (P) / Attacker (A)

```
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.

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.



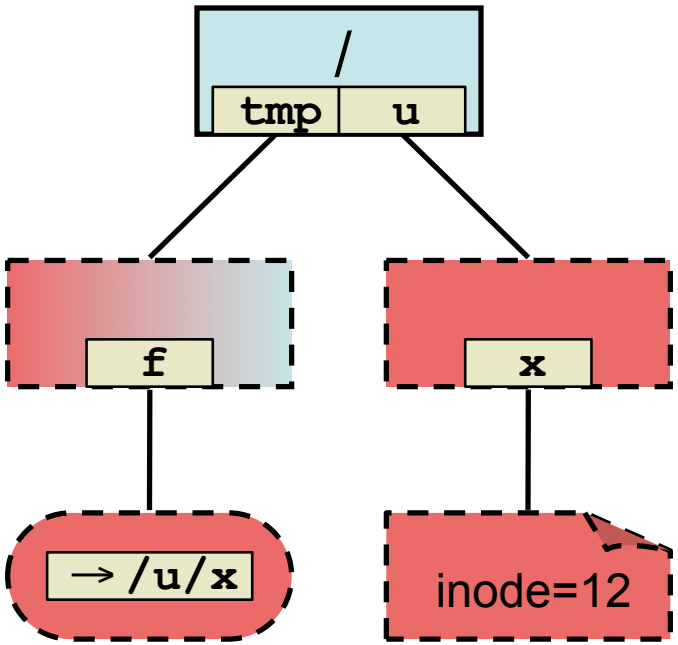
Program (P) / Attacker (A)

```
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.

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.

Trusted Path

- Trusted path - only the set of trusted users and groups can modify
  - which object the path refers
  - the contents of the object
- Especially important for applications with elevated privileges
- Not secure to check just the trust of
  - the object
  - directories from the object to the root directory

Trust of a Directory Entry

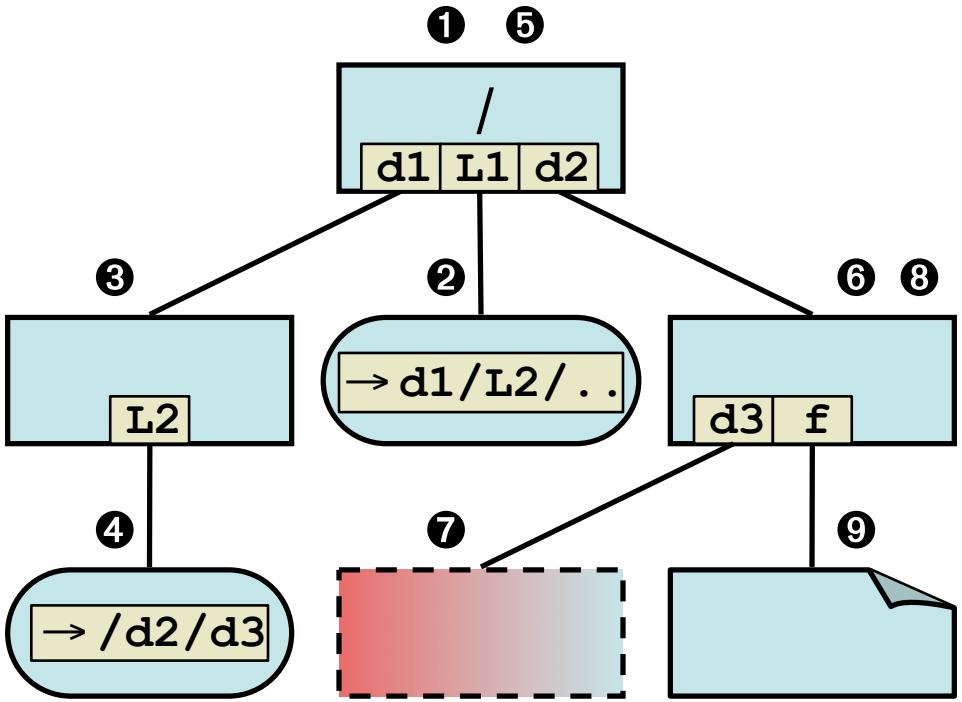
- Trust of a file system object is determined by
  - Permission bits
  - Owner and group
  - Trusted users and groups
  - Sticky bit for directories
- Types of trust
  - Untrusted
  - Trusted
  - Sticky directory trusted (such as /tmp) - limited trust to
    - Directories inside
    - Files you create inside and only access through returned file descriptor

Trust of a Path

- Must check every object encountered along the path and all must be trusted
  - check the same objects as the OS
  - relative paths must check all directories from current to root
  - if the parent directory is sticky dir trusted, non-directories are not trusted
  - handle symbolic links
    - referent path must be checked before proceeding
    - detect loops
    - handle large path lengths

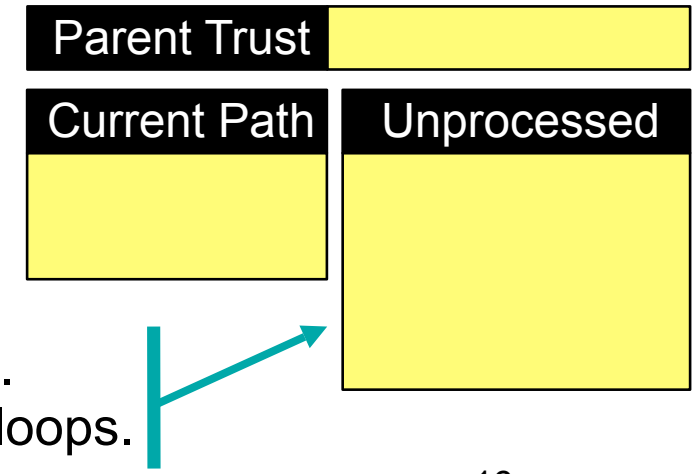
safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **`/L1/f`**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

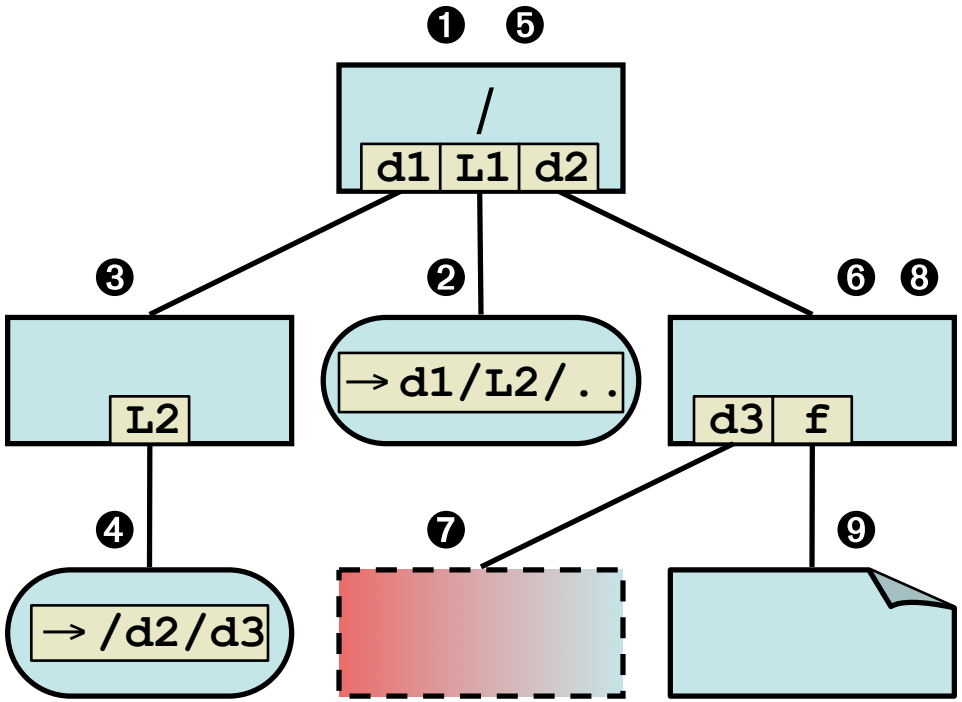
Internal State



Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

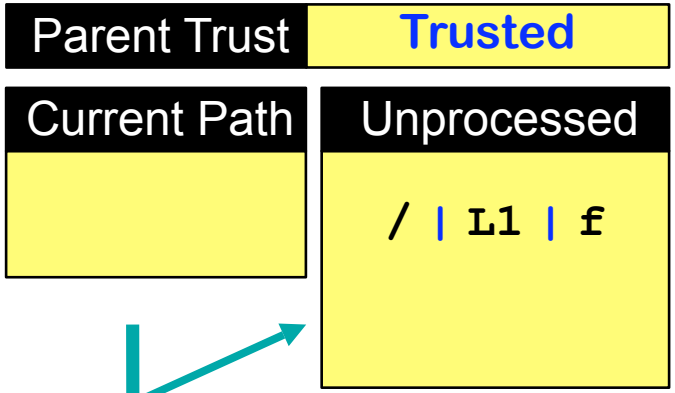
safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **/L1/f**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

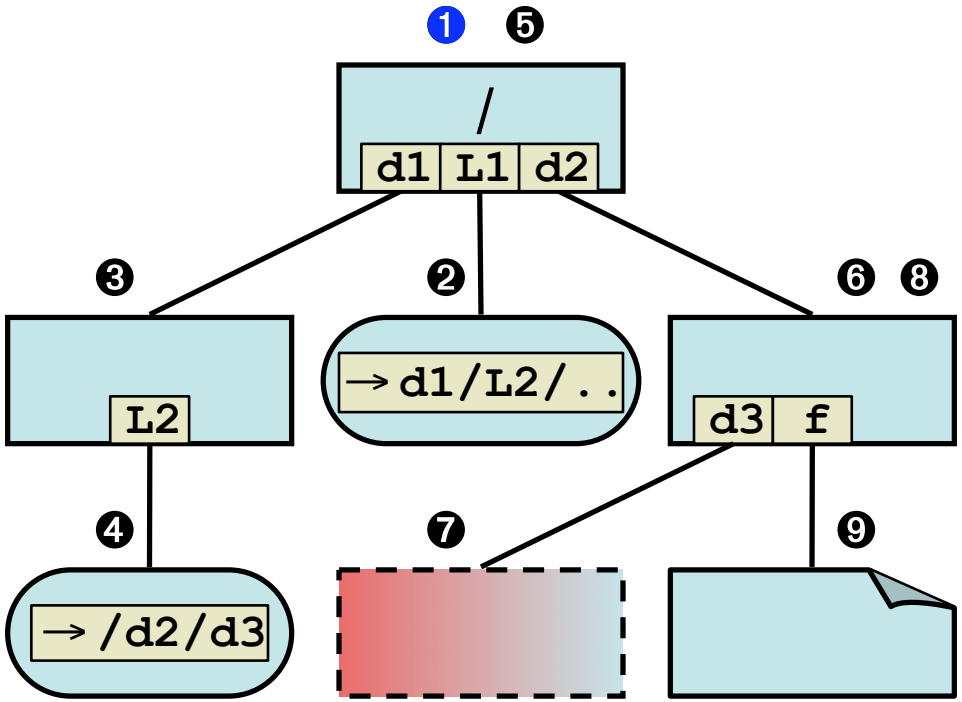
Internal State



Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **/L1/f**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

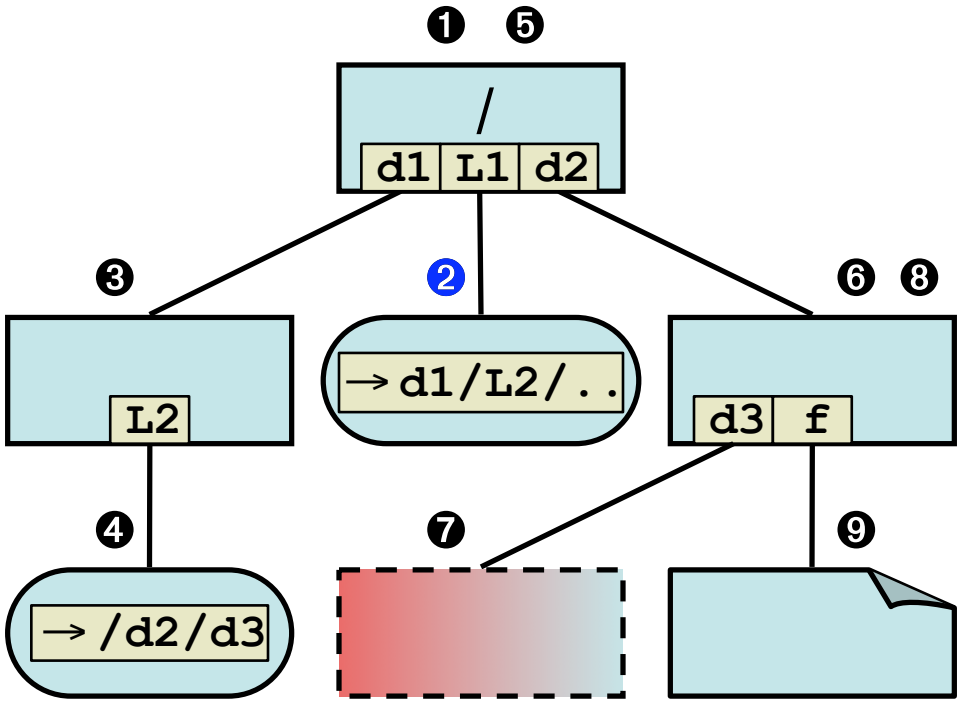
Internal State

Parent Trust	Trusted
Current Path	Unprocessed
/	L1 | f

Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **/L1/f**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

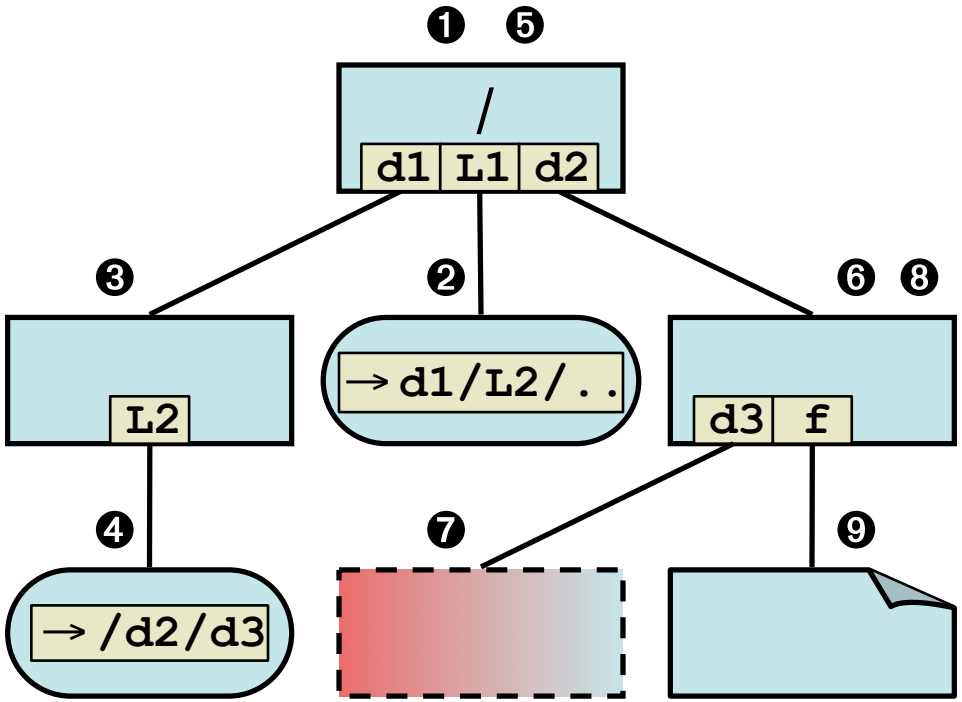
Internal State

Parent Trust	Trusted
Current Path	Unprocessed
/L1	f

Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **`/L1/f`**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

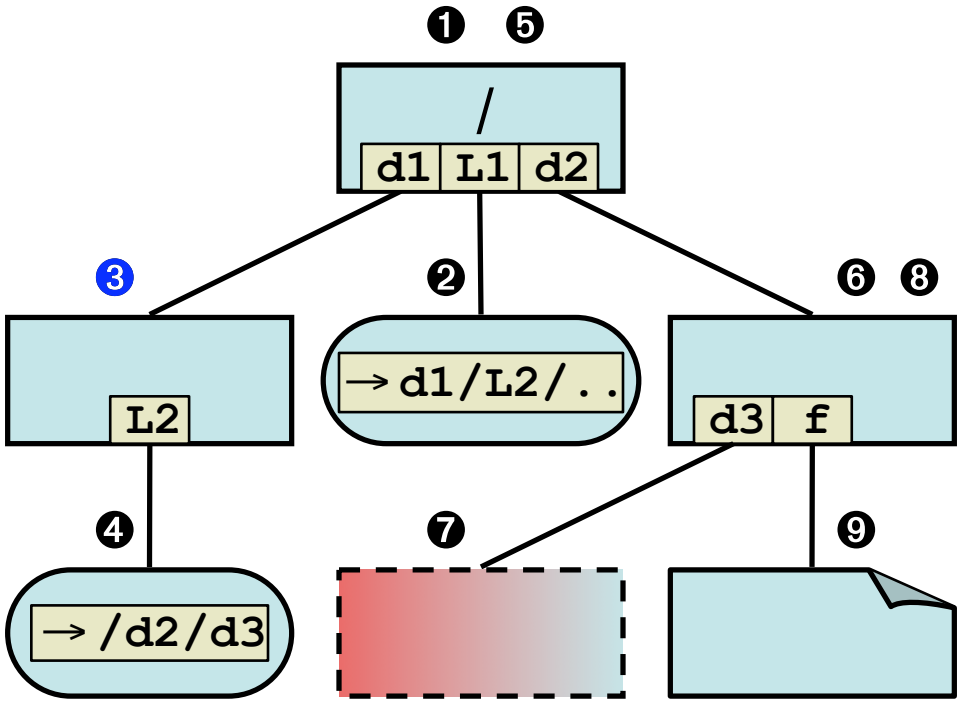
Internal State

Parent Trust	Trusted
Current Path	Unprocessed
/	d1 | L2 | .. f

Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **/L1/f**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

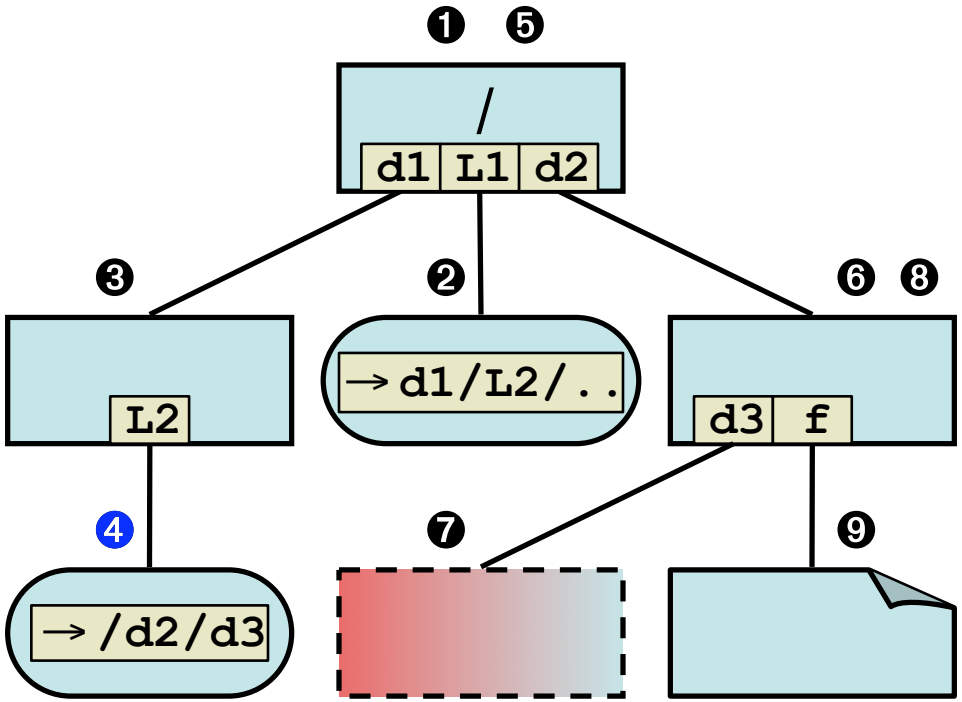
Internal State

Parent Trust	Trusted
Current Path	Unprocessed
/d1	L2 | .. f

Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **/L1/f**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

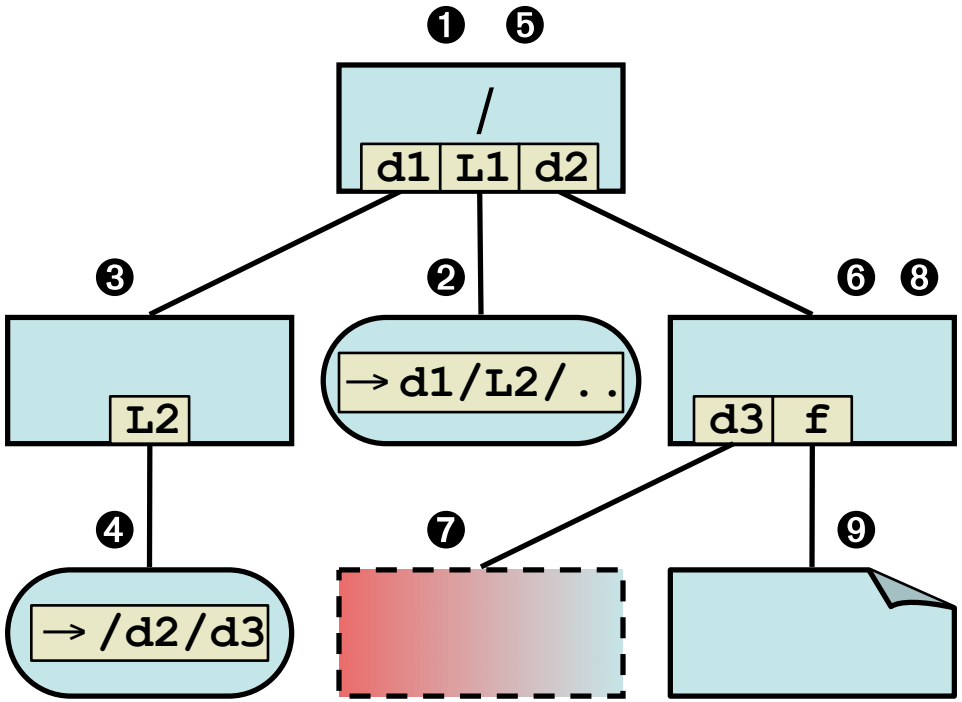
Internal State

Parent Trust	Trusted
Current Path	Unprocessed
/d1/L2	.. f

Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **/L1/f**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

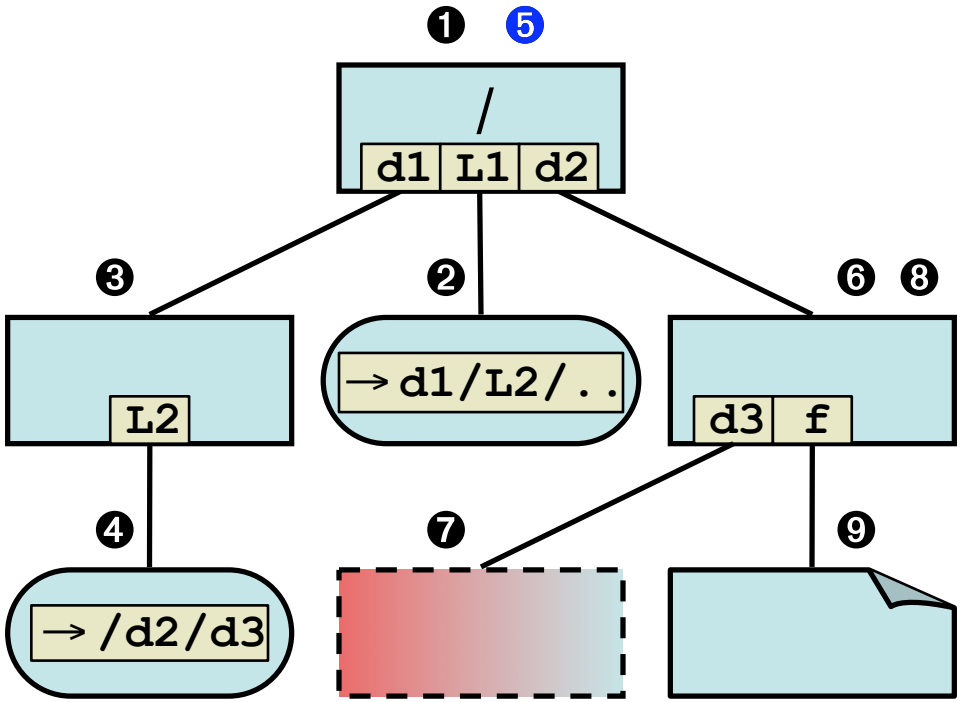
Internal State

Parent Trust	Trusted
Current Path	Unprocessed
/d1	/ | d2 | d3 .. f

Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

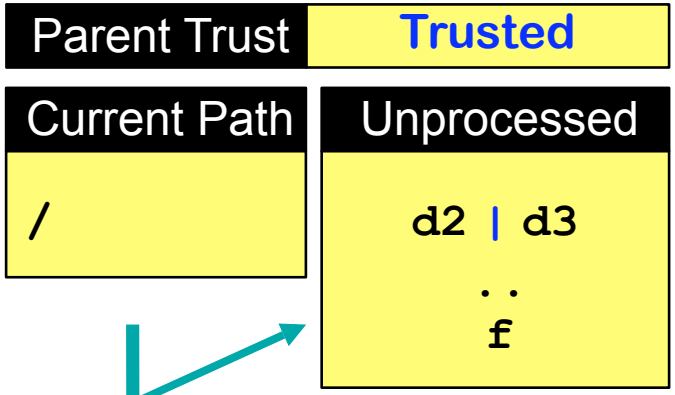
safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **`/L1/f`**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

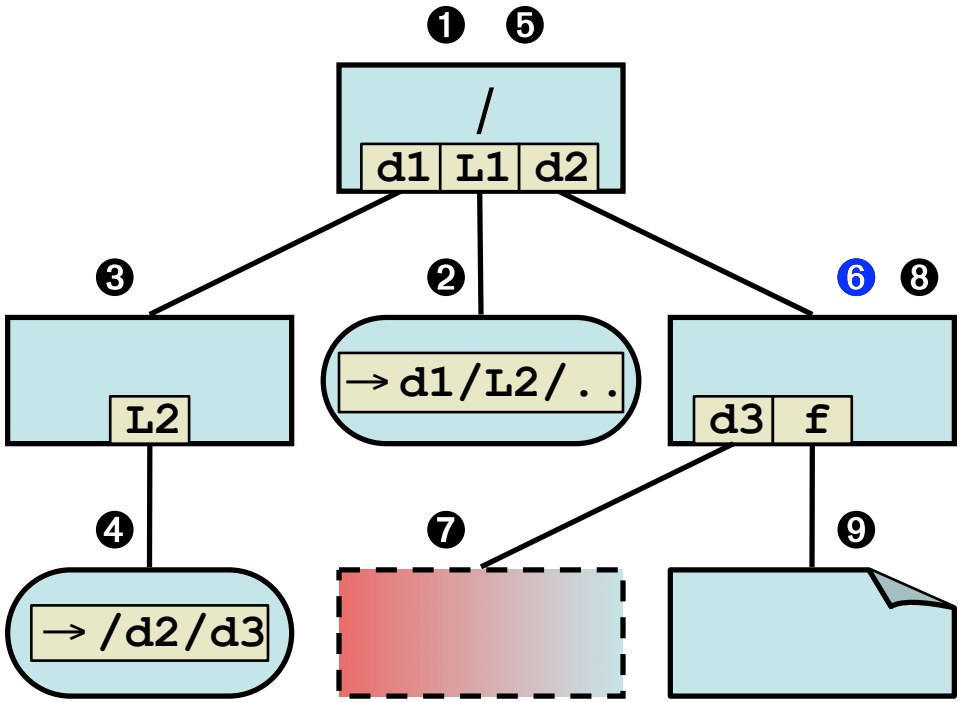
Internal State



Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

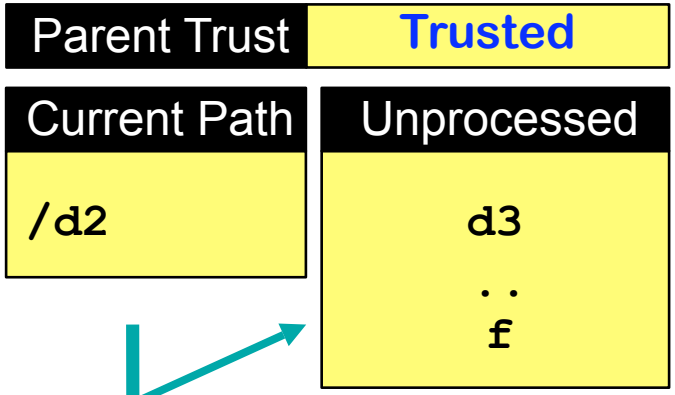
safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **`/L1/f`**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

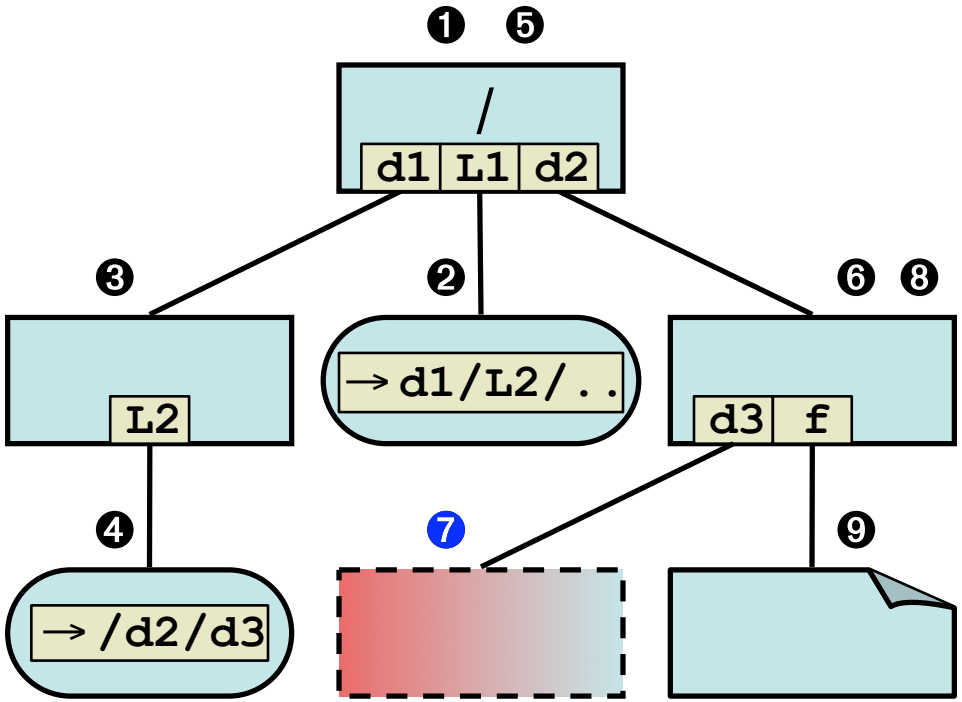
Internal State



Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **`/L1/f`**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

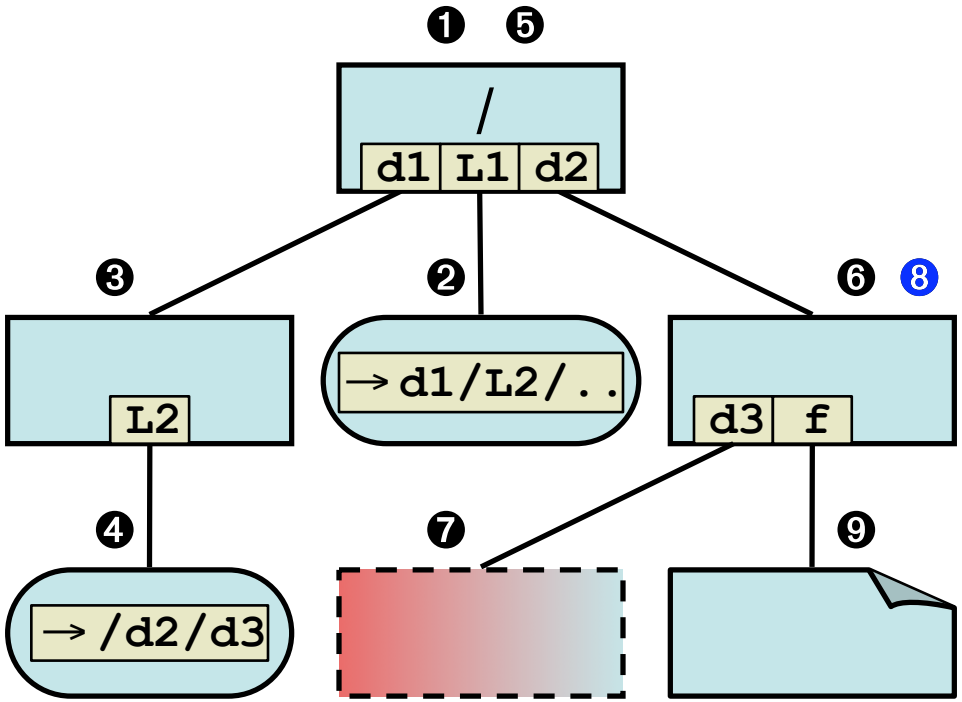
Internal State

Parent Trust	Sticky Trust
Current Path	Unprocessed
/d2/d3	.. f

Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **`/L1/f`**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

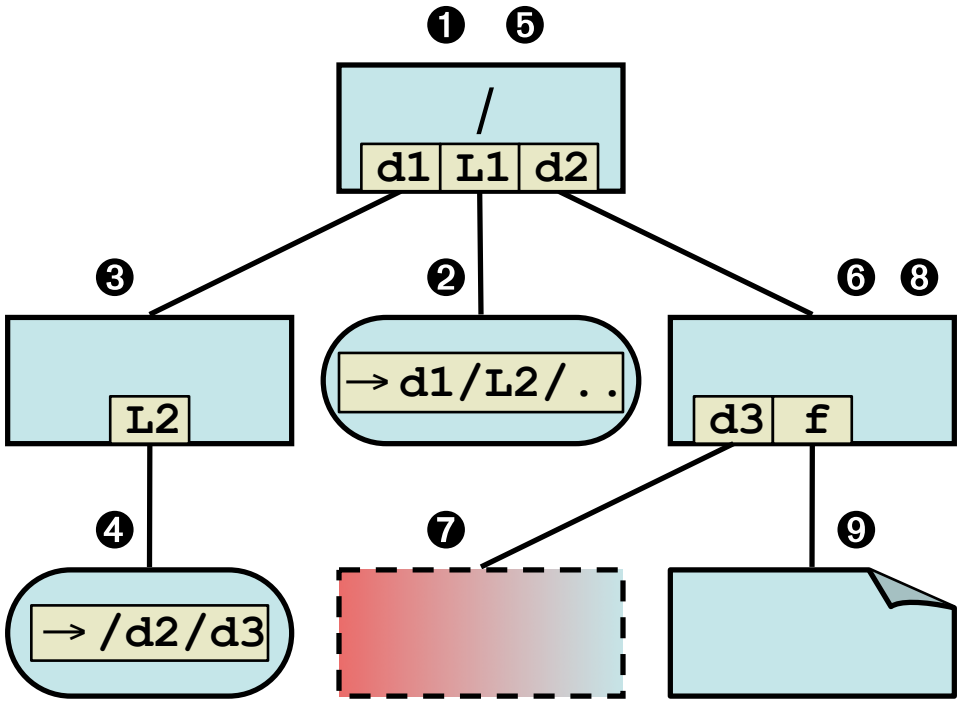
Internal State

Parent Trust	Trusted
Current Path	Unprocessed
/d2/d3/..	f

Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

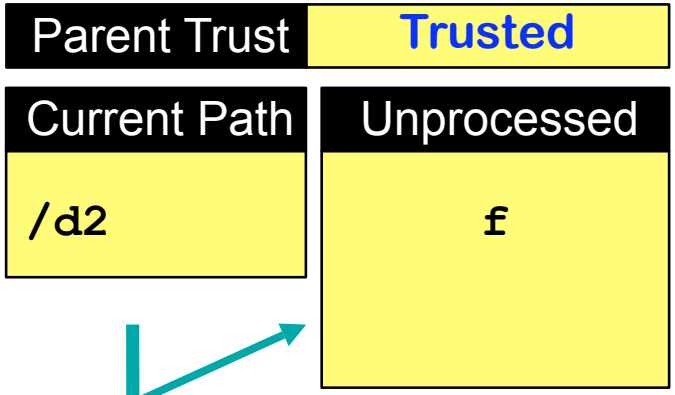
safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **`/L1/f`**. The trust of each object is checked in the same order as the OS.



Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

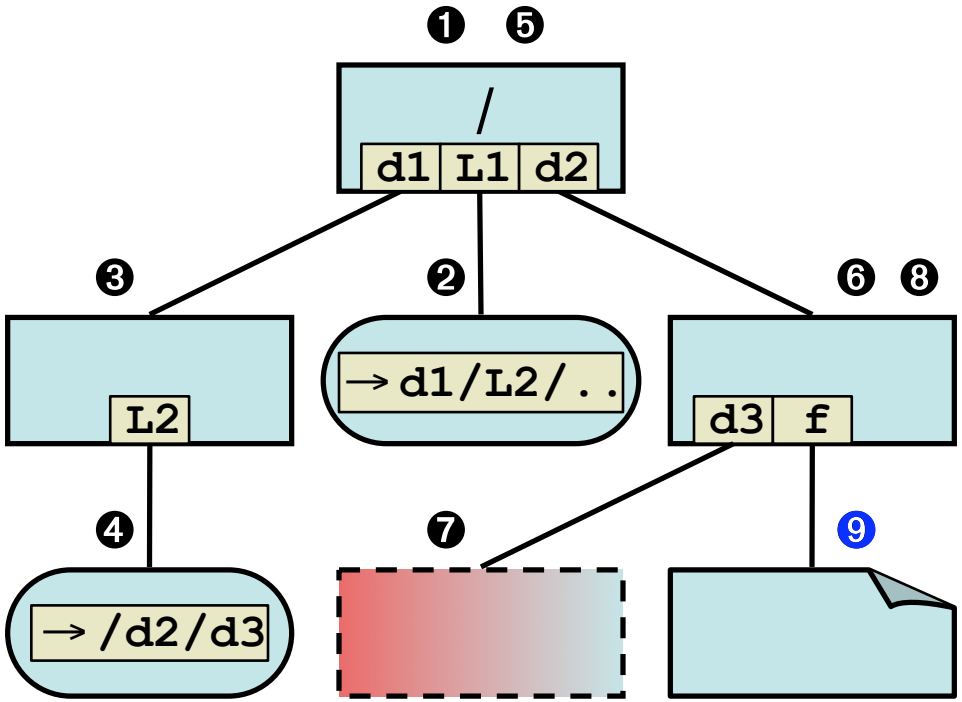
Internal State



Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

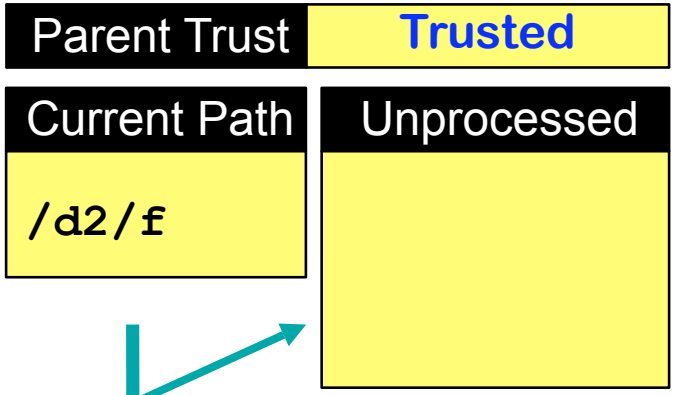
safe_is_path_trusted_r

safe_is_path_trusted_r checking the trust of **/L1/f**. The trust of each object is checked in the same order as the OS.



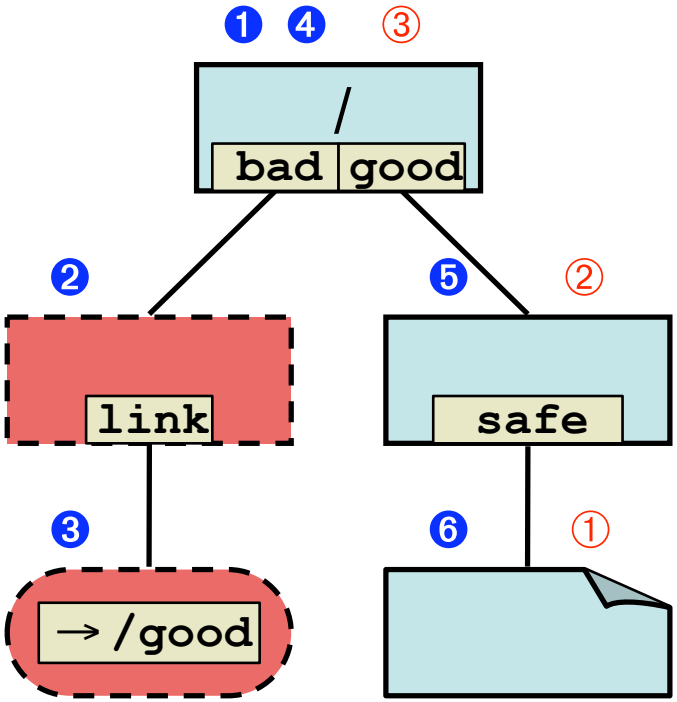
Processing stops when:
- untrusted object found
- unprocessed is empty
- an error occurs

Internal State



Symbolic link referents are pushed on the stack.
The stack depth is used to detect symbolic link loops.

Prior Work - `safe_dir`



Ex. check `/bad/link/safe`

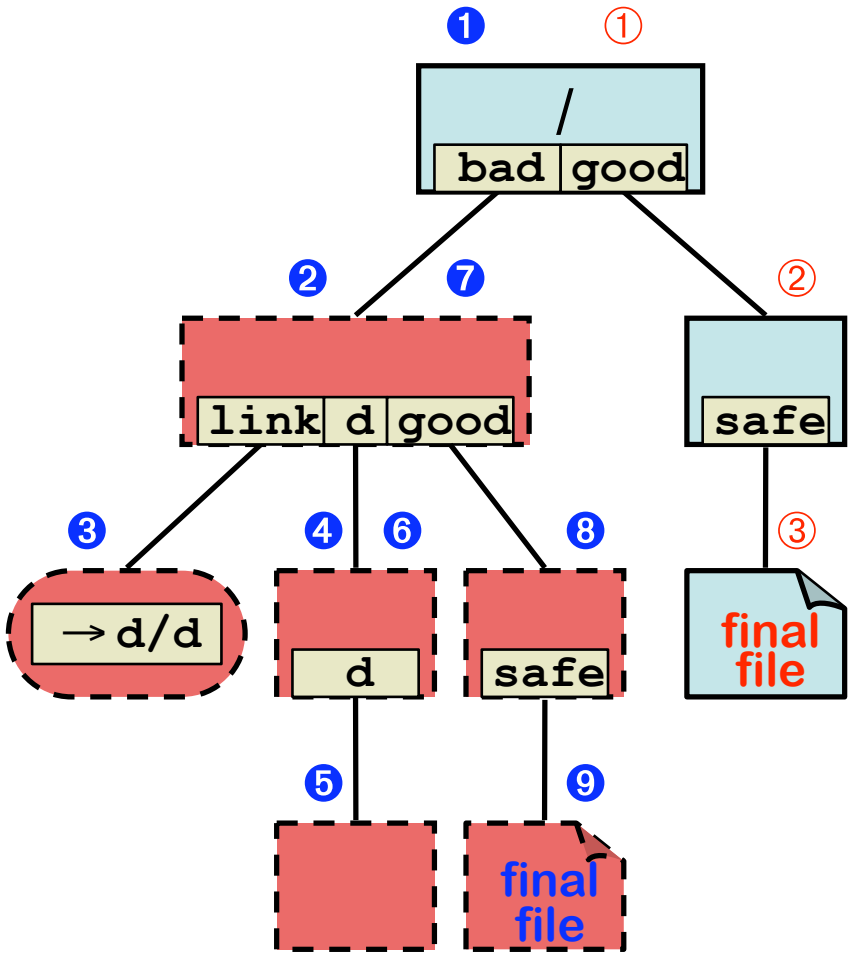
❶ OS order

① `safe_dir` order

After check attacker can change link

- McGraw & Viega in Building Secure Software
  - only directory paths
  - checks directories from path directly to root directory, not the path given
  - correct for '.' path (current working dir)

Prior Work - `trustfile`



- Bishop - SANS2002 Tutorial
  - unsafe textual transform `A/B/../C` to `A/C` before check
    - not valid if `B` is a symbolic link or not trusted
  - fails to detect symbolic link loops

Ex. check `/bad/link/../../good/safe`
Incorrectly checks `/good/safe`

❶ OS order
① `trustfile` order

Trusted Path Algorithm

	McGraw & Viega's safe_dir	Bishop's trustfile	safe_is_path_trusted_r
multiple trusted accounts	✘	✔	✔
supports all file types	✘	✔	✔
supports all valid paths	✔	✘	✔
properly checks symlinks	✘	✘	✔
detects symlink loops	✔	✘	✔
handles sticky bit dirs	✘	✘	✔
efficient	✘	✘	✔
concurrency safe	✘	✔	✔
trusted result is not attackable	✘ ✔ **for path of '.'**	✘	✔

Opening an Untrusted File

- Problems
  - `O_CREAT` w/o `O_EXCL` will follow symbolic link to create files
  - `open` takes 3 arguments
    - 3rd is permissions when creating
    - no warning if missing, get random perms
  - Checks can require `open`, `lstat` and compare
    - `O_TRUNC` is destructive
- Desired solution
  - Similar API
  - Solves above problems
  - Detection of active attacks on path
  - Additional richer functionality

open Replacement Functions

Direct replacement function:

```
safe_open_wrapper(filename, flags, perms)
```

Advanced replacement functions:

```
safe_open_no_create
safe_create_fail_if_exists
safe_create_keep_if_exists
safe_create_replace_if_exists
```

Symbolic link following to an existing file:

```
safe_open_wrapper_follow
safe_open_no_create_follow
safe_open_keep_if_exists_follow
```

Similar functions for `fopen`

All support attack detection mechanism

safe_open_no_create

```
safe_open_no_create(fn, flags)
{
if (O_CREATE or O_EXCL in flags)
    return error

loop
    remove O_TRUNC from flags
    fd = open(fn, flags)
    lstat(fn)
    fstat(fd)
    if (both succeed and are same file)
        if (had O_TRUNC)
            ftruncate(fd, 0)
        return fd
    if (errors are consistent)
        return error
    AttackDetected(fn)
}
```

check for invalid input

cryogenic sleep attack prevented by `open` before `lstat` - prevents dev/inode reuse

handle truncate after check to not truncated wrong file

attacker beat race between `lstat` and `open`; report attack detected and try again

Source Code Available

safefile library and documentation under Apache license at

http://www.cs.wisc.edu/~kupsch/safefile

Source Code Available

safefile library and documentation under Apache license at

http://www.cs.wisc.edu/~kupsch/safefile