

# CS202 – Computer Security

Filesystem Security

March 3, 2025

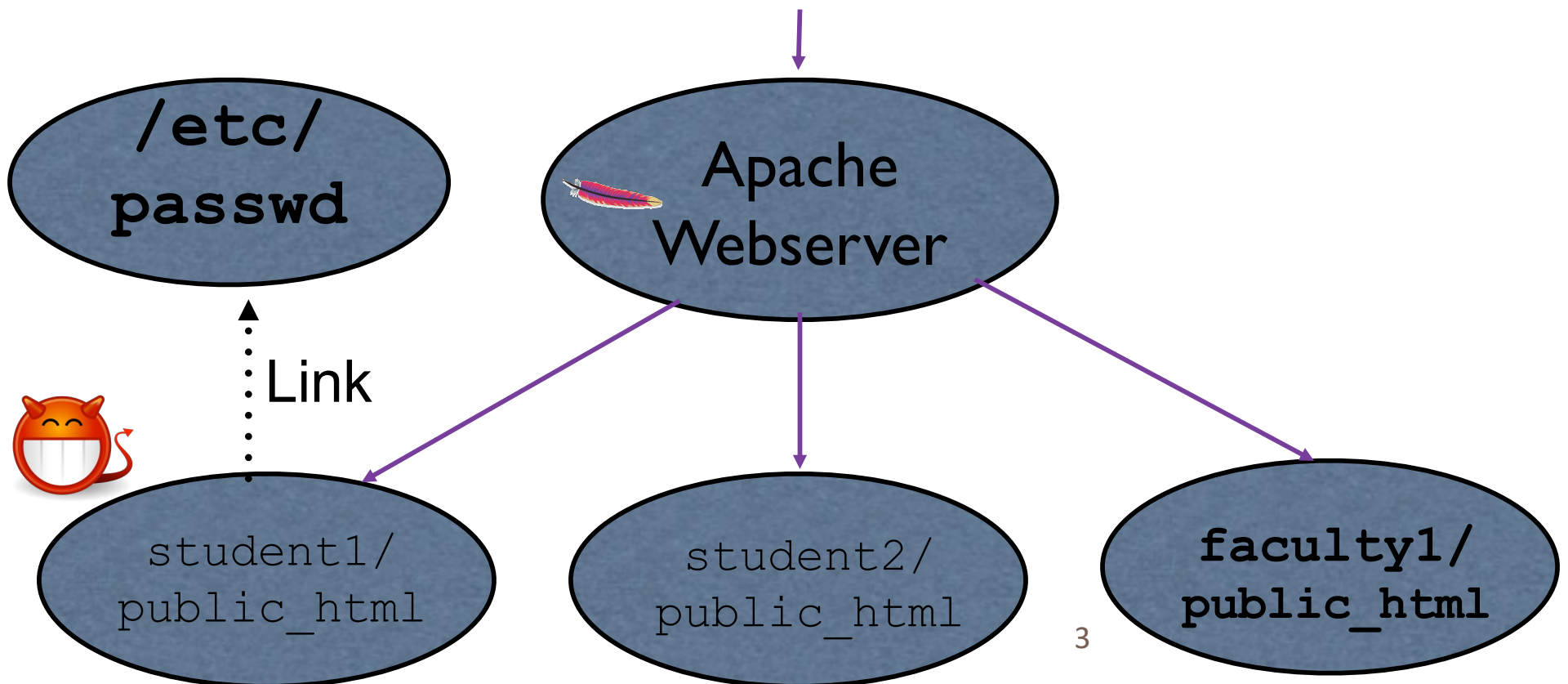
# File Open

- Processes need resources from system
  - ▣ Just a simple `open(filepath, ...)` right?
  - ▣ But, adversaries can cause victims to access resources of their (the adversaries') choosing
  - ▣ And if your program has some valuable privileges, an adversary may want to trick you into using them to implement a malicious operation

# A Webserver's Story ...

- Consider a university department webserver ...

**GET /~student1/index.html HTTP/1.1**



# Links in UNIX

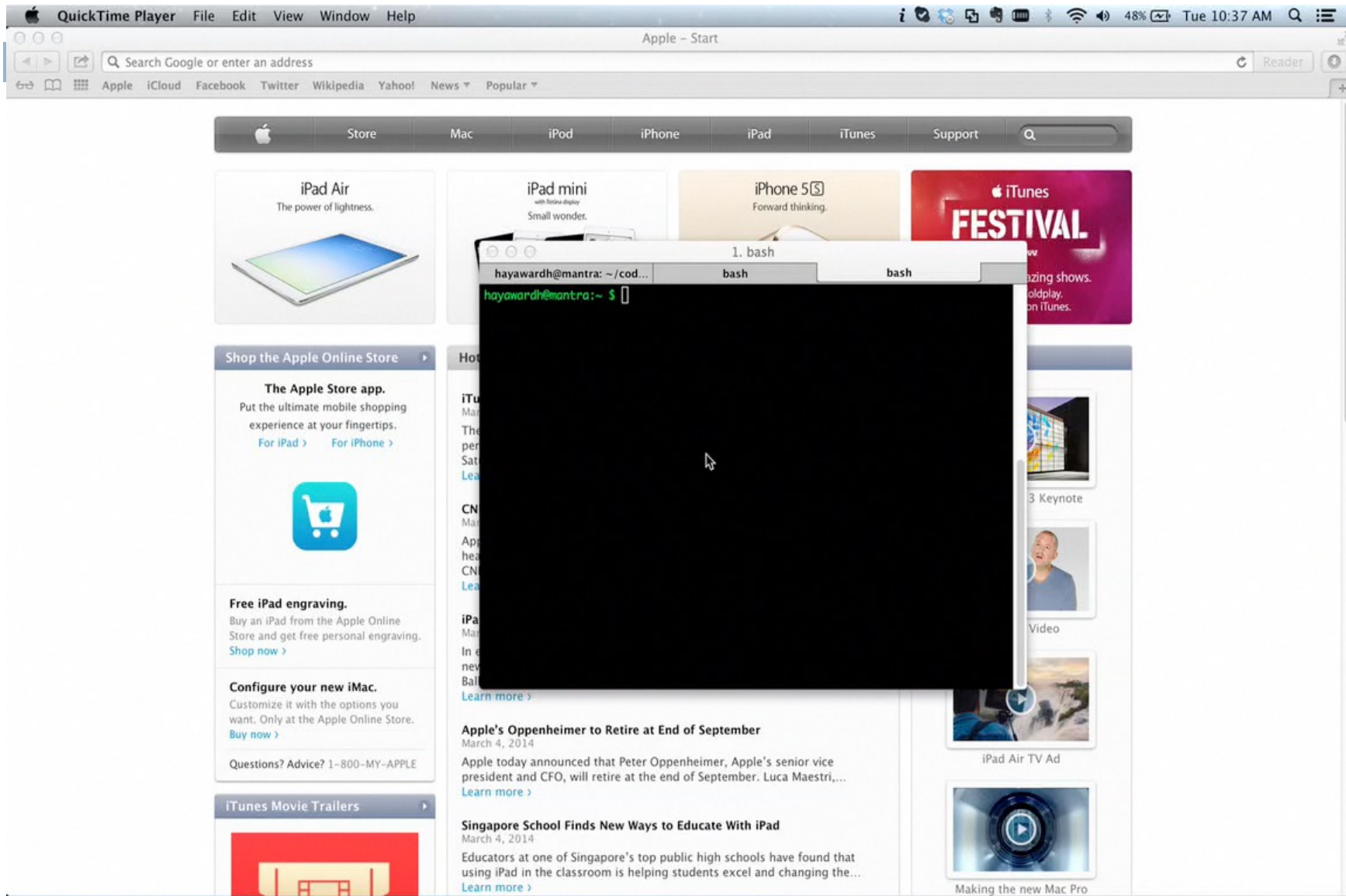


- Makes a file appear in more than 1 directory.
- Is a convenience in several situations.
- 2 types of links:
  - ▣ Symbolic (soft) links
  - ▣ Hard links

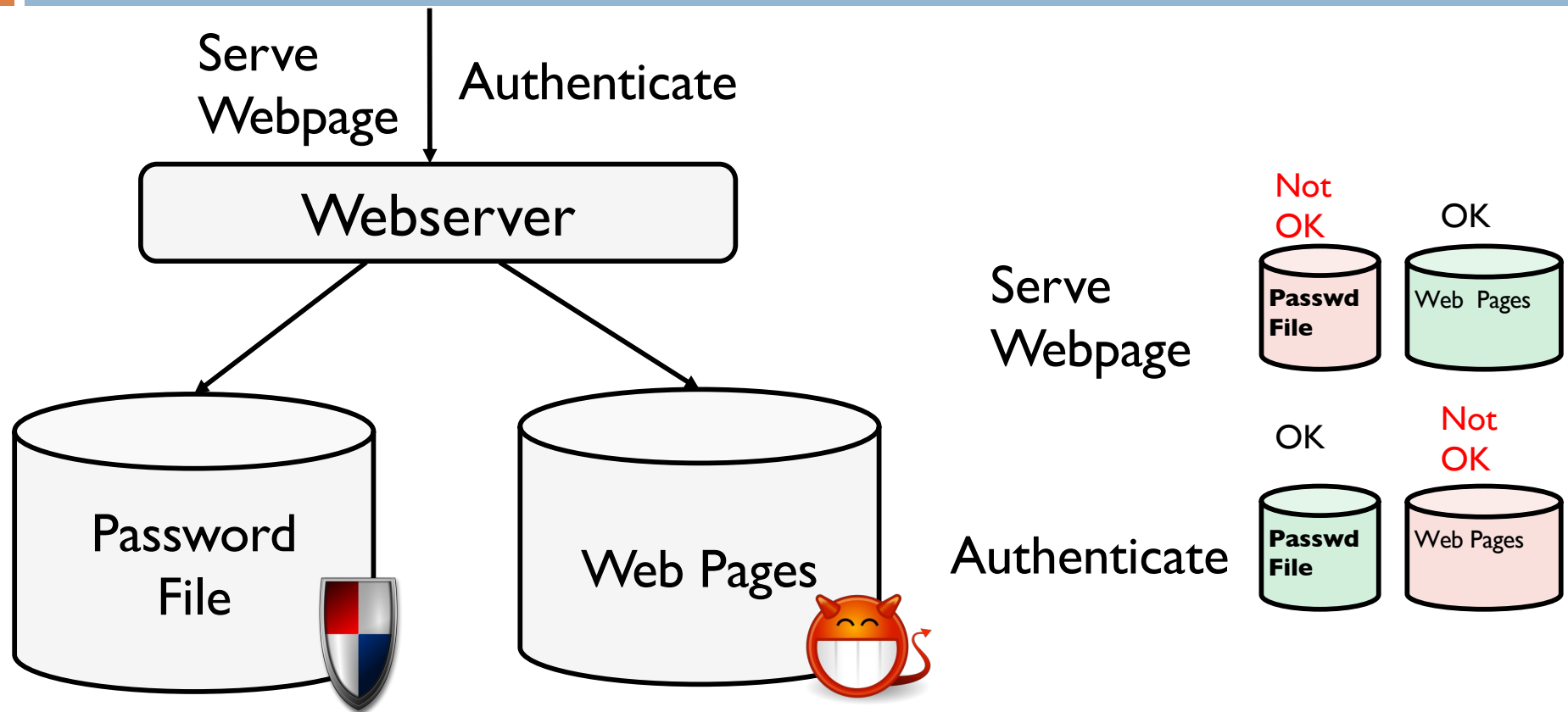
# Symbolic Links

- UNIX supports special files whose values are pathnames to files or directories
  - ▣ In `-s <target-pathname> <link-filename>`
- Suppose you create a symbolic link “foo” in the directory “/home/trj1” with a target pathname of “/etc/passwd”
  - ▣ What will happen?

# Attack Video





# What Just Happened?



□ Program acts as a *confused deputy*

□  when expecting 

□  when expecting 

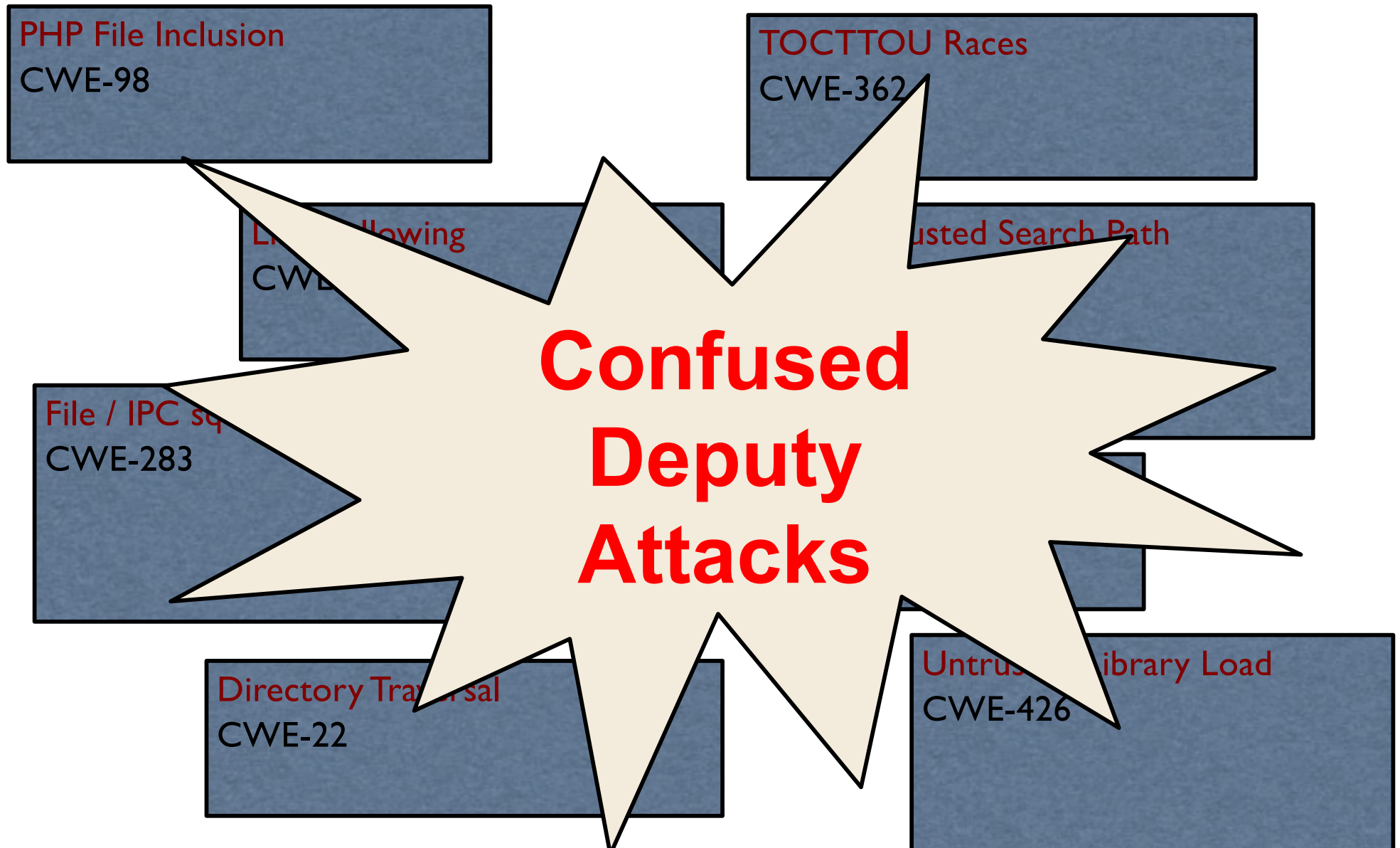
# Integrity (and Secrecy) Threat

- **Confused Deputy**
  - ▶ *Process is tricked into performing an operation on an adversary's behalf that the adversary could not perform on their own*
    - Write to (read from) a privileged file





# Confused Deputy Attacks



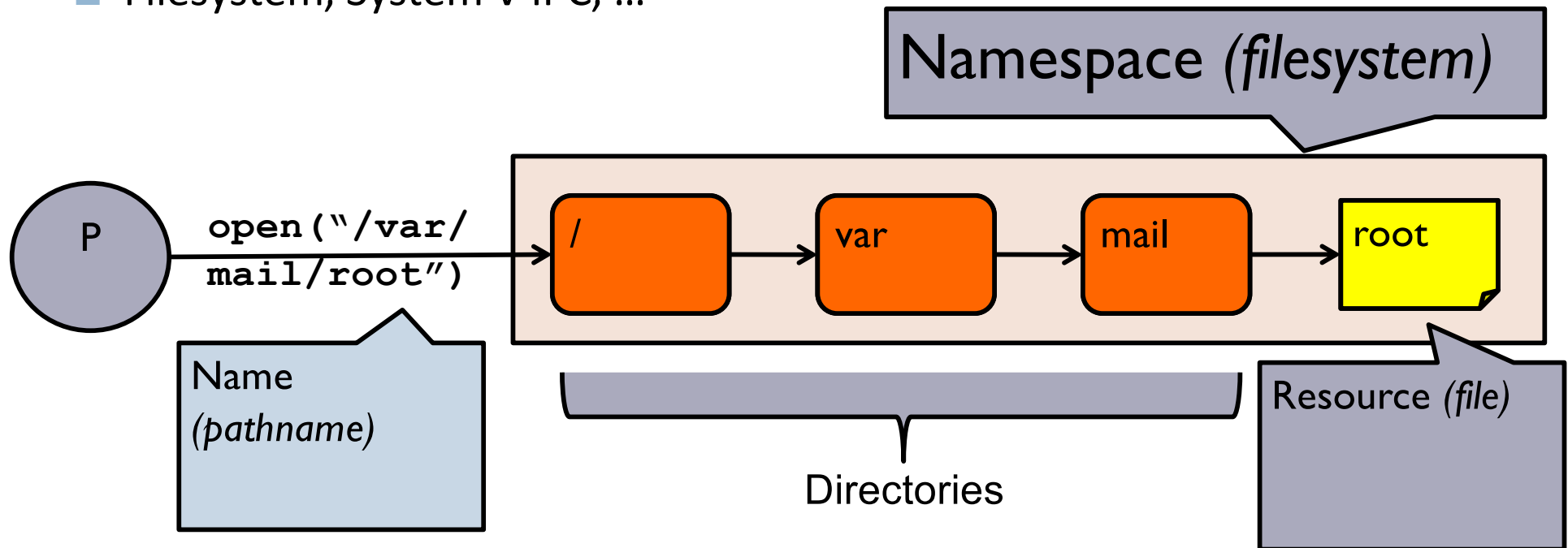
# Lesson



- Opening a file is fraught with danger
  - ▣ We must be careful when using an input that may be adversary controlled when opening a file
    - Or anything else...

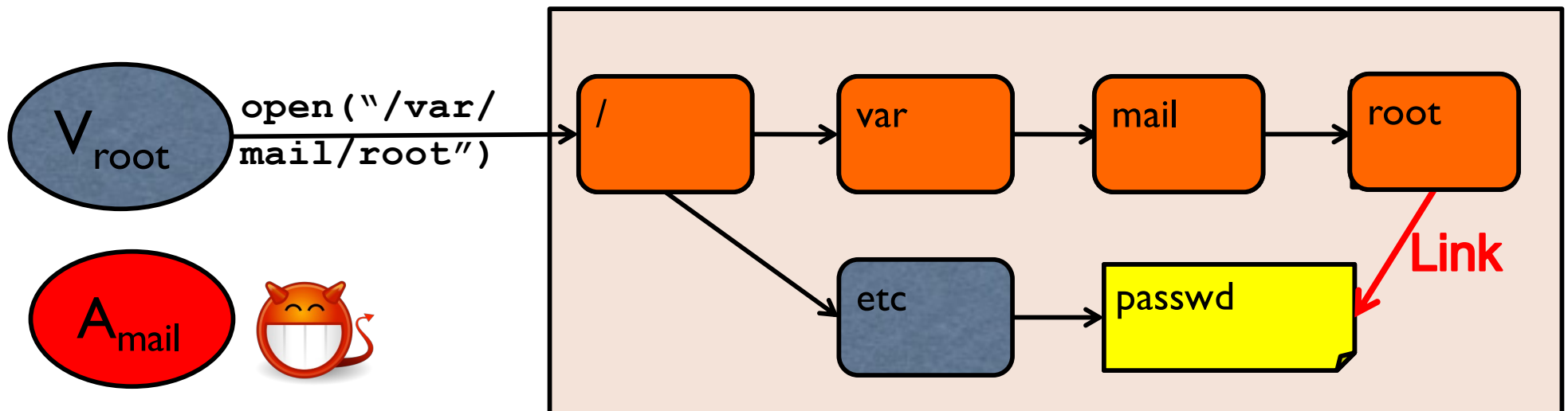
# Name Resolution

- Processes often use *names* to obtain access to *system resources*
- A *nameserver* (e.g., OS) performs *name resolution* using a *namespace* (e.g., *directories*) to convert a *name* (e.g., *pathname*) into a *system resource* (e.g., *file*)
  - Filesystem, System V IPC, ...



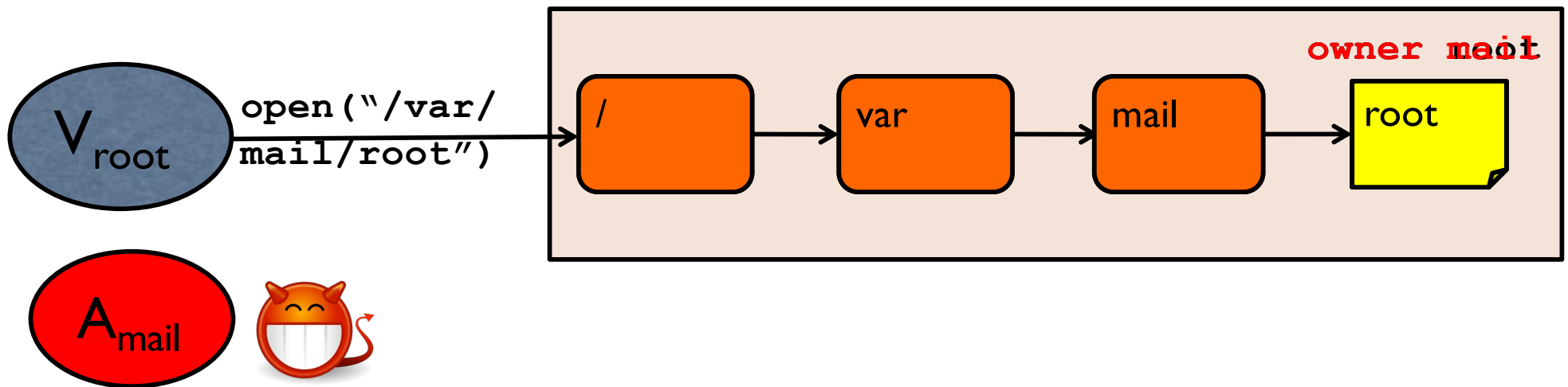
# Link Traversal Attack

- Adversary controls **links** to **direct a victim** to a resource not normally accessible to the adversary
- Victim expects one resource, gets another instead



# File Squatting Attack

- Adversary predicts a resource to be created by a victim – **creates that resource in advance**
- Victim accesses a resource controlled by an adversary instead



# Common Threat

- What is the threat that enables link traversal and file squatting attacks?
  - ▣ Common to both



# Common Threat



- What is the threat that enables link traversal and file squatting attacks?
  - ▣ Common to both
- In both cases, the **adversary has write permission to a directory** that a victim uses in name resolution
  - ▣ Could be any directory used in resolution, not just the last one
  - ▣ Enables the adversary to **plant links** and/or **files/directories** where they can write

# Android Threat Model



- ❑ Executing **untrusted code** on a host system is not ideal...
- ❑ But, that is the **default business model** for mobile phone systems like Android
  - ❑ Called **third-party applications**



# Balance Sharing and Security



## File Sharing

- Sharing media content between social apps
- Document sharing between productivity apps
- File/Data sharing between apps from the same developer



## Security

- Sandboxing through traditional access control (MAC, DAC)
- Fine-grained access control through mechanism like Scoped Storage

# Check and Use

- Some system calls enable checking of the file (**check**)
  - ▣ Does the requesting party have access to the file? (stat, access)
  - ▣ Is the file accessed via a symbolic link? (lstat)
- Some system calls use the file (**use**)
  - ▣ Convert the file name to a file descriptor (open)
  - ▣ Modify the file metadata (chown, chmod)
- Can an adversary modify the filesystem in between **check** and **use** system calls?

# TOCTTOU Races

- Time-of-check-to-time-of-use (TOCTTOU) Race Attacks
- Some system calls enable checking of the file (**check**)
  - ▣ Does the requesting party have access to the file? (stat, access)
  - ▣ Is the file accessed via a symbolic link? (lstat)
- Some system calls use the file (**use**)
  - ▣ Convert the file name to a file descriptor (open)
  - ▣ Modify the file metadata (chown, chmod)
- Can an adversary modify the filesystem in between **check** and **use** system calls? **Yes. Pretty reliably.**

# Current Defenses



- Are there defenses to prevent such attacks?
  - ▣ Yes, but the defenses are not comprehensive

# Defenses

- Variants of the “open” system call
  - ▣ Flag “O\_NOFOLLOW” – do not follow any symbolic links (prevent link traversal)
    - Does not help if you may need to follow symbolic links
    - May not be available on your system
  - ▣ Flag “O\_EXCL” and “O\_CREAT” – do not open unless the new file is created (prevent file squatting)
    - Does not help if you if your program does not know whether the file may need to be created
- These lack flexibility for protection in general

# More Advanced Defenses

## □ The “**openat**” system call

- Can open the directory (**dirfd**) separately from opening the file (**path**) to check the safety of that part of the name resolution

- *int openat(int dirfd, const char \*path, int oflag, ...);*

- Control some aspects of opening “**path**” (e.g., no links)

- E.g., used by libc for opens

```
libc_open (const char *file, int oflag, ...)  
    to
```

```
return SYSCALL_CANCEL (openat, AT_FDCWD, file, oflag, ...);
```

## □ The “**openat2**” system call

- More flags limiting “how” name resolution is done for “path”
- Not standard

# Open\_No\_Symlink Defense

- **Goal:** Ensure target file is not a symlink
- Check for symbolic link (**lstat**)
- Check for lstat-open race
- Check for inode recycling
- Do checks for each path component
  - ▣ /, var, mail, ...
- What if you want to use symlinks – just safely?

```
/* fail if file is a symbolic link */
int open_no_symlink(char *fname)
{
01  struct stat lbuf, buf;
02  int fd = 0;
03  lstat(fname, &lbuf);
04  if (S_ISLNK(lbuf.st_mode))
05    error("File is a symbolic link!");
06  fd = open(fname);
07  fstat(fd, &buf);
08  if ((buf.st_dev != lbuf.st_dev) ||
09      (buf.st_ino != lbuf.st_ino))
10    error("Race detected!");
11  lstat(fname, &lbuf);
12  if ((buf.st_dev != lbuf.st_dev) ||
13      (buf.st_ino != lbuf.st_ino))
14    error("Cryogenic sleep race!");
15  return fd;
}
```

# Find Filesystem Vulnerabilities



- How do we detect when
  - ▣ One of these filesystem attacks is possible?
  - ▣ And whether the program is vulnerable?



# Find Filesystem Vulnerabilities



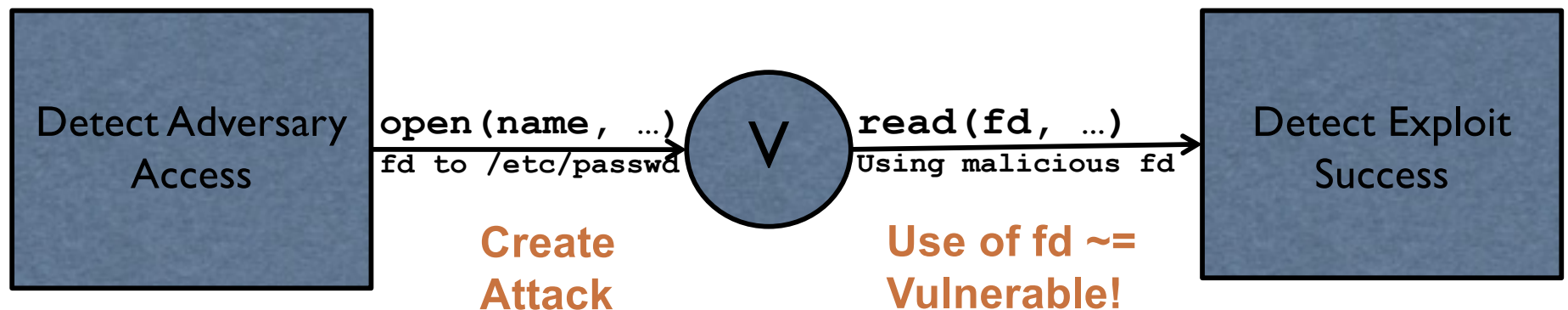
- How do we detect when
  - ▣ One of these filesystem attacks is possible?
    - Accessible to adversary-chosen action
      - Squatted file
      - Link

# Find Filesystem Vulnerabilities

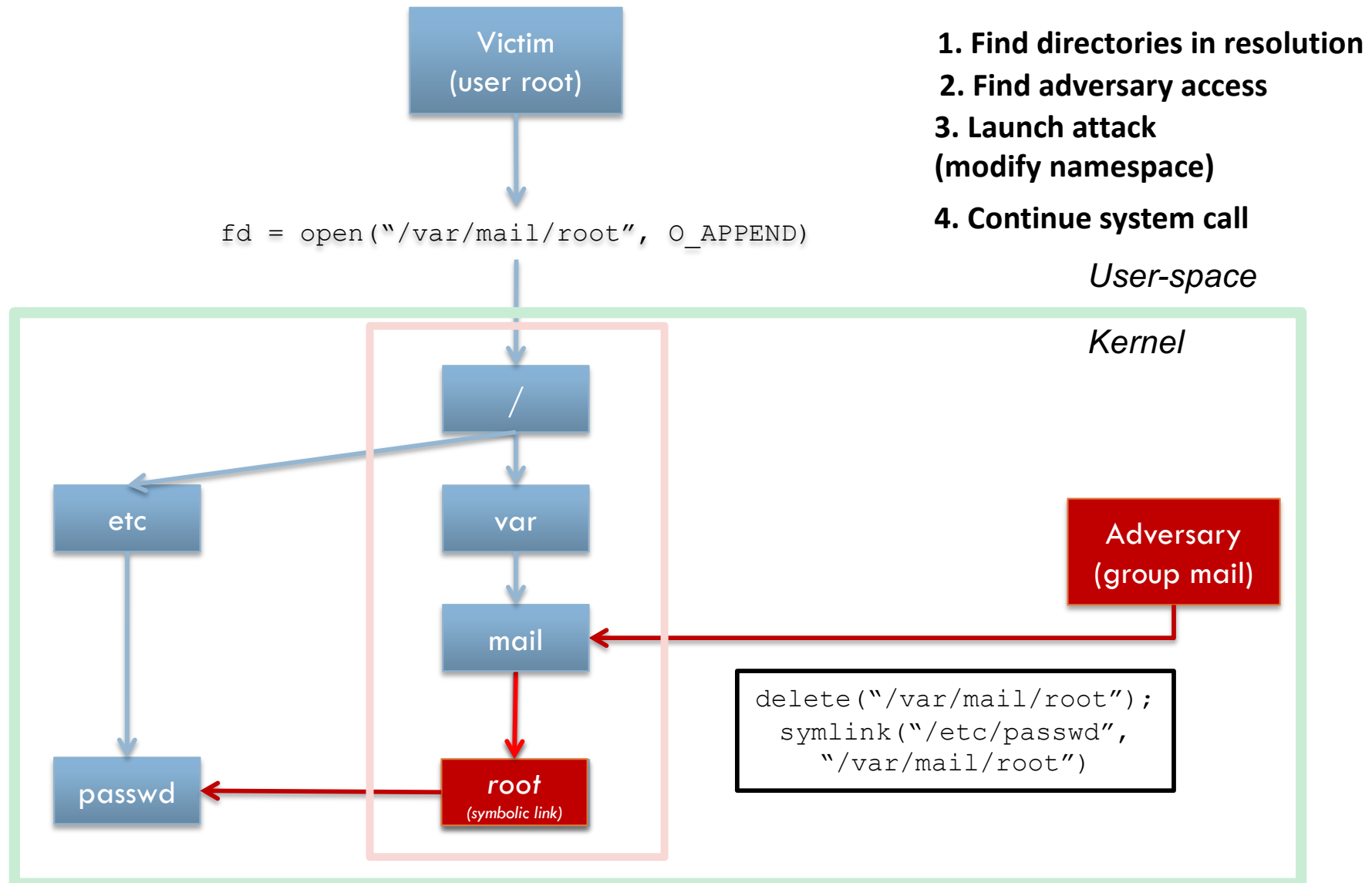
- How do we detect when
  - ▣ One of these filesystem attacks is possible?
    - Accessible to adversary-chosen action
      - Squatted file
      - Link
  - ▣ And whether the program is vulnerable?
    - What makes the program vulnerable?
      - Use of the squatted file
      - Use of the target of the adversary-chosen link

# Dynamic Testing [STING]

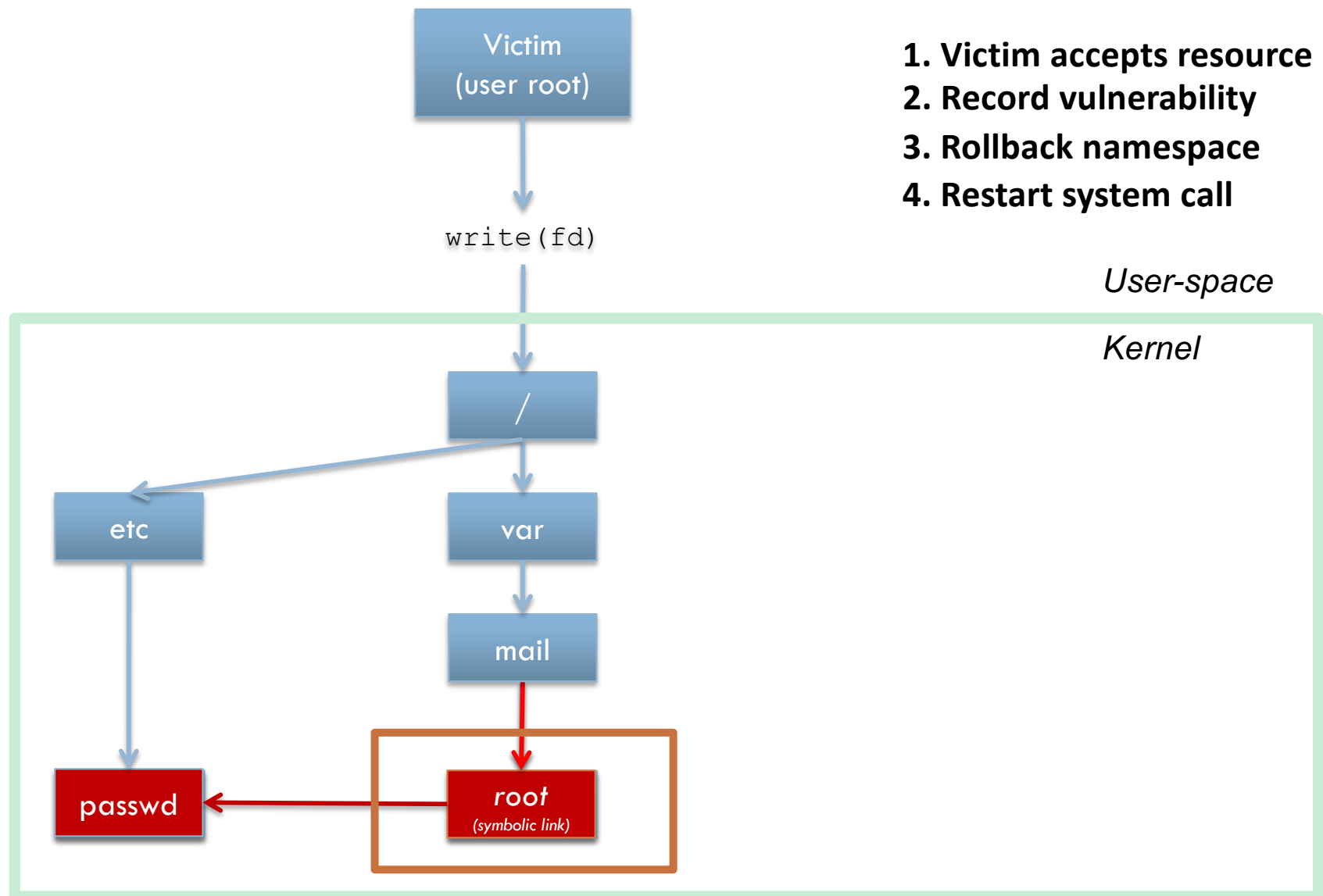
- We actively change the namespace whenever an adversary can write to a directory that is actually used in a name resolution
  - ▣ **Fundamental problem:** adversaries may be able to write directories used in name resolution



# STING Launch Phase



# STING Detect Phase



# STING Detects TOCTTOU Races

- ❑ STING can deterministically create races, as it is in the OS

Victim

Adversary

```
SOCKET_DIR=/tmp/.X11-unix

set_up_socket_dir () {
  if [ "$VERBOSE" != no ]; then
    log_begin_msg "Setting up $SOCKET_DIR..."
  fi
  if [ -e $SOCKET_DIR ] && [ ! -d $SOCKET_DIR ]; then
    mv $SOCKET_DIR $SOCKET_DIR.$$
  fi
  mkdir -p $SOCKET_DIR
  chown root:root $SOCKET_DIR
  chmod 1777 $SOCKET_DIR
  do_restorecon $SOCKET_DIR
  [ "$VERBOSE" != no ] && log_end_msg 0 || return 0
}
```

```
ln -s /etc/passwd
    /tmp/.X11-unix
```

# Results – Vulnerabilities - 2012

Program	Vuln. Entry	Priv. Escalation DAC: uid->uid	Distribution	Previously known
dbus-daemon	2	messagebus->root	Ubuntu	Unknown
landscape	4	landscape->root	Ubuntu	Unknown
Startup scripts (3)	4	various->root	Ubuntu	Unknown
mysql	2	mysql->root	Ubuntu	1 Known
mysql_upgrade	1	mysql->root	Ubuntu	Unknown
tomcat script	2	tomcat6->root	Ubuntu	Known
lightdm	1	*->root	Ubuntu	Unknown
bluetooth-applet	1	*->user	Ubuntu	Unknown
java (openjdk)	1	*->user	Both	Known
zeitgeist-daemon	1	*->user	Both	Unknown
mountall	1	*->root	Ubuntu	Unknown
mailutils	1	mail->root	Ubuntu	Unknown
bsd-mailx	1	mail->root	Fedora	Unknown
cupsd	1	cups->root	Fedora	Known
abrt-server	1	abrt->root	Fedora	Unknown
yum	1	sync->root	Fedora	Unknown
x2gostartagent	1	*->user	Extra	Unknown
<b>19 Programs</b>	<b>26</b>			<b>21 Unknown</b>

Both old and new programs

Special users to root

Known but unfixed!

# Conclusions

64

- Adversaries can attack your use of the filesystem
- **Local exploit** on shared access to the filesystem that your program may use in **name resolution**
  - ▣ If an adversary has **write permission to any directory used** in name resolution
    - **File squatting** can control file content used by your program
    - **Link traversal** can redirect your program to other files
- Turns out to be a difficult problem to prevent in shared filesystems (Android)
  - ▣ **Openat** is the most secure option, but not perfect



# Questions

65

