CS165 – Computer Security

Filesystem Security

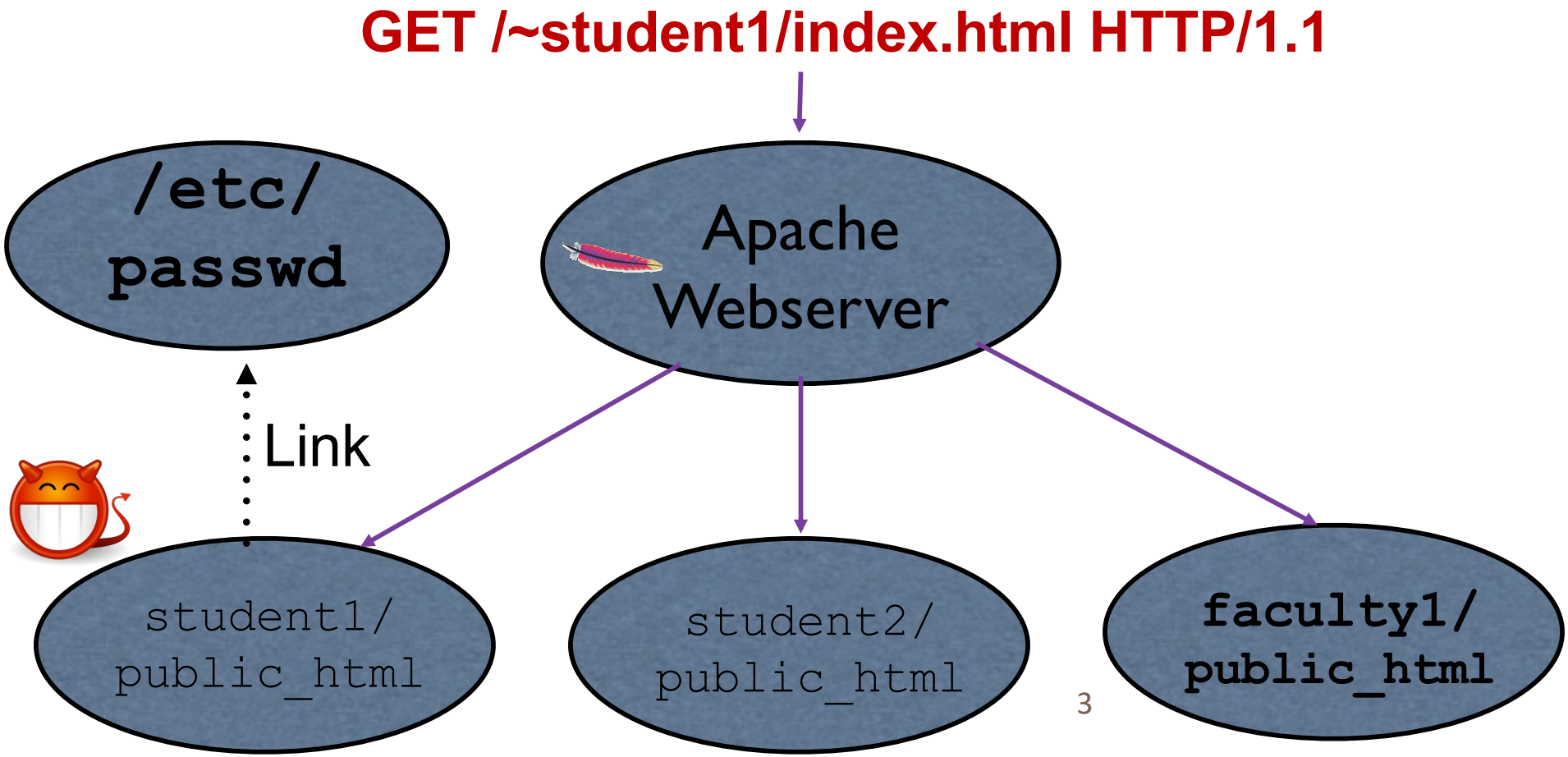
March 5, 2024

File Open

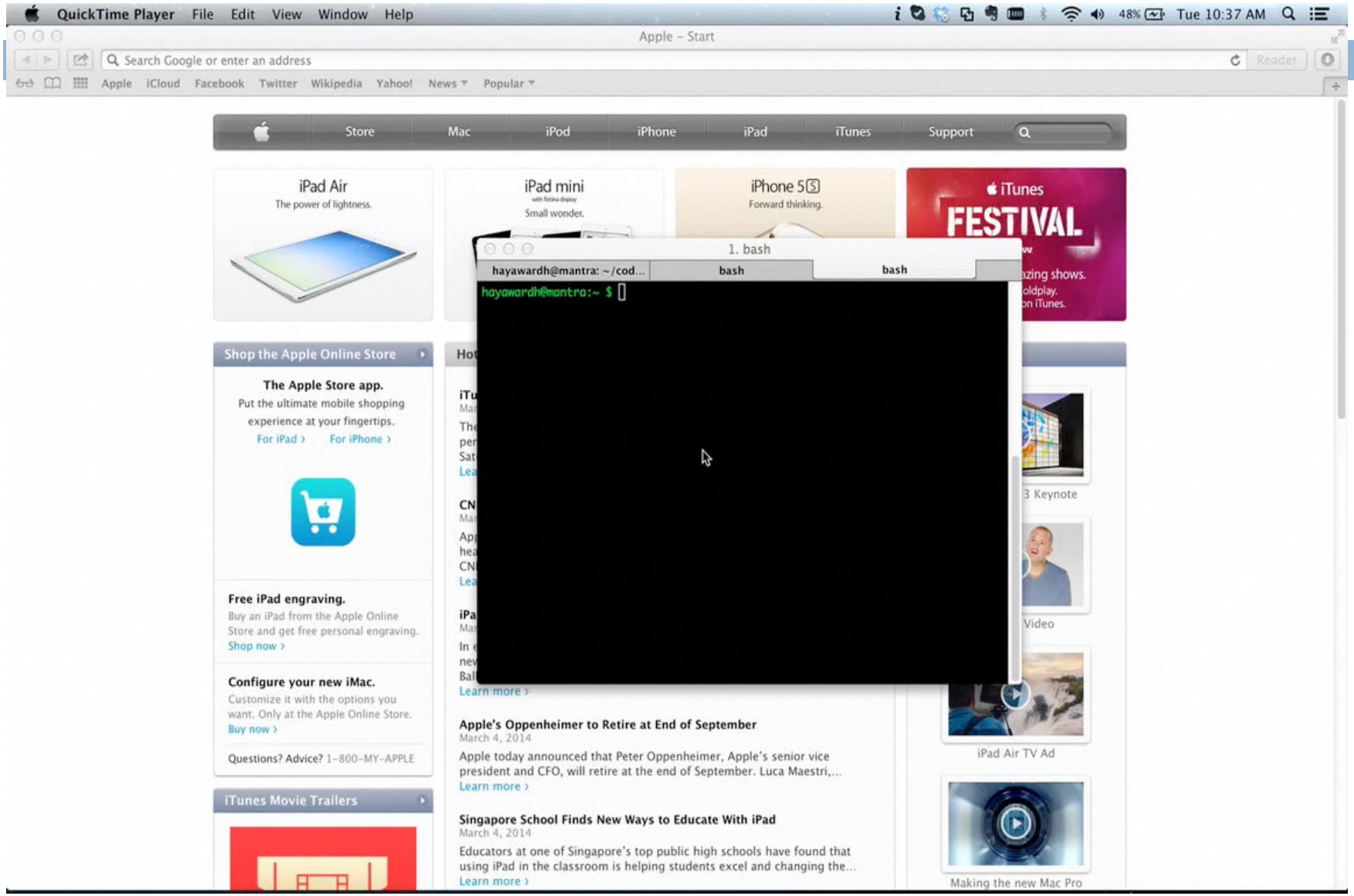
- Problem: Processes need resources from system
  - Just a simple `open(filepath, …)` right?
  - But, adversaries can cause victims to access resources of their choosing
  - And if your program has some valuable privileges, an adversary may want to trick you into using them to implement a malicious operation

A Webserver's Story …

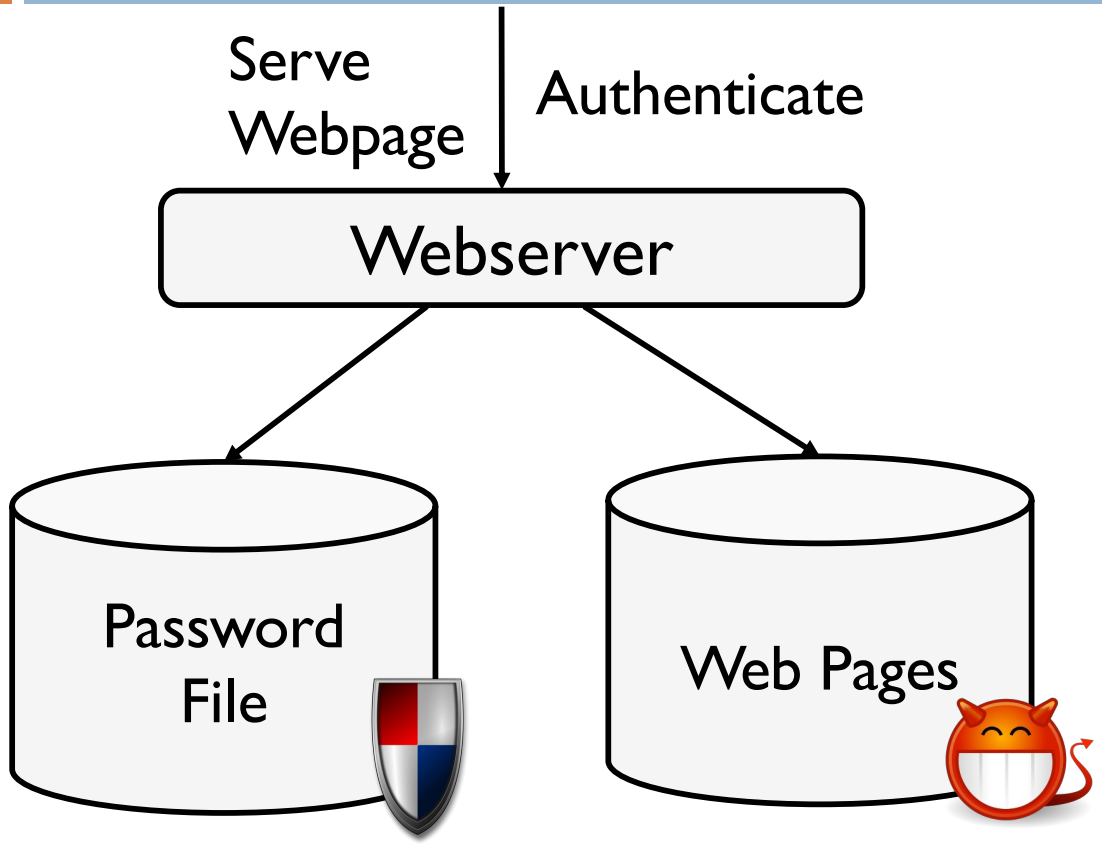
- Consider a university department webserver …

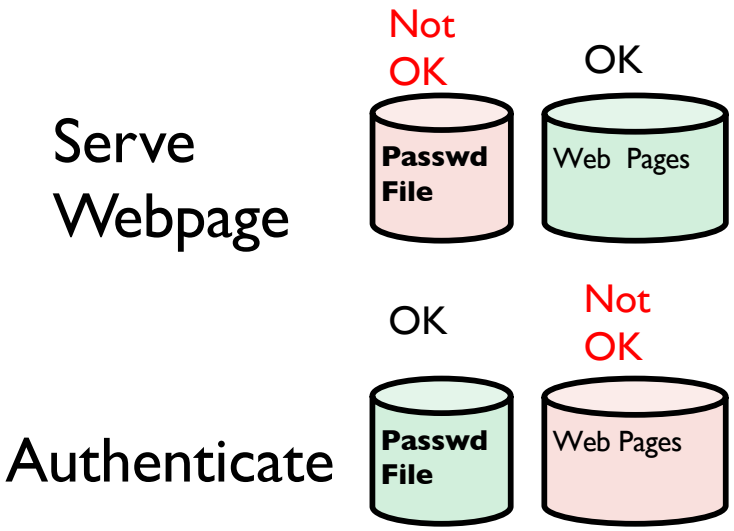


Attack Video



What Just Happened?



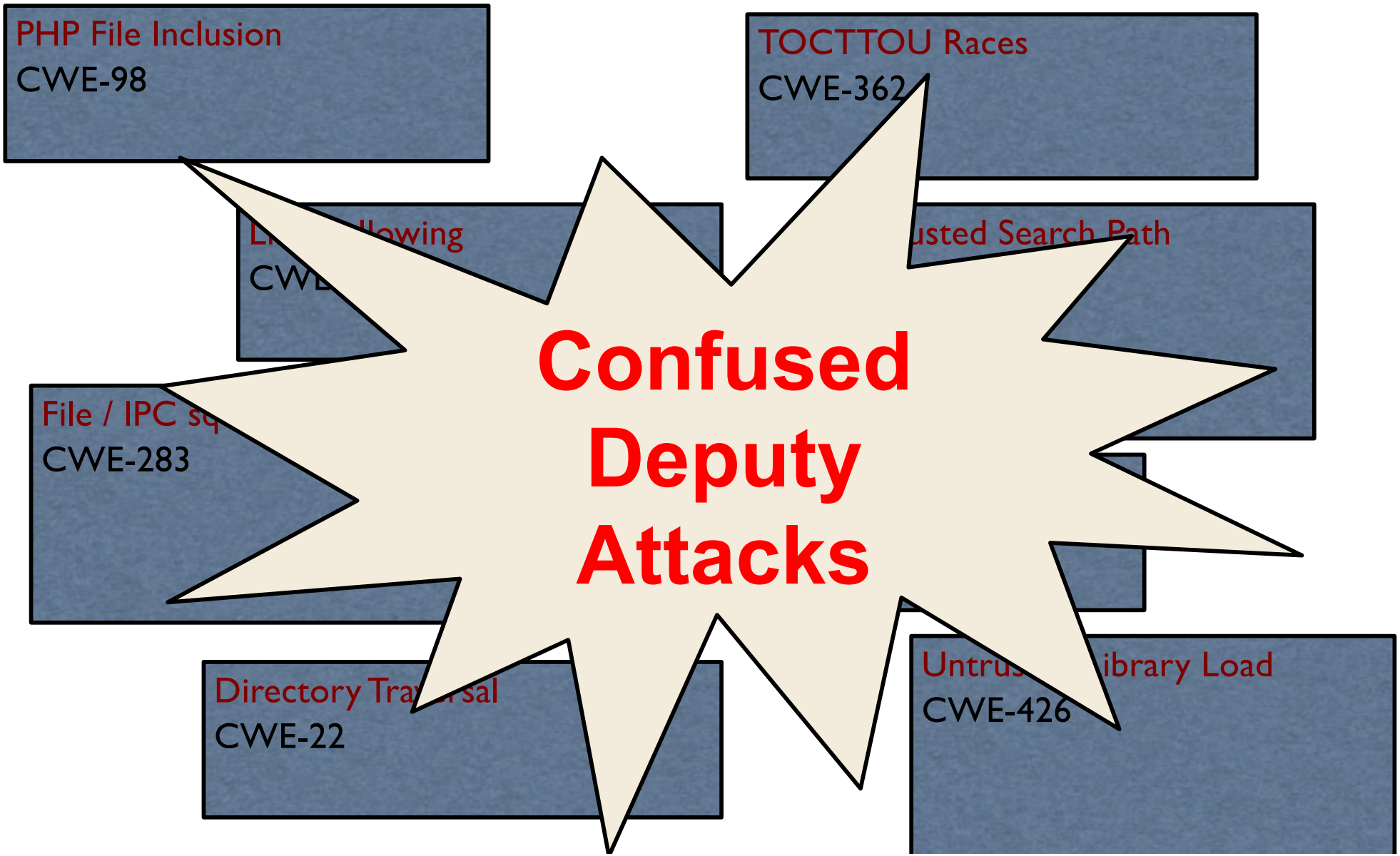


- Program acts as a *confused deputy*
  -  when expecting 
  -  when expecting 

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

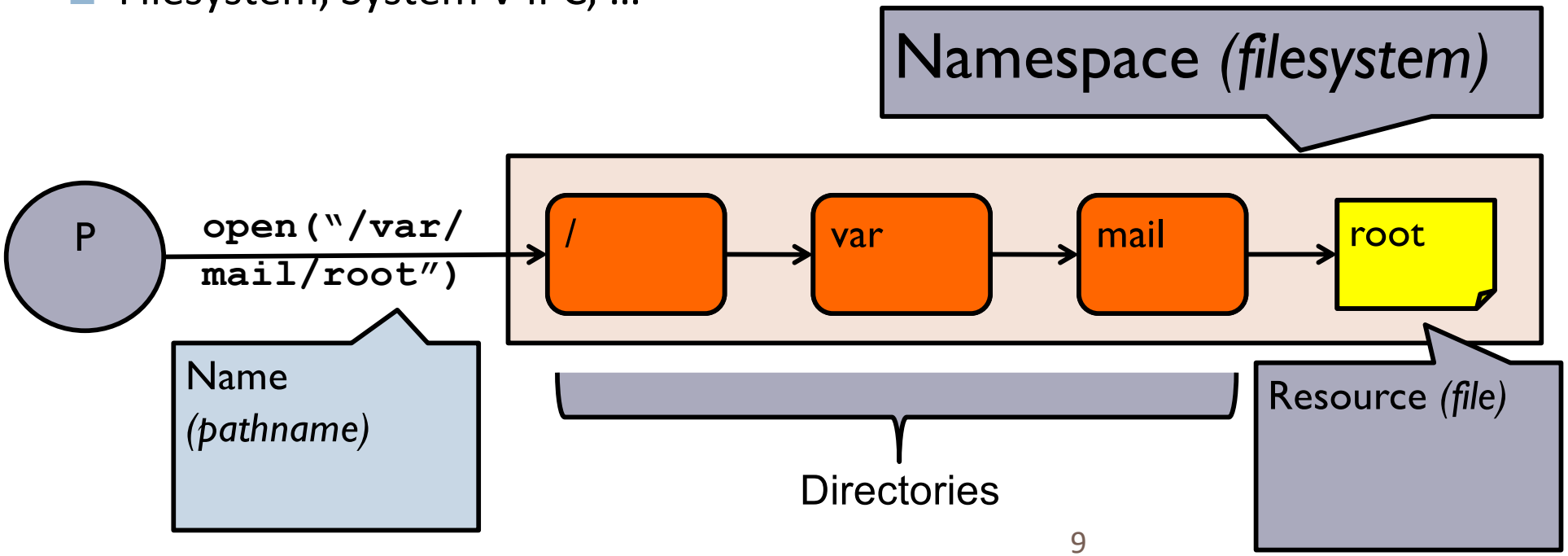


Lesson

- 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…

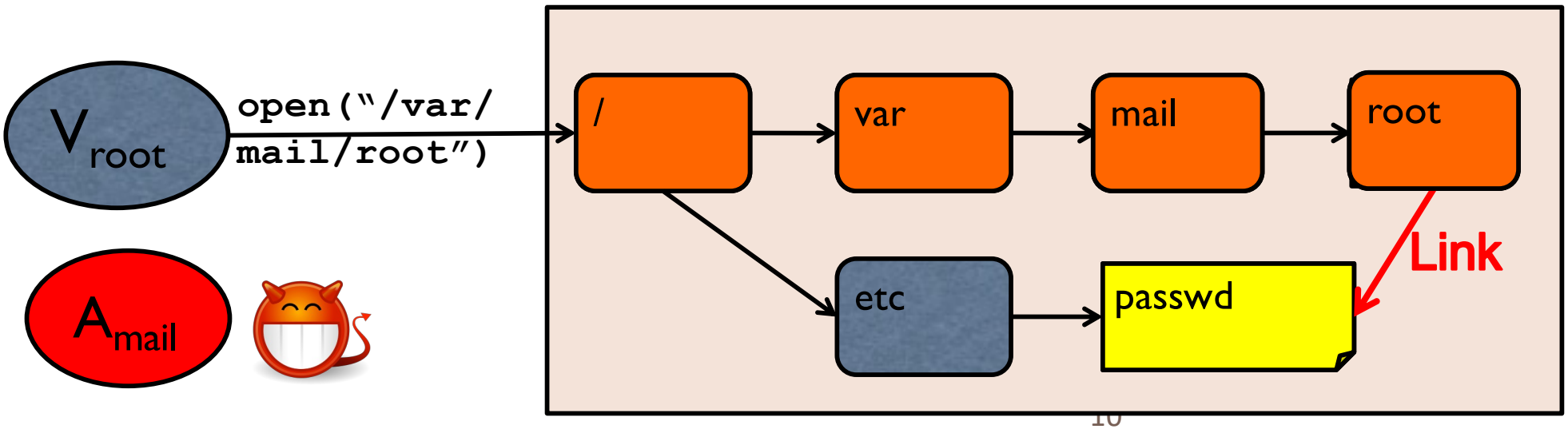
Name Resolution

- 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, …



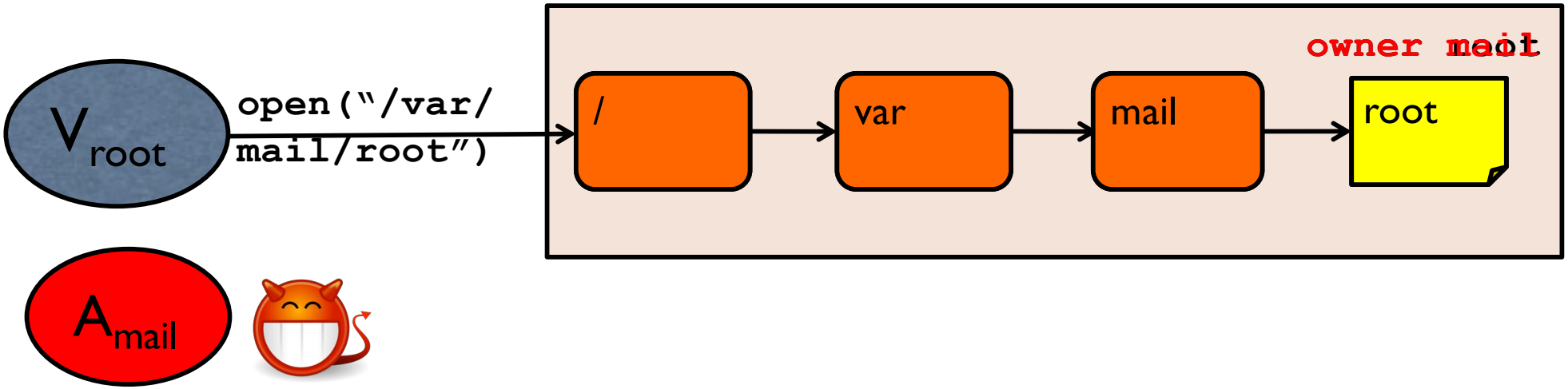
Link Traversal Attack

- 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

Threat Example

- An adversary may be authorized to write to a directory you use in resolving a file path
- E.g., groups and others may have write permission to a directory
  - Consider the directory /tmp
  - `ls -la /tmp`
    - `drwxrwxrwx --- root root ---  .`
    - Means?

Threat Example

- Suppose your program wants to create a new file at “/tmp/just_a_normal_file_here”
  - What file will you create/open?

File Squatting

- Suppose your program wants to create a new file at “/tmp/just_a_normal_file_here”
  - What file will you open?
    - An adversary could have created this file already (file squat) and given you permissions, so that you can use it
      - Can be difficult to verify the origins of a file
  - Causes your program to use a file under adversary control when you expect your own file

Threat Example

- Suppose your program is asked to open the file path "/tmp/just_a_normal_file_here"
  - What file will you open?

Link Traversal

- Suppose your program is asked to open the file path "/tmp/just_a_normal_file_here"
  - What file will you open?
    - An adversary could have created this as a symbolic link to any file in the system that you can access
    - And it is difficult/expensive to verify that this is not a symbolic link
      - stat – provides file system information – e.g., permissions
      - lstat – provides file system information (like "stat") for the link, rather than the file/directory the link refers to
  - Causes your program to access an adversary-chosen file

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 in libc

```
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

Openat Usage Example

- Suppose you want to open "/var/mail/root" safely with "openat"
  - How would you do it?

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

- Three steps
  - (1) Open "/var/mail" to obtain a "dirfd"
  - (2) Validate that the resulting file descriptor refers to "/var/mail"
  - (3) Open the file "root" using "openat" using options to protect the open from attacks
    - O_NOFOLLOW to prevent use of symbolic links (i.e., prevent link traversal)
    - O_EXCL with O_CREAT to ensure a fresh file is created (i.e., to prevent file squatting)
- Two options for obtaining a valid "dirfd" value for "/var/mail"
  - (a) If you can run the program from "/var/mail/" then you can use `AT_FDCWD` for "dirfd" – guaranteed by the OS
    - `openat(AT_FDCWD, "root" , O_NOFOLLOW | O_EXCL | O_CREAT);`
  - (b) Open and validate "/var/mail/" yourself and then use as the "dirfd"
    - `openat(dirfd, "root" , O_NOFOLLOW | O_EXCL | O_CREAT);`

Validating directories

- How do you validate a directory for “dirfd”?
- Three steps
  - (1) Open “/var” to obtain its “fd”
  - (2) Collect the “stat” structure for this “fd”
    - From the file descriptor using fstat

```
int fstat(int fd, struct stat *buf);
```

  - (3) Check that this “fd” refers to a directory

```
S_ISDIR(mode_t buf.st_mode);  // see ”struct stat” format
```

  - (4) Repeat (1-3) for “mail” opened from this “fd” (i.e., “/var”)

```
int openat(int fd, const char “mail”, int oflag, ...);
```

Vulnerabilities Easily Overlooked

- Manual checks can easily overlook vulnerabilities
- Misses file squat at line 03!

```
01 /* filename = /var/mail/root */
02 /* First, check if file already exists */
03 fd = open (filename, flg);
04 if (fd == -1) {
05     /* Create the file */
06     fd = open(filename, O_CREAT|O_EXCL);
07     if (fd < 0) {
08         return errno;
09     }
10 }
11 /* We now have a file. Make sure
12 we did not open a symlink. */
13 struct stat fdbuf, filebuf;
14 if (fstat (fd, &fdbuf) == -1)
15     return errno;
16 if (lstat (filename, &filebuf) == -1)
17     return errno;
18 /* Now check if file and fd reference the same file,
19    file only has one link, file is plain file.  */
20 if ((fdbuf.st_dev != filebuf.st_dev
21     || fdbuf.st_ino != filebuf.st_ino
22     || fdbuf.st_nlink != 1
23     || filebuf.st_nlink != 1
24     || (fdbuf.st_mode & S_IFMT) != S_IFREG)) {
25     error (_("%s must be a plain file
26         with one link"), filename);
27     close (fd);
28     return EINVAL;
29 }
30 /* If we get here, all checks passed.
31    Start using the file */
32 read(fd, ...)
```

Squat during create (resource)

Symbolic link

Hard link, race conditions

Find Filesystem Vulnerabilities

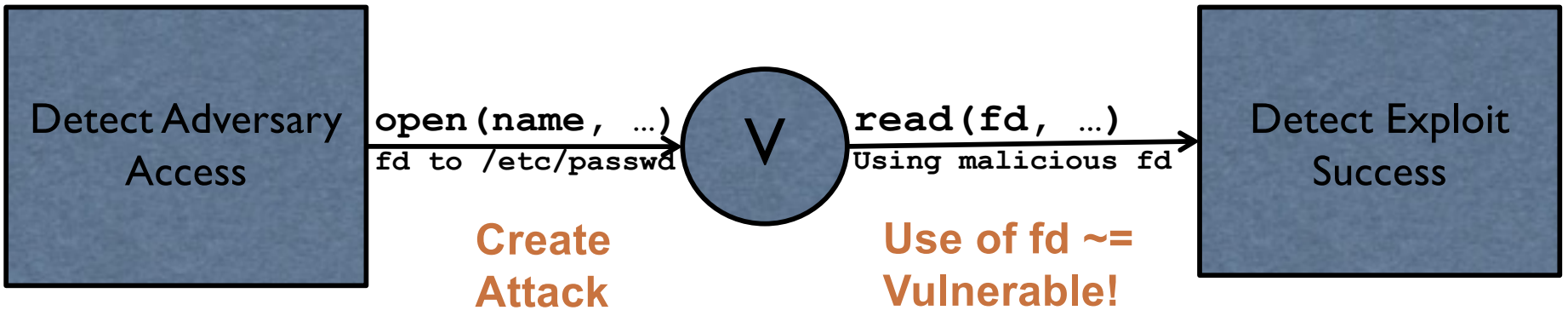
- How do we detect when
  - One of these filesystem attacks is possible?
  - And whether the program is vulnerable?

Find Filesystem Vulnerabilities

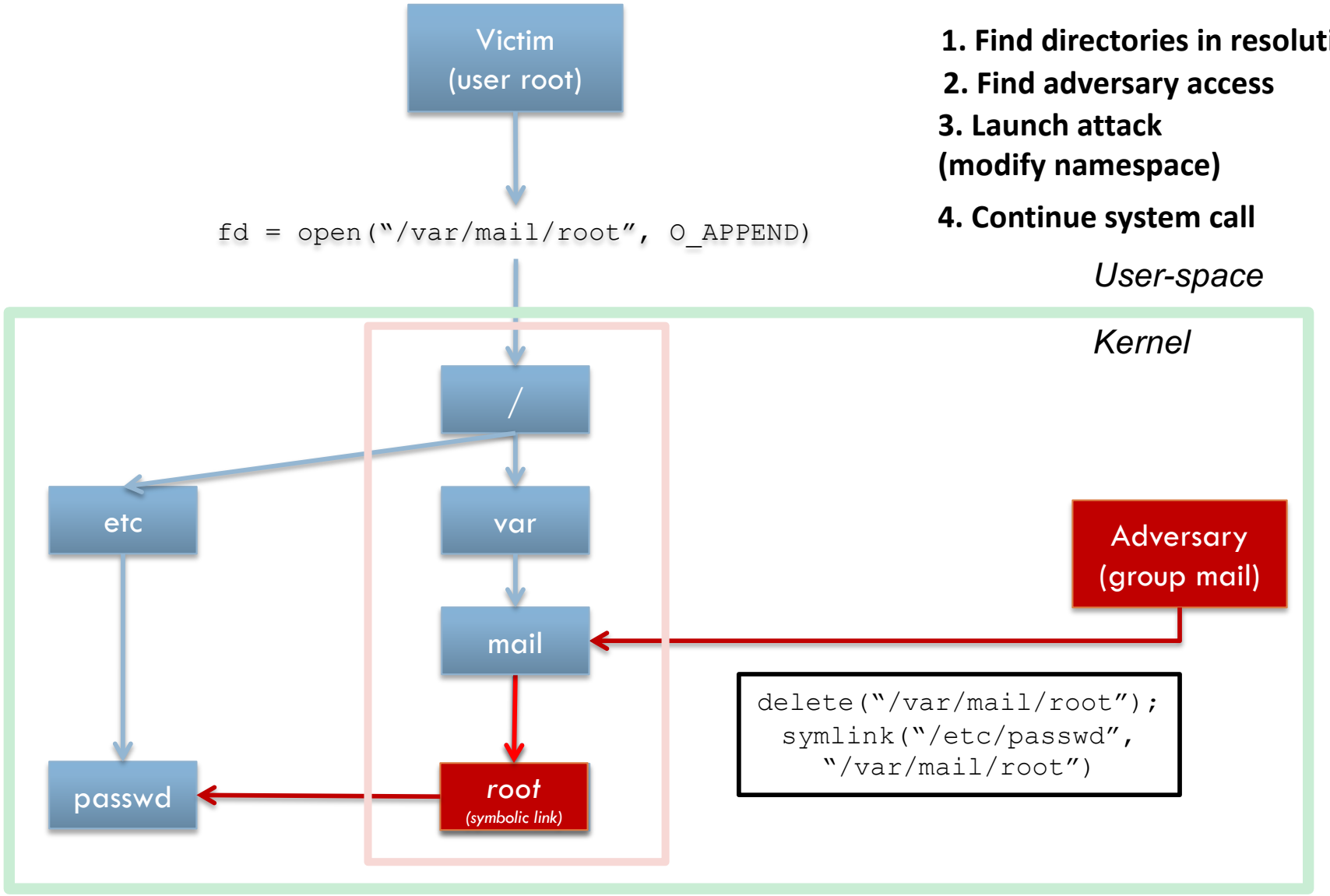
- How do we detect when
  - One of these filesystem attacks is possible?
    - Accessible
  - And whether the program is vulnerable?
    - Flaw that is exploitable

Dynamic Testing [STING]

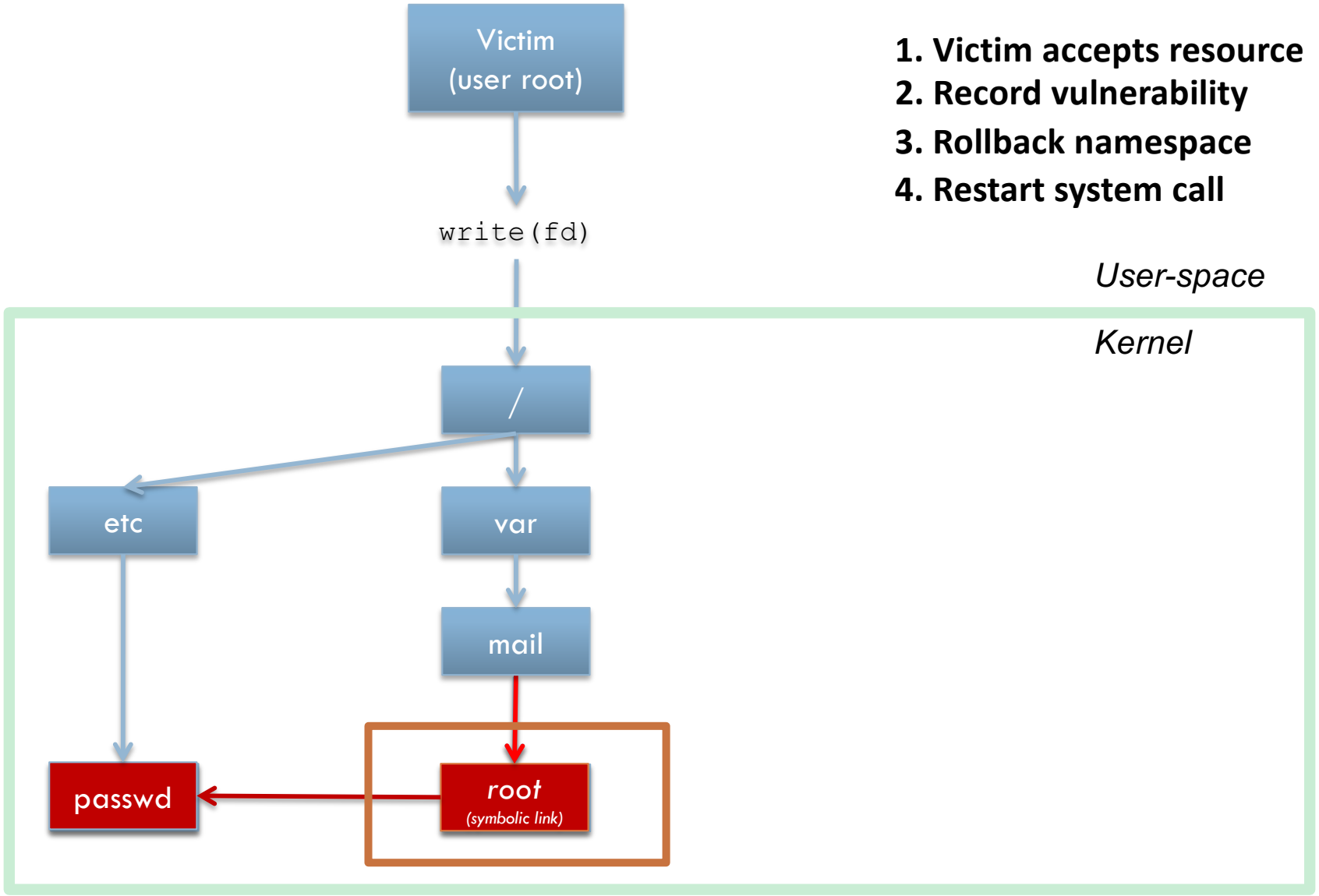
- We actively change the namespace whenever an adversary can write to a directory that is actually used in a name resolution
  - Fundamental problem: adversaries may be able to write directories used in name resolution



STING Launch Phase



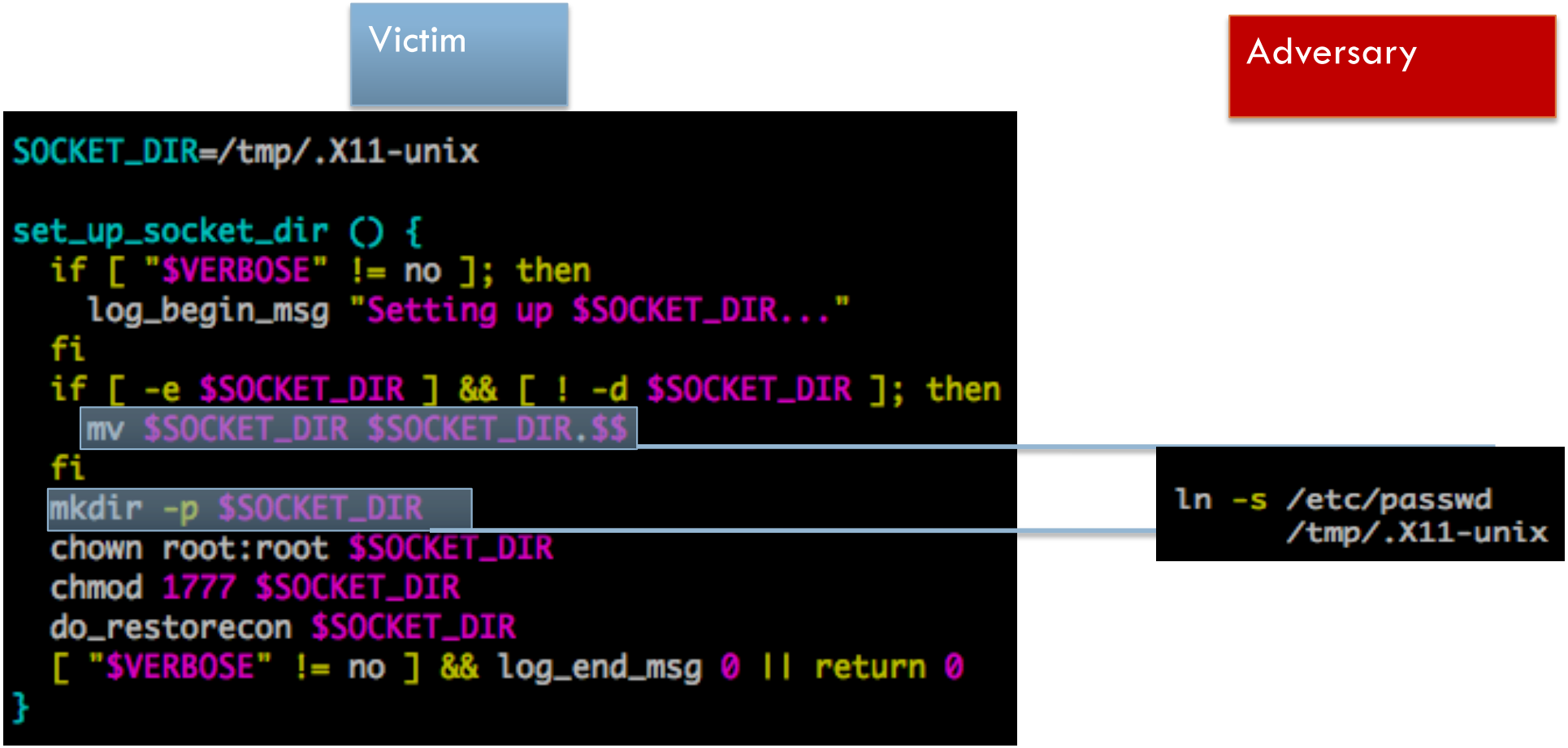
STING Detect Phase



1. **Victim accepts resource**
2. **Record vulnerability**
3. **Rollback namespace**
4. **Restart system call**

STING Detects TOCTTOU Races

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



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
19 Programs	**26**			**21 Unknown**

Both old and new programs

Special users to root

Known but unfixed!

Results – Vulnerabilities - 2024

TABLE II: **Part of Real-world FHVulns Detected by JERRY and Confirmed by Developers.** The abbreviations Ins, Uni, Up, Rep, SU and Us represent Installation, Uninstallation, Updating, Repairing, Starting Up and Usage, respectively. The abbreviations PC, IL, RD, CT, MV and DT represent Process Creation, Image Loading, Reading, Creating, Moving and Deleting, respectively. The Symbol "★" indicates that the corresponding software is pre-installed.

No.	Software Name	# Download	Stage	Operation	Status
1	Adobe Reader DC	465,124,436	Ins	CT	Confirmed
2	Adobe Reader DC	465,124,436	Uni	DT	Confirmed
3	Chrome	97,544,900	Ins	CT	CVE-2023-2939
4	Chrome	97,544,900	Ins	RD	Fixed
5	Firefox	40,111,618	Uni	DT	CVE-2023-4052
6	JRE8	24,394,580	Ins	CT	Fixed
7	Visual Studio	10,670,579	Ins	CT	CVE-2023-21567
8	Visual Studio	10,670,579	Us	PC	Confirmed
9	Git for Windows	10,256,420	Ins	PC	CVE-2022-31012
10	Git for Windows	10,256,420	SU	RD	CVE-2022-24765
11	Git for Windows	10,256,420	Us	PC	CVE-2022-41953
12	Git for Windows	10,256,420	Us	PC	CVE-2023-23618
13	Git for Windows	10,256,420	SU	PC	CVE-2023-29012
14	Git for Windows	10,256,420	SU	RD	CVE-2023-29011
15	Openssh for Windows	5,884,392	SU	RD	CVE-2022-26558
16	Sysinternals	5,859,086	SU	IL	Confirmed
17	Nodejs	5,353,689	SU	RD	Confirmed

339 new vulnerabilities detected!

Local Exploits

- Attacks on filesystems, such as link traversal and file squatting often require that an adversary already controls code running on the host
  - Often called "local exploits"
- Can be achieved by downloading malware or hijacking a running process
  - So, defenders are often less concerned about these attacks, although these are often used
- But, in some systems, local exploits are a first-level issue

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

Find Where Attacks Are Possible

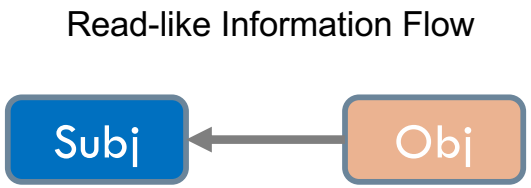
- How can we find where attacks may be possible?

Find Where Attacks Are Possible

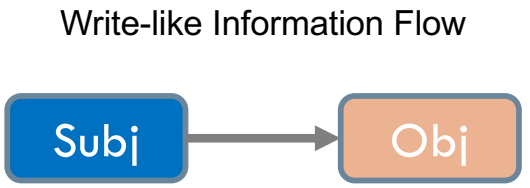
- How can we find where attacks may be possible?
  - Use information flow
- Question: Can an adversary of a victim process write to a directory used in name resolution (i.e., is readable) by the victim?

Access Control Policy Analysis

- Access control policies determine what files and directories can be read and written by each subject

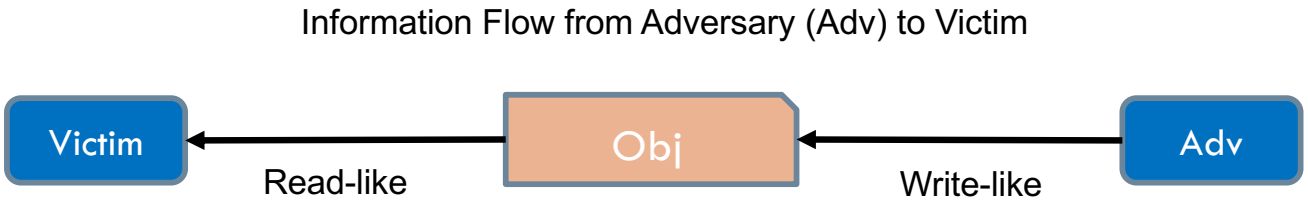


Subject can read Object



Subject can write object

- Look for cases where an adversary subject can write a directory that can be read by a victim



Who's An Adversary?

- Good question
  - Every other program? May trust some…
  - Only known untrusted? How do you know?
- Hard to get perfect, but many programs need not be trusted
  - In case they become adversarial

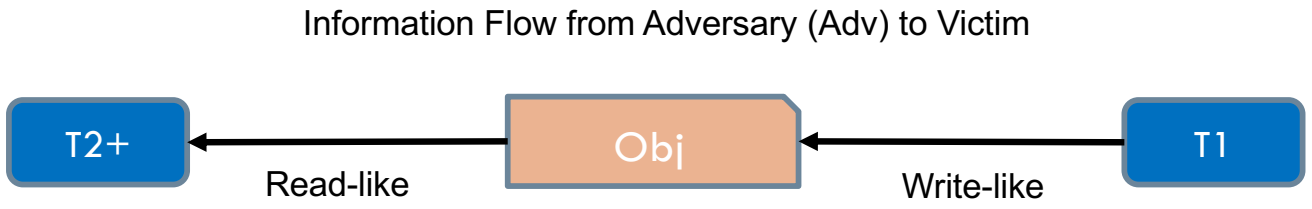
Android Privilege Levels

Process Level[1]	Level Membership Requirements
Root Process (T5)	Process running with UID root
System Process (T4)	Process running with UID system
Service Process (T3)	AOSP core service providers
Trusted Application Process (T2)	AOSP default and vendor apps
Untrusted Application Process (T1)	Third-party applications
Isolated Process (T0)	Processes assumed to be under adversary control

- Android defines process privilege levels roughly based on provided of the app – 3rd party T1, OEM T2-T3, Google T4-T5
  - Each program is assigned a privilege level
- Can assume program of a lower privilege level is adversarial
  - E.g., a program a T1 is an adversary of T2

Back to Access Control Policy Analysis

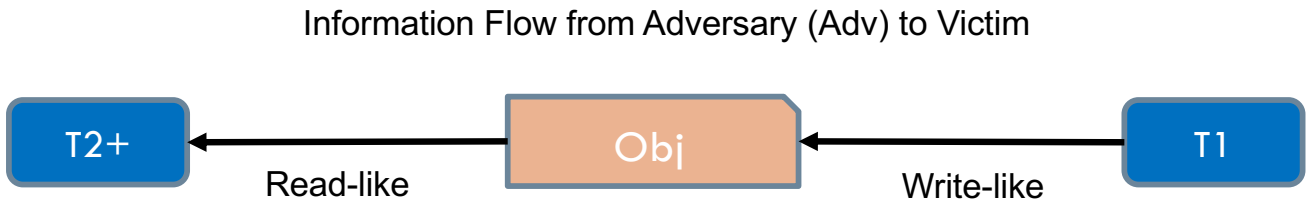
- Look for cases where an adversary subject can write a directory that can be read by a victim



- How do we use the Android Privilege Levels to help?

Back to Access Control Policy Analysis

- Look for cases where an adversary subject can write a directory that can be read by a victim



- How do we use the Android Privilege Levels to help?
  - Find any directory (Obj) that a T1 program has write permission for and a T2+ program has read/execute permission for - check for vulnerability at runtime (STING)

Conclusions

- 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
    - File squatting can control file content used by your program
    - Link traversal can redirect your program to other files
- Can identify the resources (directories) prone to such attacks via access control analysis
  - Remains a major problem

Questions