CIS 4360
Secure Computer Systems

File System Security

Professor Qiang Zeng
Spring 2017

TEMPLE
UNIVERSITY

Previous Class

- Virus vs. Worm vs. Trojan
- Drive-by download
- Botnet
- Rootkit

Trojan vs. Virus vs. Worm

	Trojan	Virus	Worm
Self-replicated	N	Y	Y
Self-contained	Y	N	Y
Relying on exploitation of vulnerabilities	N	Maybe (e.g., scripting viruses)	Y

Previous Class

It is possible that an experienced attacker may combine the techniques of viruses and worms (called blended attack). Could you find a concrete example among the famous worm attacks?

For example, Melissa (1998) sends itself through emailing, which is the behavior of worms; besides, it also infects local documents by copying itself into them, which is the behavior of viruses

There are many such examples that combine worms and viruses: Nimda, Conficker, Stuxnet

Previous Class

Does a drive-by download attack always succeed when you open a malicious webpage?

No. If there are no vulnerabilities in your browser, drive-by downloads cannot succeed. By design the scripting code (e.g., Javascript code) should not cause harms; it relies on exploiting vulnerabilities of browsers to gain extra privileges to download and install malware. So it is important to keep your browser up to date

Previous Class

Describe the main components in a classic botnet structure

(1) Botmaster
(2) C&C Servers
(3) Bots

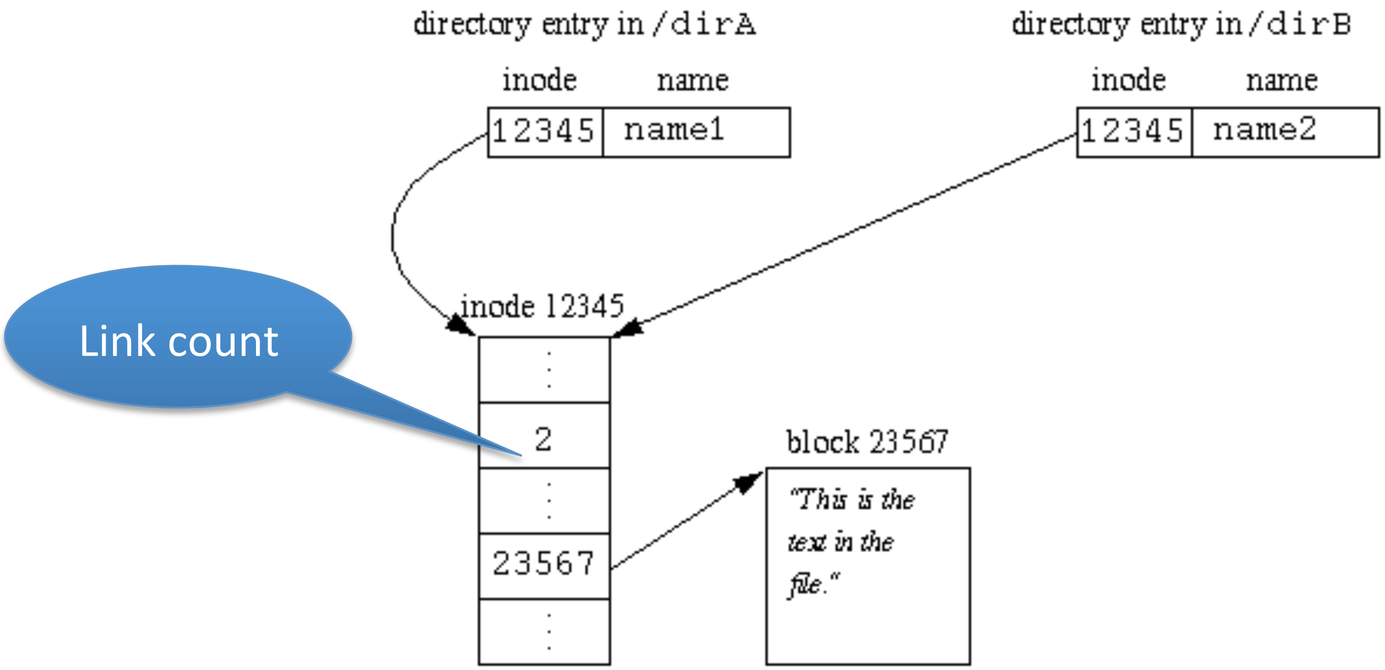
Outline

- Hard links vs. symbolic links
- File access permissions
- User and process credentials
- Special flags: setuid, sticky bit
- TOCTTOU

Hard link and soft link

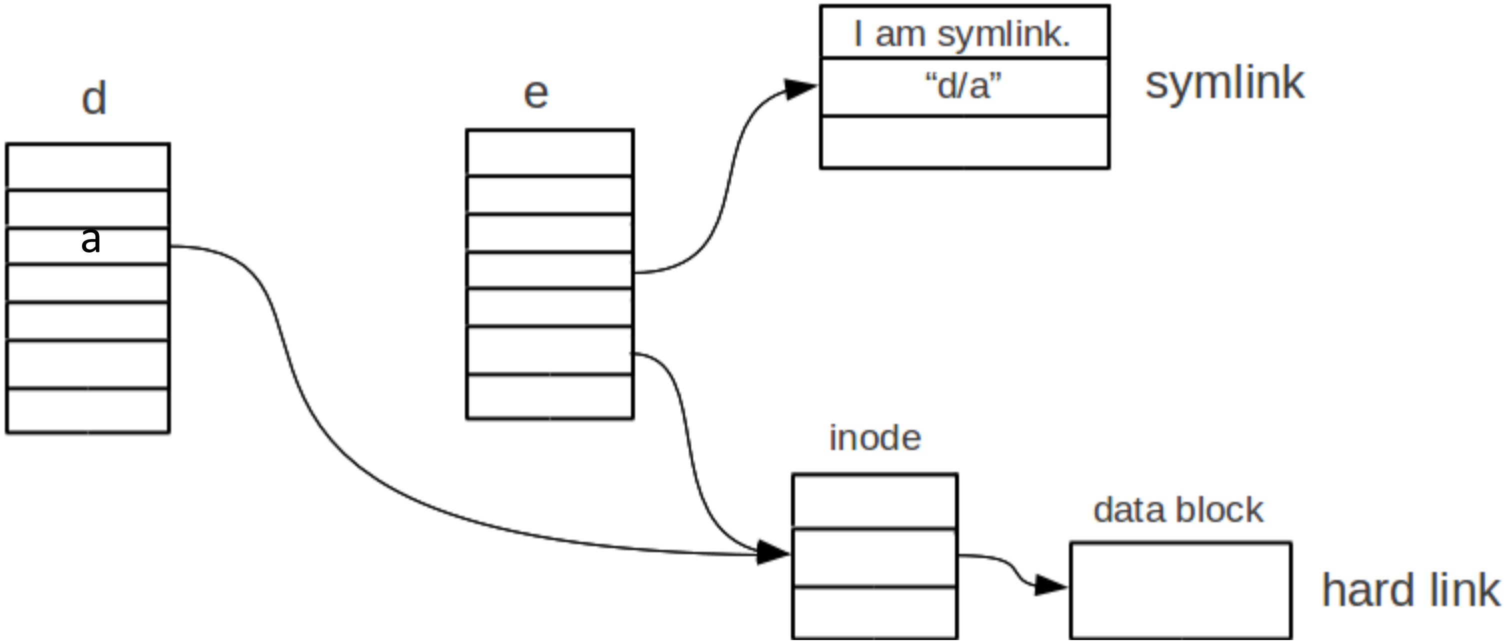
- In Linux/Unix, a file consists of a block, called inode, for storing metadata (file type, size, owner, etc.) and zero or more data blocks
- A hard link: a mapping from a file name to the id of an inode block
- A soft/symbolic link: a mapping from a file name to another file name

Hard link



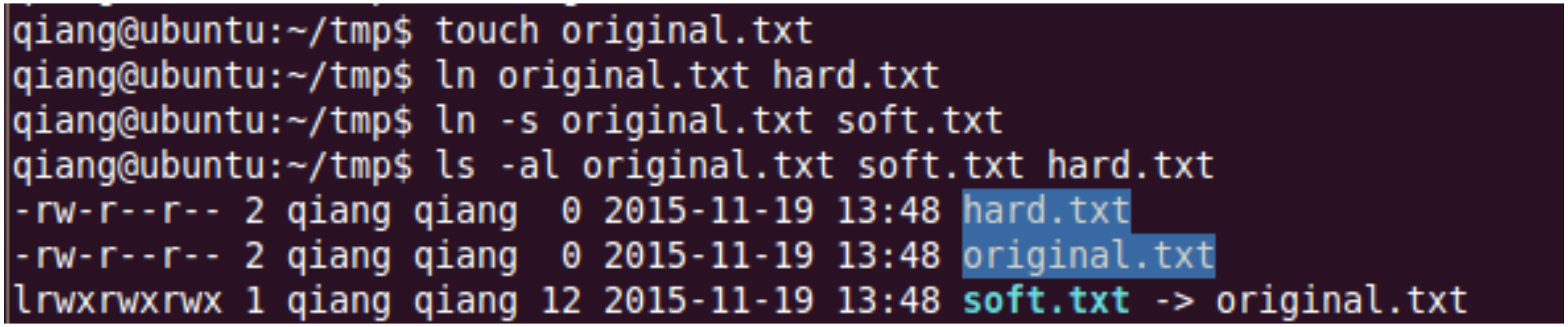
- When you create a hard link you simply create another link that points to the inode block.
- Only after the last hard link is removed (and no runtime file descriptors point to it), will the underlying file be deleted

Symbolic link



- The inode of a symbolic file contains:
  - A flag saying that "I am symbolic link"
  - A file name of the target file
- Symbolic links are very important for software upgrade
  - After upgrade, you just redirect the symbolic link to the new version
- A symbolic link may get dangling if the target file has been deleted

Create hard link and soft (symbolic) link



```
qiang@ubuntu:~/tmp$ touch original.txt
qiang@ubuntu:~/tmp$ ln original.txt hard.txt
qiang@ubuntu:~/tmp$ ln -s original.txt soft.txt
qiang@ubuntu:~/tmp$ ls -al original.txt soft.txt hard.txt
-rw-r--r-- 2 qiang qiang  0 2015-11-19 13:48 hard.txt
-rw-r--r-- 2 qiang qiang  0 2015-11-19 13:48 original.txt
lrwxrwxrwx 1 qiang qiang 12 2015-11-19 13:48 soft.txt -> original.txt
```

- We have created a file *original.txt*, and a hard link named *hard.txt*, and a symbolic link named *soft.txt*
- Can you distinguish *original.txt* and *soft.txt*?
  - Certainly
- Can you distinguish *original.txt* and *hard.txt*?
  - Hmmm…

Question

- If you modify a file through a hard link, will the modification time of another hard link of the same file be updated as well?
  - Yes
  - They point to the same inode block, which stores the modification time and other metadata
  - Hard links of a file share the same piece of “metadata and data” of the file; the only difference is the names

Outline

- Hard links vs. symbolic links
- File access permissions
- User and process credentials
- Special flags: setuid, sticky bit
- TOCTTOU

File permissions

- File permissions are about who can access the file and how it can be accessed
- Who:
  - U: the file owner
  - G: a group of user
  - O: other users
  - (A: everybody)
- How:
  - Read, write and execute

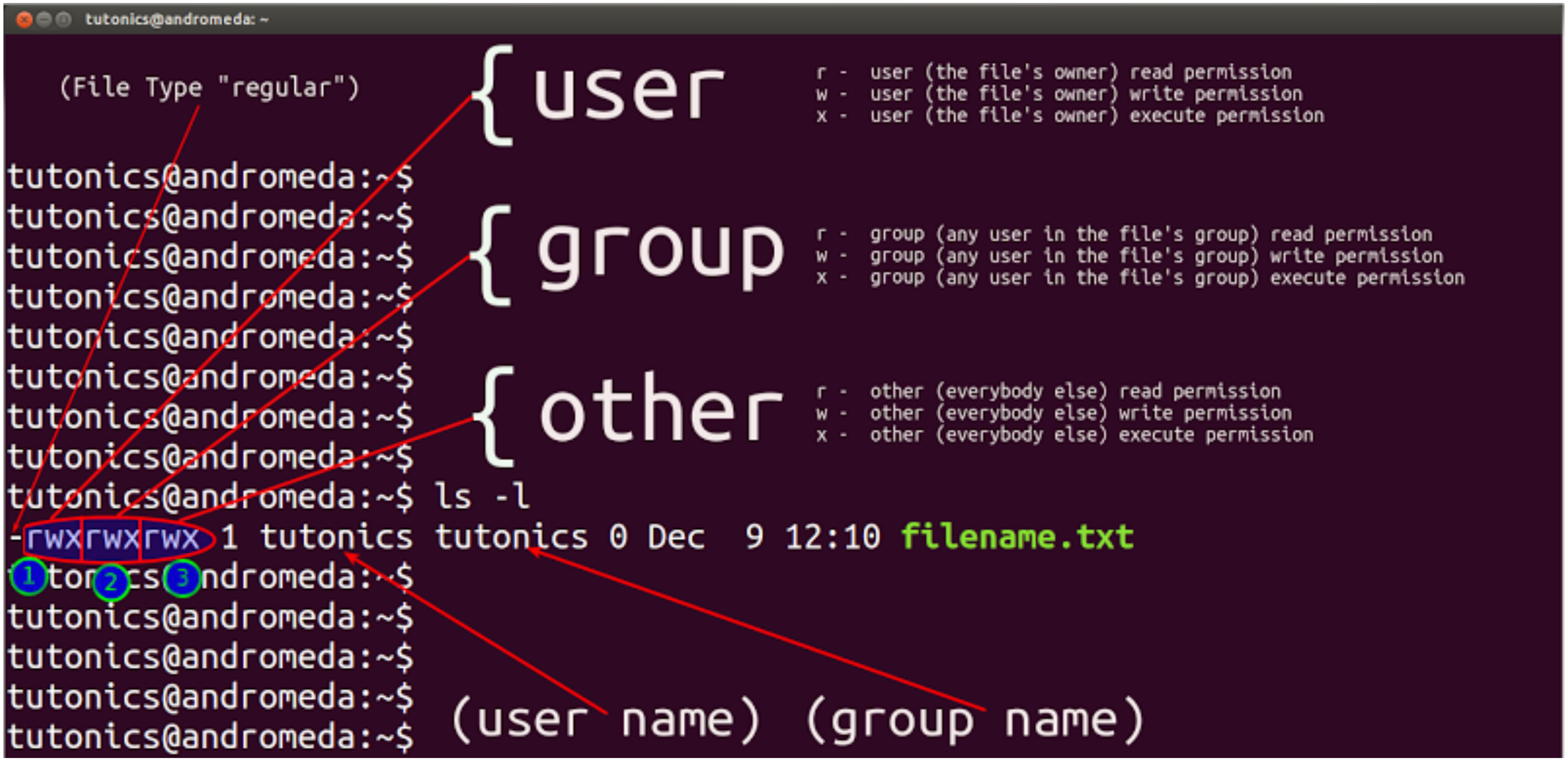
Permission on Directories

- Read: **list the files in the directory**
- Write: create, rename, or delete files within it
- Execute: **lookup a file name in the directory**

Questions

- To read */a/b/c.txt,* you need
  - the execute permission for /, *a,* and *b*
  - the read permission for *c.txt*
- To remove */a/b/c.txt,* you need
  - the execute permission for /, *a* and *b*
  - the write permission for *b*

Three subsets (for u, g, o) of bits;
each subset has three bits (for r, w, x)



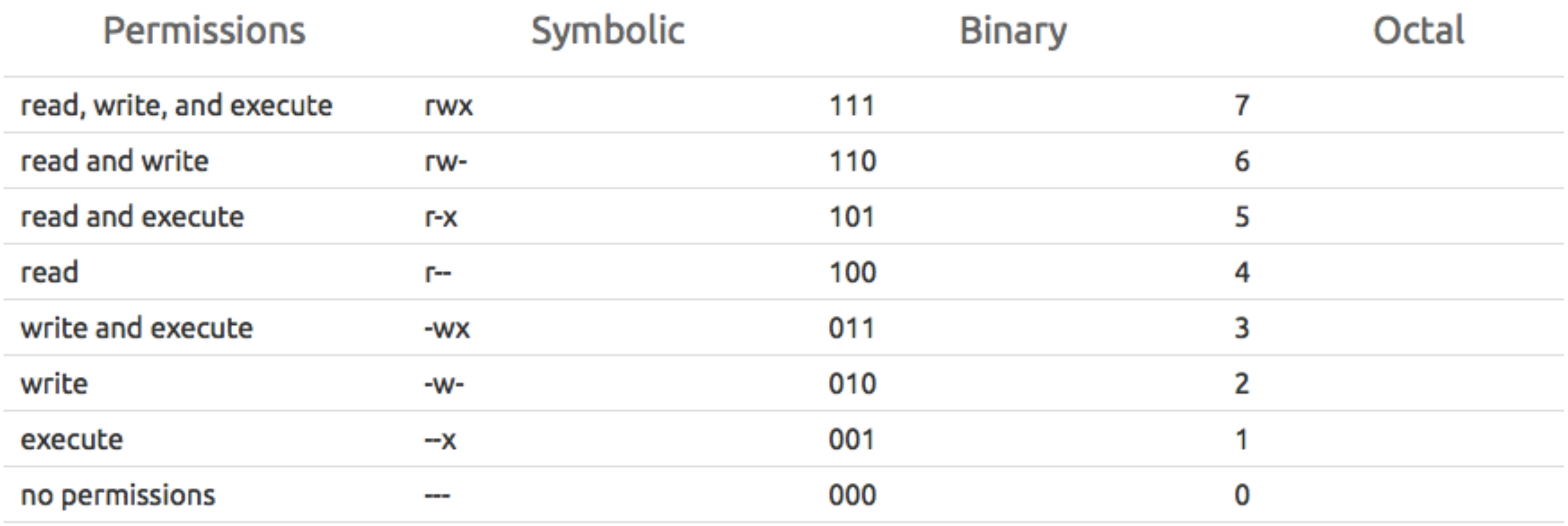
```
r -  user (the file's owner) read permission
w -  user (the file's owner) write permission
x -  user (the file's owner) execute permission

r -  group (any user in the file's group) read permission
w -  group (any user in the file's group) write permission
x -  group (any user in the file's group) execute permission

r -  other (everybody else) read permission
w -  other (everybody else) write permission
x -  other (everybody else) execute permission
```

```
tutonics@andromeda:~$
tutonics@andromeda:~$
tutonics@andromeda:~$
tutonics@andromeda:~$
tutonics@andromeda:~$
tutonics@andromeda:~$
tutonics@andromeda:~$
tutonics@andromeda:~$
tutonics@andromeda:~$ ls -l
-rwxrwxrwx 1 tutonics tutonics 0 Dec  9 12:10 filename.txt
tutonics@andromeda:~$
tutonics@andromeda:~$
tutonics@andromeda:~$
tutonics@andromeda:~$
tutonics@andromeda:~$
```

Octal representation



Permissions	Symbolic	Binary	Octal
read, write, and execute	rwx	111	7
read and write	rw-	110	6
read and execute	r-x	101	5
read	r--	100	4
write and execute	-wx	011	3
write	-w-	010	2
execute	--x	001	1
no permissions	---	000	0

Application of the octal representation

- `755: rwxr-xr-x`
  - `chmod 755 dir`
  - Specify the permissions of *dir*
- `644: rw-r--r--`
  - `chmod 644 a.txt`
  - Specify the permissions of *a.txt*

Changing file permissions using symbolic-mode

- To add x permissions for all
  - chmod a+x filename
- To remove w permissions for g and o
  - chmod go-w filename
- To overwrite the permissions for owner
  - chmod u=rw filename

Questions

- Why is it dangerous to operate on files in a publicly writable directory?
  - "A directory is publicly writable" means anyone including the attacker can create, delete, rename files in that dir
  - When you open a file "x", which you believe is what you have created previously, the attacker may first delete "x" and then create a file named "x" with permissions 777; consequently,
    - Integrity: "x"'s content is actually controlled by the attacker
    - Confidentiality: the attacker can read the file
  - There are other attacks, e.g., privilege escalation, DoS, race conditions

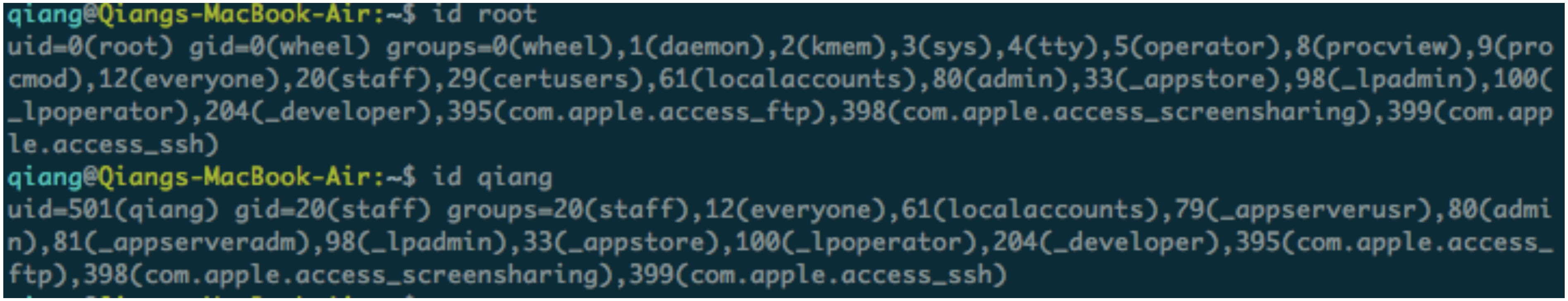
So, try you best not to use a publicly writable directory; files in such a directory should be treated untrusted

Outline

- Hard links vs. symbolic links
- File access permissions
- User and process credentials
- Special flags: setuid, sticky bit
- TOCTTOU

User credentials

- uid: user ID
- gid: the ID of a user's primary group
- groups: supplementary groups
- Collectively, they constitute the user credential

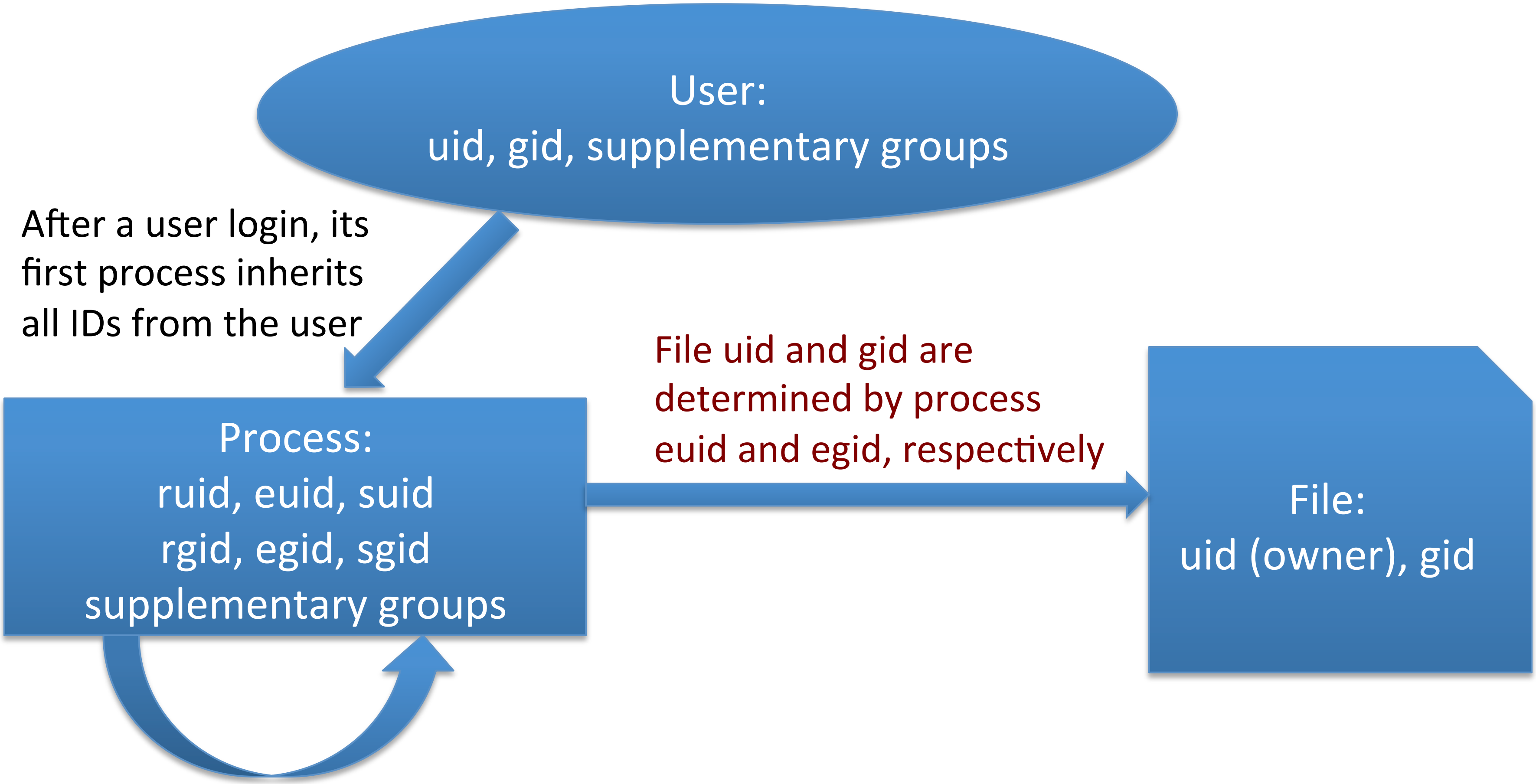


```
qiang@Qiangs-MacBook-Air:~$ id root
uid=0(root) gid=0(wheel) groups=0(wheel),1(daemon),2(kmem),3(sys),4(tty),5(operator),8(procview),9(pro
cmod),12(everyone),20(staff),29(certusers),61(localaccounts),80(admin),33(_appstore),98(_lpadmin),100(
_lpoperator),204(_developer),395(com.apple.access_ftp),398(com.apple.access_screensharing),399(com.app
le.access_ssh)
qiang@Qiangs-MacBook-Air:~$ id qiang
uid=501(qiang) gid=20(staff) groups=20(staff),12(everyone),61(localaccounts),79(_appserverusr),80(admi
n),81(_appserveradm),98(_lpadmin),33(_appstore),100(_lpoperator),204(_developer),395(com.apple.access_
ftp),398(com.apple.access_screensharing),399(com.apple.access_ssh)
```

Process credentials

- Each process has
  - Real, effective, saved user IDs (ruid, euid, suid)
  - Real, effective, saved group IDs (rgid, egid, sgid)
  - Supplementary group IDs
- After a user login, its first process inherits all its IDs from the user
  - E.g., if a user (uid = 1000, gid=2000) logs in, then its first process's ruid=euid=suid=1000 and rgid=egid=sgid=2000
- At fork(), all the IDs are inherited by the child

A little wrap-up



User:
uid, gid, supplementary groups

After a user login, its first process inherits all IDs from the user

Process:
ruid, euid, suid
rgid, egid, sgid
supplementary groups

File uid and gid are determined by process euid and egid, respectively

File:
uid (owner), gid

When a process is forked, the child inherits all the IDs

Permission checking

- Note that process's credential is used (rather than the user's) during permission checking
- Recall that the permissions of each file has three groups of three bits (e.g., rwxr-x--x)
  - If process euid = file owner ID, the 1st group ("rwx") is used
  - If process egid or any of the supplementary group IDs = file group ID, the 2nd group ("r-x") is used
  - The 3rd group ("--x") is used if neither above holds

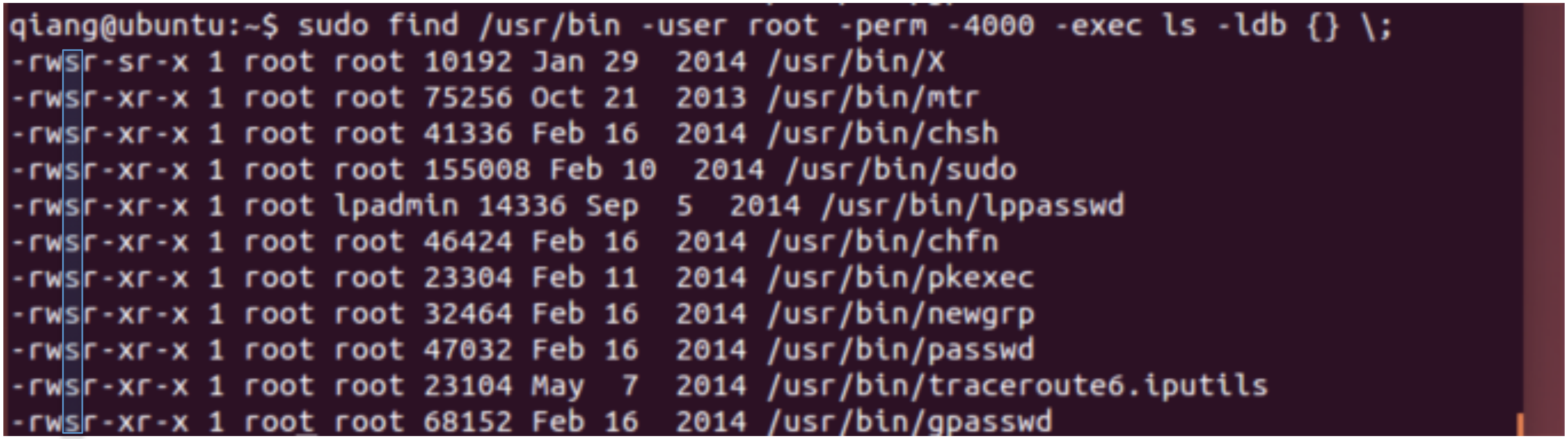
Outline

- Hard links vs. symbolic links
- File access permissions
- User and process credentials
- Special flags: setuid, sticky bit
- TOCTTOU

Setuid programs

- Setuid: short for “set user ID upon execution”
- When a non-setuid program is executed, its user IDs are inherited from its parent
- However, when a setuid program is executed, its effective and saved user ID will be set as the owner of the program
  - The process has the privileges of the program owner
  - If the program owner is root, we call it a setuid-root program, or the program is setuid to root; such processes have root privileges

Examples

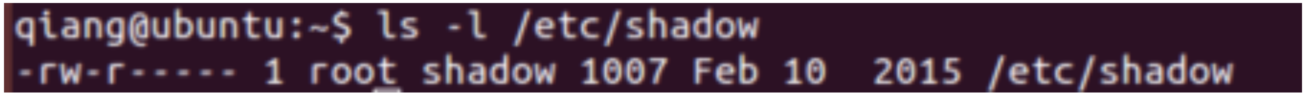


```
qiang@ubuntu:~$ sudo find /usr/bin -user root -perm -4000 -exec ls -ldb {} \;
-rwsr-sr-x 1 root root 10192 Jan 29  2014 /usr/bin/X
-rwsr-xr-x 1 root root 75256 Oct 21  2013 /usr/bin/mtr
-rwsr-xr-x 1 root root 41336 Feb 16  2014 /usr/bin/chsh
-rwsr-xr-x 1 root root 155008 Feb 10  2014 /usr/bin/sudo
-rwsr-xr-x 1 root lpadmin 14336 Sep  5  2014 /usr/bin/lppasswd
-rwsr-xr-x 1 root root 46424 Feb 16  2014 /usr/bin/chfn
-rwsr-xr-x 1 root root 23304 Feb 11  2014 /usr/bin/pkexec
-rwsr-xr-x 1 root root 32464 Feb 16  2014 /usr/bin/newgrp
-rwsr-xr-x 1 root root 47032 Feb 16  2014 /usr/bin/passwd
-rwsr-xr-x 1 root root 23104 May  7  2014 /usr/bin/traceroute6.iputils
-rwsr-xr-x 1 root root 68152 Feb 16  2014 /usr/bin/gpasswd
```

Take */usr/bin/passwd* as an example; it is a setuid-root program

Why are setuid programs needed?

- Consider the *passwd* example
- It is to update the password file /etc/shadow
- Obviously, its file permission is 640 and it is owned by root



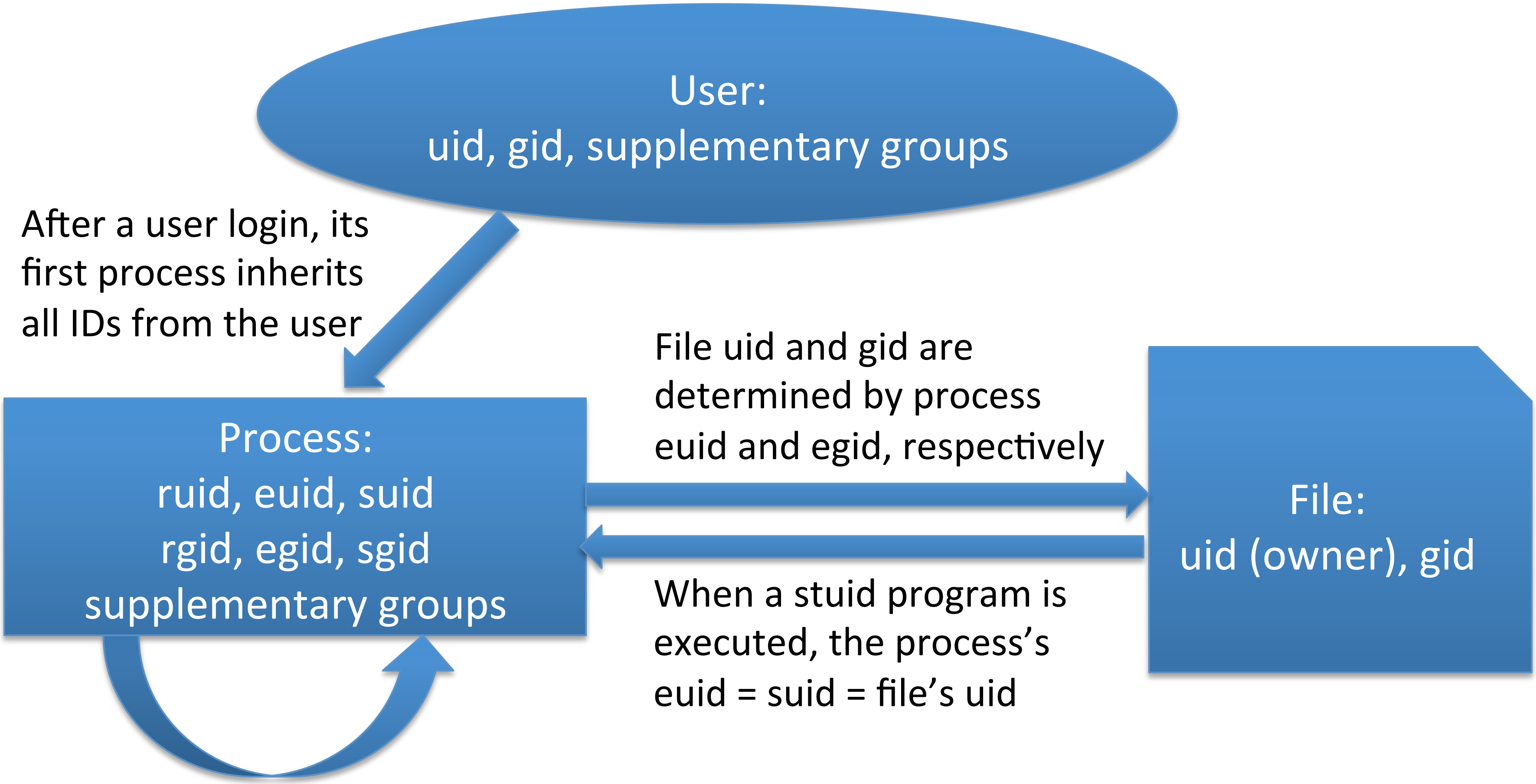
```
qiang@ubuntu:~$ ls -l /etc/shadow
-rw-r----- 1 root shadow 1007 Feb 10  2015 /etc/shadow
```

- Then, how can a process created by non-root user modify the sensitive file?
- Answer: setuid program
  - So that when it is run, it has the effective ID = file owner, which enables it to modify /etc/shadow

Setgid

- Setgid programs have similar effects as setuid ones
  - egid = program's gid
- Setuid only makes sense with executable files
- Setgid makes sense with executable files; it also makes sense with directories
  - Any files created in that directory will have the same group as that directory.
  - Also, any directories created in that directory will also have their setgid bit set
  - The purpose is usually to facilitate file sharing through the directory among users
- Setgid even makes sense with non-executable files to flag mandatory locking files. Please refer to the article
  - https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt

Another little wrap-up



User:
uid, gid, supplementary groups

After a user login, its first process inherits all IDs from the user

Process:
ruid, euid, suid
rgid, egid, sgid
supplementary groups

File uid and gid are determined by process euid and egid, respectively

File:
uid (owner), gid

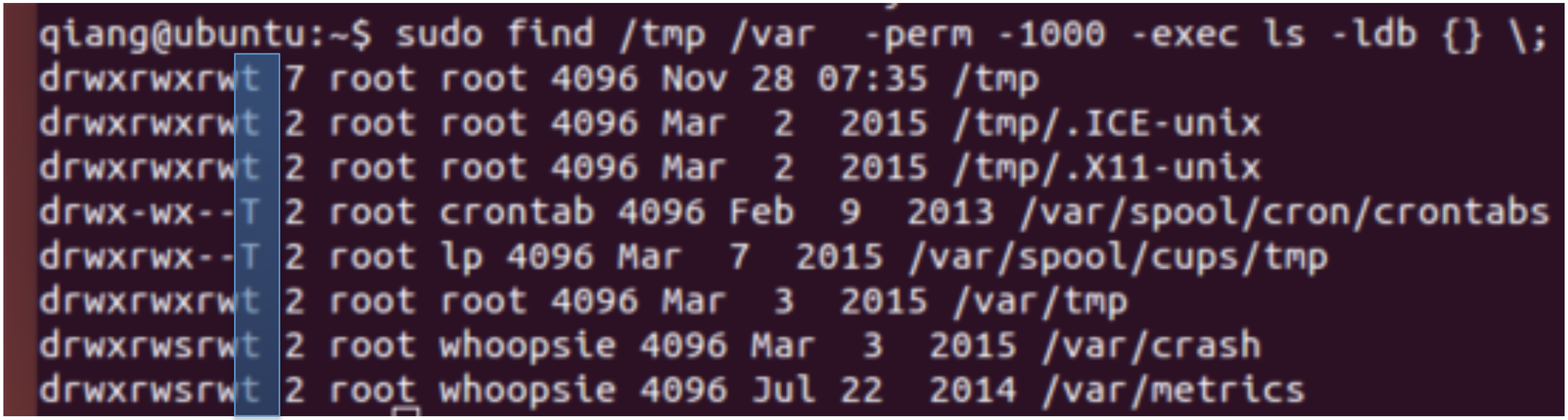
When a stuid program is executed, the process's euid = suid = file's uid

When a process is forked, the child inherits all the IDs

Sticky bit

- Historically, it got the name because it makes the related files stick in main memory
- Now it only makes sense with directories
- Normally, if a user has write permission for a directory, he/she can delete or rename files in the directory regardless of the files' owner
- But, files in a directory with the sticky bit can only be renamed or deleted by the file owner (or the directory owner)

Example



```
qiang@ubuntu:~$ sudo find /tmp /var  -perm -1000 -exec ls -ldb {} \;
drwxrwxrwt 7 root root 4096 Nov 28 07:35 /tmp
drwxrwxrwt 2 root root 4096 Mar  2  2015 /tmp/.ICE-unix
drwxrwxrwt 2 root root 4096 Mar  2  2015 /tmp/.X11-unix
drwx-wx--T 2 root crontab 4096 Feb  9  2013 /var/spool/cron/crontabs
drwxrwx--T 2 root lp 4096 Mar  7  2015 /var/spool/cups/tmp
drwxrwxrwt 2 root root 4096 Mar  3  2015 /var/tmp
drwxrwsrwt 2 root whoopsie 4096 Mar  3  2015 /var/crash
drwxrwsrwt 2 root whoopsie 4096 Jul 22  2014 /var/metrics
```

In the x-bit location for others:
x + sticky = t
- + sticky = T

Why is the sticky bit needed?

- /tmp, for example, typically has 777 permissions, which means everyone has r/w/x privileges for it
- If you don’t want your files created in /tmp to be deleted by others, the Sticky bit is your choice

How to set the setuid, setgid, and Sticky bits

- 4 = setuid, 2 = setgid, 1 = sticky bit
- chmod 4766 filename
  - Set the setuid bit along with permissions 766
- chmod 2771 filename
  - Set the setgid bit along with permission 771
- Symbolic-mode
  - chmod u+s filename
  - chmod g-s dirname
  - chmod +t dirname

Outline

- Hard links vs. symbolic links
- File access permissions
- User and process credentials
- Special flags: setuid, sticky bit
- TOCTTOU

Race condition

- A Race Condition is a bug when the result depends on the sequence or timing of events
- In file systems, the race condition is usually due to TOCTTOU (Time Of Check To Time Of Use)
- A TOCTTOU vulnerability involves two sys calls
  - Check: learn some fact about a file
    - E.g., whether a file is accessible, exists etc.
  - Use: based on the previous fact
    - E.g., access the file, create a file if a file doesn't exist

TOCTTOU attack against file systems

- A TOCTTOU attack exploits a TOCTTOU vulnerability with a concurrent attack launched between “check” and “use”
- When the process proceeds to the “use” step, it still believes that “fact” holds, which, however, is not true due to the attack

Printing example

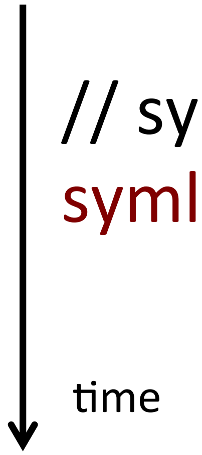
- A printing program is setuid-root for accessing the printer device
- When a user requests to print a file by providing a pathname, the printing process should not accept the request directly. Why?
  - Because a malicious user may provide a pathname like *"/etc/shadow"*, which stores the password information of all users

Printing does not want to be fooled

- It first checks whether the user has the permission to access that file
- System call *access(pathname, r|w|x|exist)*
  - uses the process's *ruid/rgid/groups* to return whether the user has the correct permission for the file with the specified pathname

Typical wrong implementation: access/ open pair

- System call *access(pathname, rlwlexist)*
  - uses the process's *ruid/rgid/groups* to return whether the user has the correct permission for the file with the specified pathname
- Attacks changes the file associated with the pathname between check and use



```
if(!access("foo", R_OK)) {
                                // symlink(target, linkpath)
                                symlink("/etc/shadow", "foo");

  fd = open("foo", read);
  …
}
```

Ad-hoc solution to resolving issues due to setuid process's privileges

- Relinquish the privileges temporarily. *seteuid(new_euid)* allows you to change the euid of the process as long as *new_euid = ruid ll new_euid = suid*
  - (1) The printing process calls *seteuid(ruid)* to relinquish the privileges due to the euid
  - (2) Call *open()* // now you cannot cheat
  - (3) Call *seteuid(suid)* // recover the privileges

Another TOCTTOU example: file creation in /tmp

```
(1) filename = “/tmp/X”;
(2) error = lstat(filename, metadata_buf);
(3) if (error) // “/tmp/X” does not exist
(4)    f = open(file, O_CREAT); // create the file
```

- Between line (2) and line (4), an attacker may create a symlink pointing to some secret file that the victim process can access
- So that when the victim process calls open(), it opens the secret file instead of creating “/tmp/X”

Securing Programming Guideline

- The system should service “check” and “use” as a transaction; i.e., to do them in an atomic way
- To deal with the stat/open problem, you should use
  - open(filename, O_CREAT | O_EXCL) // it atomically checks the existence of file and creates it only if it doesn’t exist. It returns error if it already exists;
  - Or mkstemp() atomically coin a unique a name at the specified directory and creates it (so it is impossible for an attacker to create a link with that name)

History and references about TOCTTOU

- ’95, Bishop first systematically described the TOCTTOU flaws in file systems
  - “Race conditions, files, and security flaws”
- ’03, Tsyrklevich & Yee proposed pseudo transaction and a couple of nice points
  - “Dynamic Detection and Prevention of Race Conditions in File Accesses” Usenix Security
- ’04, Dean & Hu proves that it has no deterministic solution without changing kernel, and proposed a probabilistic defense
  - “Fixing Races for Fun and Profit: How to use access(2)” Usenix Security

History and references

- ‘05, quickly, the defense was “beautifully and thoroughly demolished” by Borisov et al.
  - “Fixing Races for Fun and Profit: How to use atime”, Usenix Security
- ‘08, later, the defense was enhanced by Tsafrir et al., who “claims” that it cannot be bypassed
  - "Portably Solving File TOCTTOU Races with Hardness Amplification”, FAST
- ‘09, soon, the enhanced defense was broken
  - "Exploiting Unix File-System Races via Algorithmic Complexity Attacks”, Security & Privacy

Summary

- Hard links vs. symbolic links
- File access permissions
- User and process credentials
- Special flags: setuid, sticky bit
- TOCTTOU

Writing Assignments

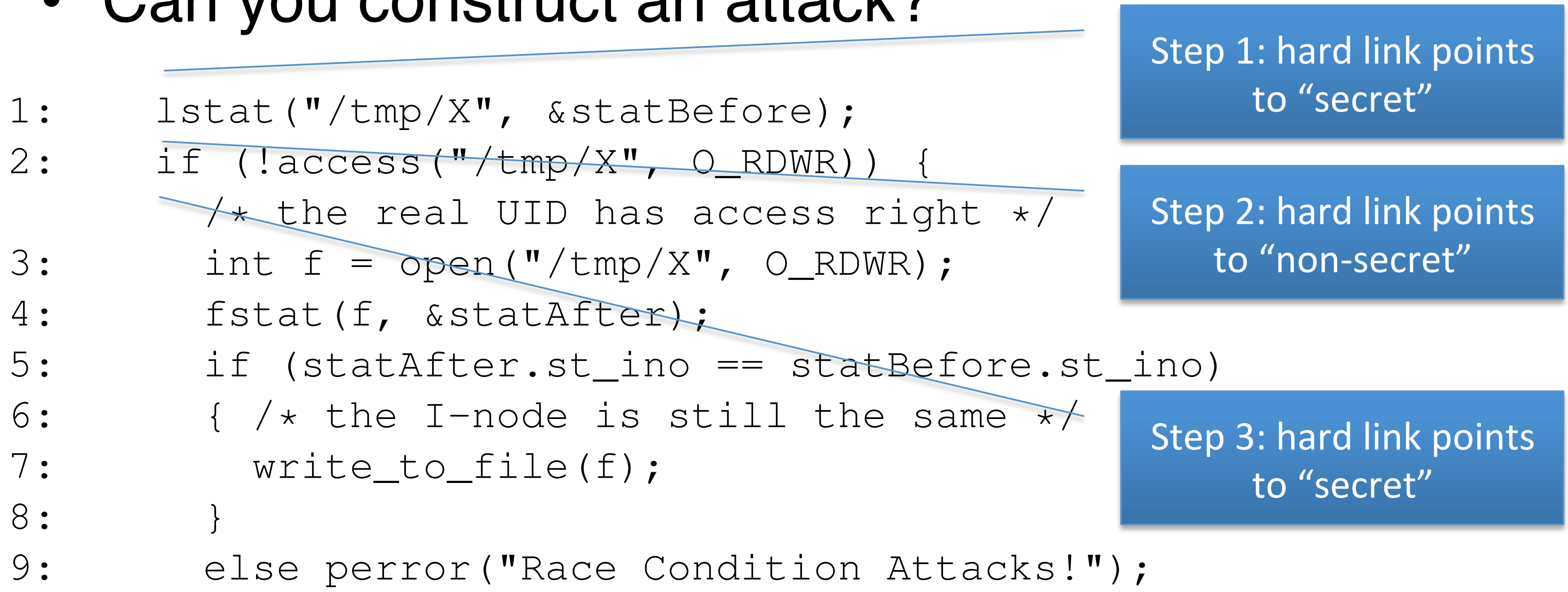
- How doe the sticky bit improve the security of using the directory /tmp?
- Describe, even with the sticky bit, why /tmp is till insecure to use? What is the lesson?

Background

- lstat(): retrieve the metadata of a file given its pathname; if the file is a symbolic link, retrieve the metadata about the symbolic link instead of the target
- stat(): similar to lstat(), but if the pathname is a symbolic link, retrieve the metadata of the target
- fstat(): retrieve the metadata of a file given the file descriptor pointing to the file's inode block

A seemingly smart but wrong implementation: check-use-check-again

- The following code is copied from a security expert's notes. He thinks it is correct
- Can you construct an attack?



```
lstat("/tmp/X", &statBefore);
if (!access("/tmp/X", O_RDWR)) {
  /* the real UID has access right */
  int f = open("/tmp/X", O_RDWR);
  fstat(f, &statAfter);
  if (statAfter.st_ino == statBefore.st_ino)
  { /* the I-node is still the same */
    write_to_file(f);
  }
  else perror("Race Condition Attacks!");
```

Step 1: hard link points to "secret"

Step 2: hard link points to "non-secret"

Step 3: hard link points to "secret"

How does an attacker win with a very high probability?

- At first glance, the chance for the attacker to win is very low. After all, the time window between access and open is usually very small
- Attacker's target: enlarge the time window. How?
  - The key is the pathname
  - File system mazes: force the victim to resolve a path that is not in the OS cache and thus involves I/O
  - Algorithmic complexity attacks: force the victim to spend its scheduling quantum to traverse the cache's hash table; adverse hash collision with the specified pathname