

CIS 5512 - Operating Systems

File Systems Security

Professor Qiang Zeng
Fall 2017



Previous class...

- File and directory
 - Hard link and soft link
 - Mount
 - Layered structure
- File system design
 - Naïve: linked list of blocks
 - FAT32: tabular design based on its file allocation table
 - UFS/ext2/ext3: multi-level search tree



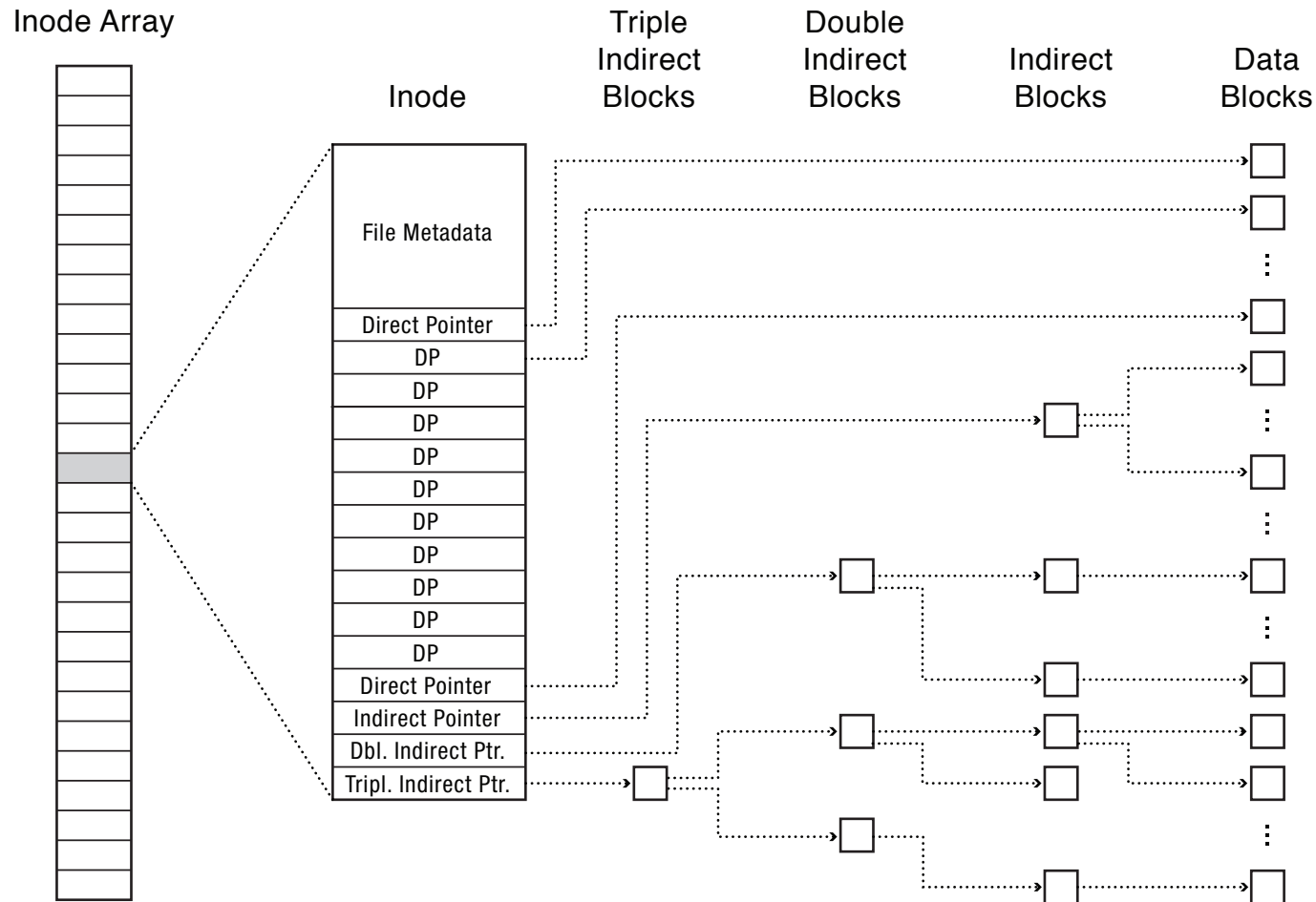
Previous class...

Why does not the file system design use a uniform two or three-level index design, as used in page tables

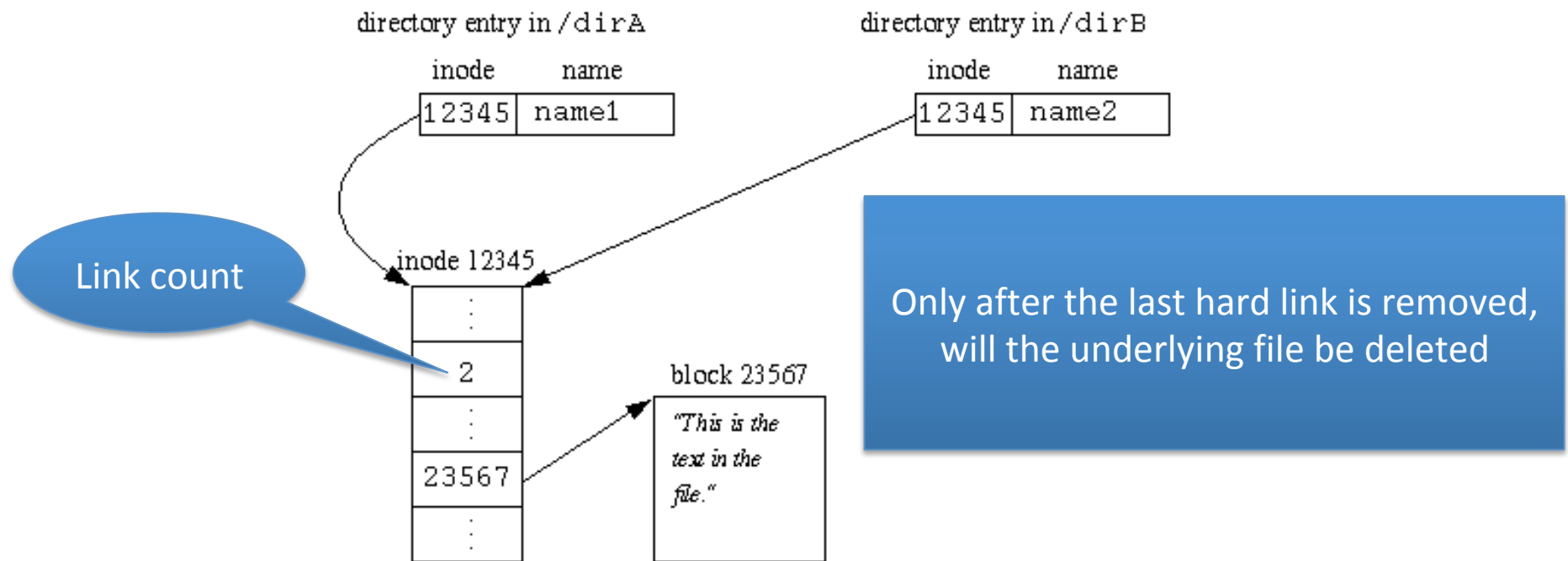
Compared to a uniform n -level index design, the multilevel index design is more scalable and efficient. E.g., if the file size is small, a direct pointer is sufficient, which is quick and saves space; if the file is very large, those indirect pointers can be used



Multilevel Index design

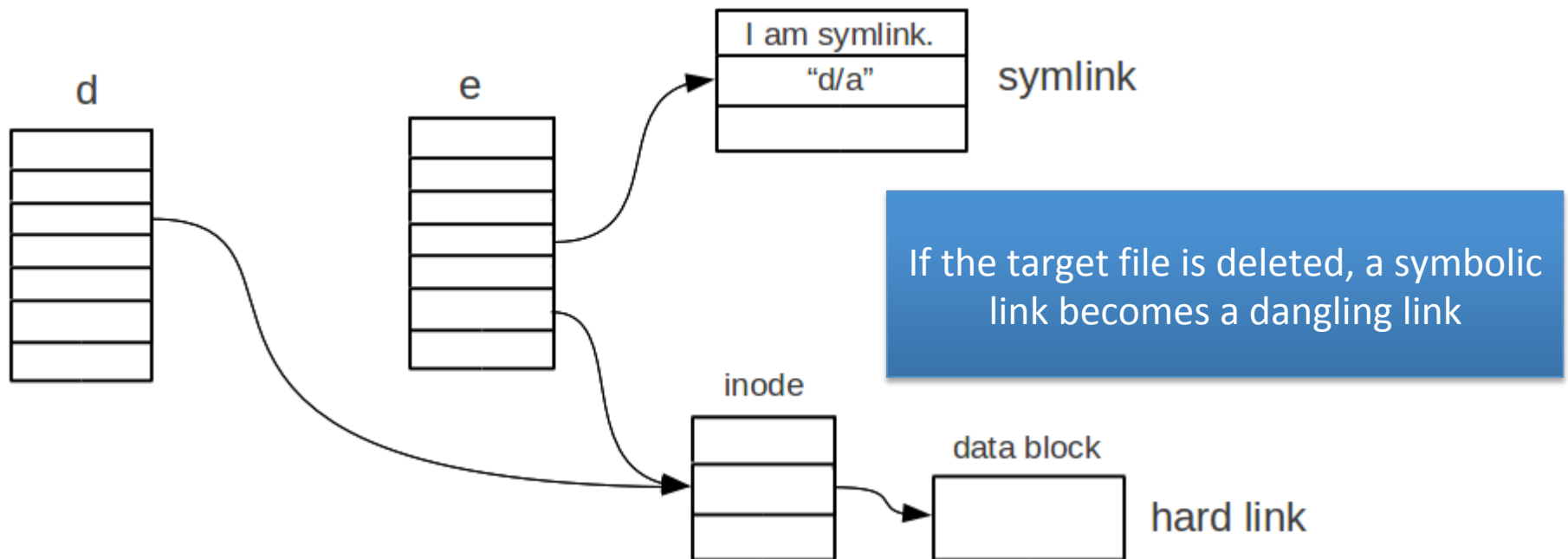


Truth about hard link



- inode block: this is the formal name used to refer to “the first block of a file” in Linux;
- **Hard link: another directory entry for a given file**

Symbolic link



- The inode of a symbolic file contains:
 - A flag saying that “I am symbolic link”
 - A file name of the target file
- Very important for software upgrade
 - After upgrade, you just redirect the symbolic link to the new version



Previous class...

read() vs fread()

`read ()` is a syscall, whereas `fread()` is a libc API. Between the two, buffering is utilized, so that fewer system calls are issued when calling `fread()`. E.g., the first `fread()` can internally call `read()` and read in a large buffer of the file content, so that the following `fread()` calls retrieve data from the buffer without triggering more system calls until the buffer is empty



Previous class...

read() vs fread()

`read ()` is a syscall, whereas `fread()` is a libc API. Between the two, buffering is utilized, so that fewer system calls are issued when calling `fread()`. E.g., the first `fread()` can internally call `read()` and read in a large buffer of the file content, so that the following `fread()` calls retrieve data from the buffer without triggering more system calls until the buffer is empty



Previous class...

What does “mounting a volume” do under the hood?

Each volume contains a file system and this volume is accessed through a specific driver. Thus, when a volume is successfully mounted, registration is done at the kernel, which associates correct file system driver and device driver with the volume, such that the files in the volume are operated on using correct drivers



Outline

- File access permissions
- User and process credentials
- Special flags: setuid, setgid, Sticky bit
- Pitfalls and secure programming guidelines



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
 - For a directory
 - Read: **list the files in the directory**
 - Write: create, rename, or delete files within the directory
 - 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)**

(File Type "regular")

{ user

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

{ group

- 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

{ other

- 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:~$
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:~$
```

(user name) (group name)



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: `rw-r--r--`
 - `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

- You have a personal website; what permissions do you give the directories and files?
 - Directories: 711 (rwx--x--x)
 - Files: 644 (rw-r--r--)
- What if you want your files in a directory to be listed?
 - Directories: 755 (rwxr-xr-x)
 - Remove the index.html in that directory



Outline

- File access permissions
- User and process credentials
- Special flags: setuid, setgid, Sticky bit
- Pitfalls and secure programming guidelines



User credentials

- uid: user ID
- gid: the ID of a user's primary group
- groups: supplementary groups
- Collectively, they determine which system resources an user can access

```
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(procm),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.apple.access_ssh)
qiang@Qiangs-MacBook-Air:~$ id qiang
uid=501(qiang) gid=20(staff) groups=20(staff),12(everyone),61(localaccounts),79(_appserverusr),80(admin),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)
```

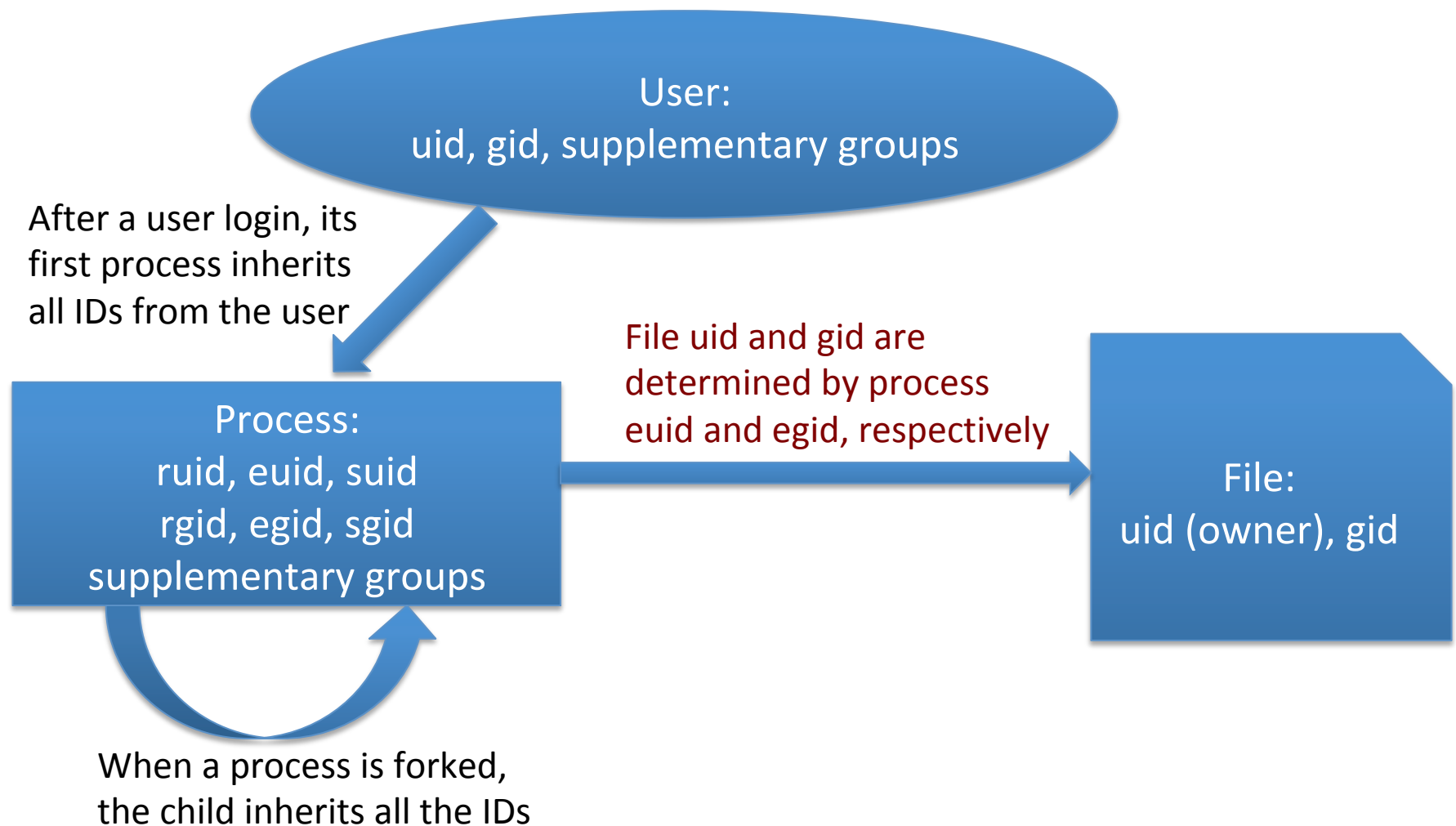


Process credentials

- Each process has
 - User IDs: real, effective, saved user IDs (ruid, euid, suid)
 - Group IDs: 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



Path resolution

- Step 1: if the path starts with '/' (i.e., absolute path), the current lookup directory is set as the root directory; otherwise, it is set as the current working directory
- Step 2: for each intermediate component
 - If the process does not have search permission on the current lookup directory, an error is returned; otherwise, search the name
 - If the component is found and it is a symbolic link, the resolution is performed on the linked path, recursively
 - If a component is resolved successfully, the current lookup directory is updated as the current directory
 - Note: “/./” is equivalent with “/”
- Step 3: for the final component
 - Similar to step 2; just it can be a file or a directory
 - http://man7.org/linux/man-pages/man7/path_resolution.7.html



Permissions checking for details path resolution and file access

- Note that process's credential is used (rather than the user's)
- 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 is used
 - If process `egid` or any of the supplementary group IDs = file group ID, the 2nd group is used
 - The 3rd group is used if neither above holds



Outline

- File access permissions
- User and process credentials
- Special flags: setuid, setgid, Sticky bit
- Pitfalls and secure programming guidelines



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: whoever runs this command, the created *passwd* process will get the root privileges (euid = root), since the program owner of is “root”



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
- 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`

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

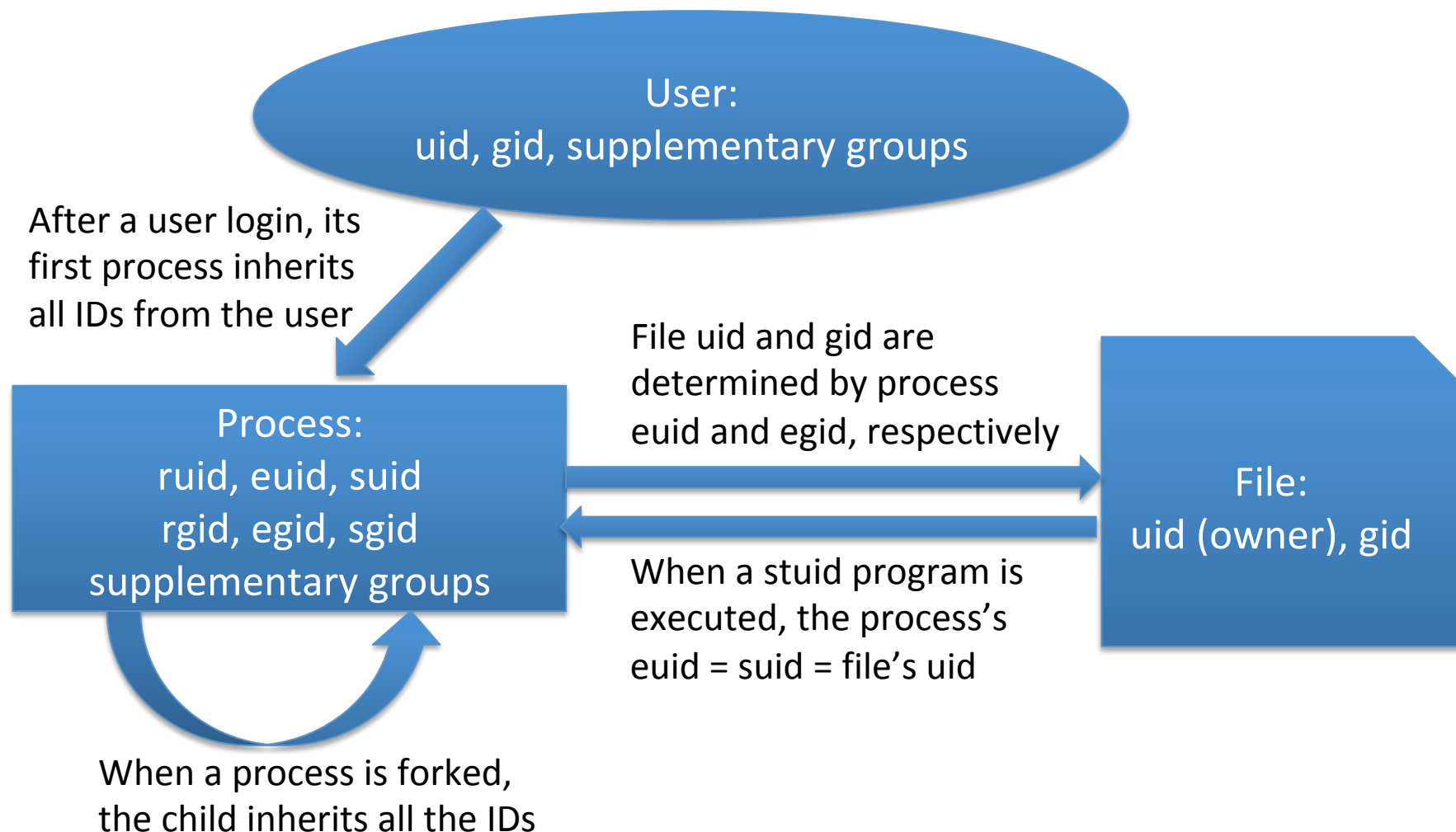


Setgid

- 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



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

- File access permissions
- User and process credentials
- Special flags: setuid, setgid, Sticky bit
- Pitfalls and secure programming guidelines



Pitfall 1: Race condition

- In file systems, the race condition is usually due to TOCTTOU (Time Of Check To Time Of Use) vulnerabilities (i.e., security bugs)
- A TOCTTOU vulnerability involves two system 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 involve a TOCTTOU vulnerability, and a concurrent attack between “check” and “use” ruins the “fact”
- But when the process proceeds to the “use” step, it still believes that “fact” holds



Printing example

- A printing program is usually setuid-root, which means that it runs with root privileges
- 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/lexist)*
 - 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

- The printing process checks file, if it's good, uses it (i.e., opens and prints it)
- Attacks changes the file associated with the pathname between check and use
- Fundamental issue: a filename doesn't mean a fixed file

```
if(!access("foo", R_OK)) {  
    // symlink(target, linkpath)  
    symlink("/etc/shadow", "foo");  
    fd = open("foo");  
    ...  
}
```

time



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* // *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
- This solution is ad hoc for setuid program, because it doesn't resolve the general TOCTTOU vulnerabilities in file systems



A little wrap-up: process ruid, euid, and suid

- ruid: indicates who created the process
- euid: the most important ID; it is used to check access permissions
- suid: used to recover privileges (after a process temporarily changed its euid)



**The rest of the slides will not be
examined in the exam**



Background

- lstat(): retrieve the metadata of a file given its **pathname**; if the file is a symbolic link, retrieve the metadata about the 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?

```
1:  lstat("/tmp/X", &statBefore);
2:  if (!access("/tmp/X", O_RDWR)) {
3:      /* the real UID has access right */
4:      int f = open("/tmp/X", O_RDWR);
5:      fstat(f, &statAfter);
6:      if (statAfter.st_ino == statBefore.st_ino)
7:      { /* the I-node is still the same */
8:          write_to_file(f);
9:      }
10:  }
11:  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



Another TOCTTOU example: file creation in /tmp

```
(1) filename = "/tmp/X";  
(2) error = stat(filename, metadata_buf);  
(3) if (error)  
(4)     f = open(file, O_CREAT); // create the file
```

- Again, assume the victim process has privileges that the attacker does not have but wants to leverage
 - Such an attack is called “**privilege escalation**” attack
- 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”



Generic solution: transactional service

- The system should service “check” and “use” as a transaction; that is, service them in an atomic way. E.g.,
- 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
 - `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)
- To deal with the access/open problem, the system should provide a flag `O_RUID` for `open()`
 - `open(filename, O_READ | O_RUID)`
 - This is not provided yet, though



History and references

- '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

