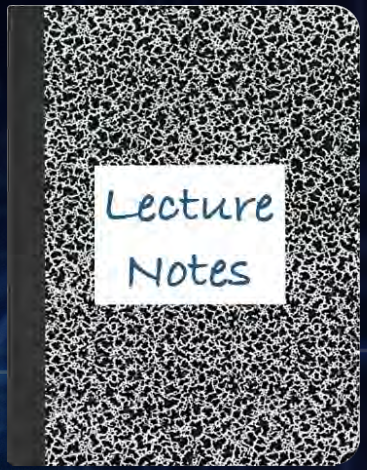
CS 419: Computer Security

# Week 6: Part 2
# POSIX Permissions



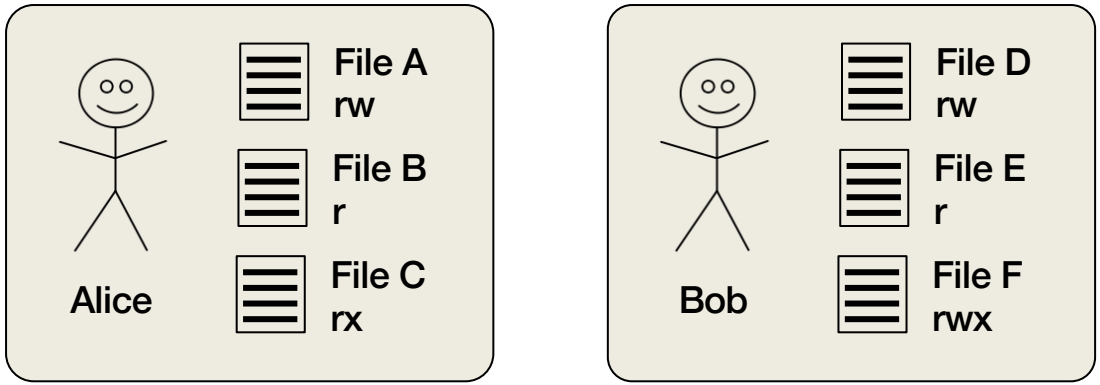

Paul Krzyzanowski

# File permissions

- **Access isn't all or nothing**
- **Objects can have different access permissions**

## UNIX (POSIX) permission model

- Access permissions: read (r), write (w), execute (x)
  - All independently set
- Each file has an owner

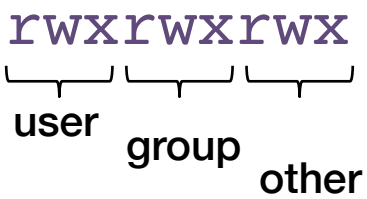


# Example: Limited ACLs in POSIX systems

- **Problem: ACLs takes up a varying amount of space**
  - They won't fit in a fixed-size inode
- **UNIX Compromise:**
  - A file defines access rights for three domains: the **owner**, the **group**, and **everyone else**
  - **Permissions**
    - Read, write, execute (for files), search (for directories)
    - Set user ID: execute with user permissions of the file's owner
    - Set group ID: execute with the group permissions of the file's group
  - Default permissions set by the `umask` system call
  - `chown` system call changes the object's owner
  - `chgrp` system call changes the object's group
  - `chmod` system call changes the object's permissions

# How do you share files?

- **Groups & everyone else (other)**
- **A user has one user ID but may belong to multiple groups**
  - One current default group ID for new objects
  - Multiple groups
- **Other = all others (users who are not the owner or group members)**
- **File access permissions are expressed as:**




```
$ ls -l /bin/ls
-rwxr-xr-x 1 root root 138216 Feb  7  2024 /bin/ls
```

# Permission checking

if you are the owner of the file
    *only* owner permissions apply

if you are part of a group the file belongs to
    *only* group permissions apply

else “other” permissions apply

I cannot read this file even if I’m in the *localaccounts* group:

```
$ ls -l testfile
 ----rw----  1 paul  localaccounts  6 Jan 30 10:37 testfile
```

# Execute permission

- **Distinct from read**
- **You may have execute-only access**
  - This takes away your right to copy the file
    … or inspect it
  - But the OS can load it & run it

# Windows

- **Windows has users & groups but more permissions**
  - Read, write, execute
  - Also: delete, change permission, change ownership
- **Users & resources can be partitioned into groups & domains**
  - Each *domain* can have its own administrator
    - HR can manage users
    - Individual departments can manage printers
- **Trust can be inherited in one or both directions**
  - ***department resources*** domains may trust the ***user*** domain
  - ***user*** domain may not trust ***department resources*** domains

# What about directories?

- **Directories are just files that map names to inode numbers**
- **Permissions have special meaning**
  - Write = permission to create a file in the directory
  - Read = permission to list the contents of a directory
  - Execute = permission to search through the directory

- **If you have *write* access to the directory of a file, you can *delete* the file**
  - Even if you don't have write access to the file itself
- **If you don't have *write* access to the directory**
  - You cannot *create* or *delete* a file … even if you have *write* access to it

# Changing permissions

## The **chmod** command

user = read, write, execute
group = read, execute
other = -none-

- Set permissions

```
$ chmod u=rwx,g=rx,o= testfile
$ ls -l testfile
-rwxr-x---  1 paul  localaccounts  6 Jan 30 10:37 testfile
```

- Add permissions

```
$ chmod go+w testfile
$ ls -l testfile
-rwxrwx-w-  1 paul  localaccounts  6 Jan 30 10:37 testfile
```

- Remove permissions

```
$ chmod o-w testfile
$ ls -l testfile
-r-xrwx---  1 paul  localaccounts  6 Jan 30 10:37 testfile
```

# Changing permissions

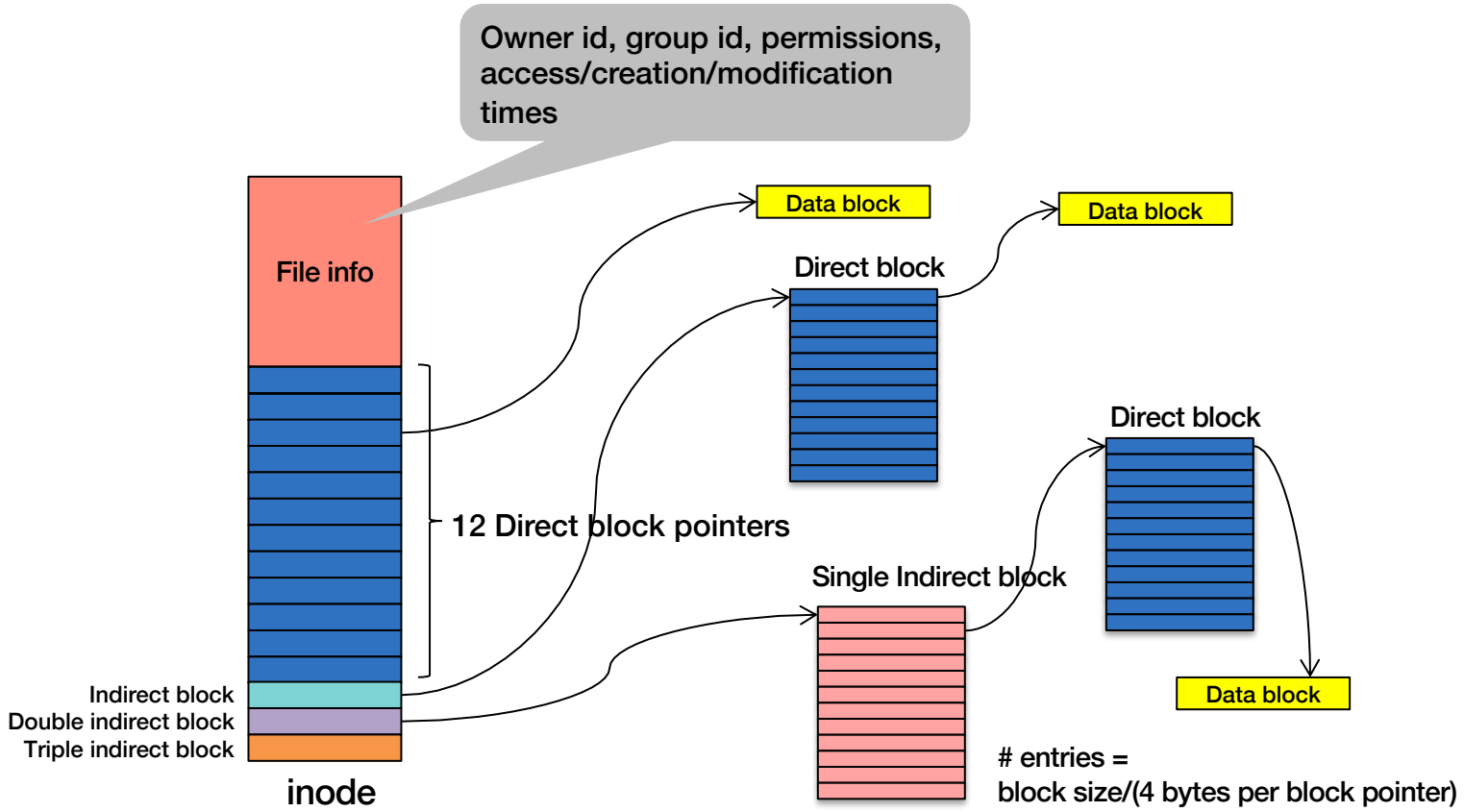
**Or the old-fashioned way – specify an octal bitmask**

- **Set permissions**

```
$ chmod 754 testfile
$ ls -l testfile
-rwxr-xr--  1 paul  localaccounts  6 Jan 30 10:37 testfile
```

| 7 | 5 | 4 |
|---|---|---|
| 111 | 101 | 100 |
| rwx | r-x | r-- |
| user | group | other |

# File permissions are stored in the file's inode



# Sometimes groups aren't enough

## Access Control Lists (ACL)

- **Explicit list of permissions for users**
- **Supported by most operating systems**
  - Windows ≥ XP
  - macOS ≥ 10.4
  - Linux ≥ ext3 file system + acl package

# Example: Full ACLs in POSIX systems

*What if we want to use a full ACL?*

- **Extended attributes: stored outside of the inode**
  - Hold an ACL
  - And other name:value attributes
- **Enumerated list of permissions on users and groups**
  - Operations on all objects:
    - *delete, readattr, writeattr, readextattr, writeextattr, readsecurity, writesecurity, chown*
  - Operations on directories
    - *list, search, add_file, add_subdirectory, delete_child*
  - Operations on files
    - *read, write, append, execute*
  - Inheritance controls

# ACLs and ACEs

**Access Control List (ACL) = list of Access Control Entries (ACE)**

- **ACE identifies a user or group & permissions**
  - Files: read, write, execute, append
  - Directories:
    list, search, read attributes, add file, add sub-directory, delete contents
- **"*Inheritance*" permission**
  - Files and directories can inherit ACL entries from the parent
- **Wildcards are often supported**
- **See *chmod* on macOS or *setfacl* on Linux**

# Example ACL

```
pxk.*          rwx
419-ta.*       rwx
*.faculty      rx
*.*            x
```

- Users `pxk` and `419-ta` have *read-write-execute* access
- Users in the faculty group have *read-execute* access
- Others only have *execute* access

# Search order

**ACEs are evaluated in the order they are entered into the ACL**

**In this case, I don't have write access to the file:**

```
419-ta.*        rwx
*.faculty       rx      ← This is me   ← This appears first & has priority
pxk.*           rwx     ← So is this
*.*             x       ← So is this
```

# Search order: ACLs + permissions

**In systems like Linux that integrate ACLs with 9-bit permissions:**

1. If you are the owner of the file, ***only*** owner permissions apply
2. If you are part of a group the file belongs to, only group permissions apply
3. Else search through the ACL entries to find an applicable entry
4. Else other permissions apply

# macOS Examples

# macOS ACL examples (1)

- **Create a file**

```
$ echo hello > hi.txt
$ cat hi.txt
hello
```

- **List the file**
  - Show ACEs with -e option to ls

```
$ ls -l hi.txt
-rw-r--r--  1 paul  wheel  6 Sep 13 23:01 hi.txt
$ ls -le hi.txt
-rw-r--r--  1 paul  wheel  6 Sep 13 23:01 hi.txt
```

**No ACL!**

# macOS ACL examples (2)

- **Take away read & write access**
  - Add an access control entry with `chmod +a`
  - Remove an access control entry with chmod –a

```
$ chmod +a "paul deny read,write" hi.txt
```

- **See what we have**

```
$ ls -le hi.txt
-rw-r--r--+ 1 paul  wheel  6 Sep 13 23:01 hi.txt
 0: user:paul deny read,write
```

ACL

- **Add append access**

```
$ chmod +a "paul allow append" hi.txt
$ ls -le hi.txt
-rw-r--r--+ 1 paul  wheel  6 Sep 13 23:01 hi.txt
 0: user:paul deny read,write
 1: user:paul allow append
```

ACL

# macOS ACL examples (3)

- Try reading and writing to the file

```
$ echo "new data" >hi.txt
bash: hi.txt: Permission denied
$ cat hi.txt
cat: hi.txt: Permission denied
```

- But we can append

```
$ echo "appended data" >>hi.txt
$ ls -l hi.txt
-rw-r--r--+ 1 paul  wheel  20 Sep 13 23:16 hi.txt
```

- Useful for granting users append-only access to a log file

It's bigger: 20 bytes vs. 6

# macOS ACL examples (4)

- **Remove Access Control Entry #0**

```
$ ls -le hi.txt
-rw-r--r--+ 1 paul  wheel  20 Sep 13 23:16 hi.txt
 0: user:paul deny read,write
 1: user:paul allow append
$ chmod -a# 0 hi.txt
$ ls -le hi.txt
-rw-r--r--+ 1 paul  wheel  20 Sep 13 23:16 hi.txt
 0: user:paul allow append
```

`chmod -a# N` **removes rule *N***

**The "deny read, write" entry is gone**

- **Now we can see the file**

```
$ cat hi.txt
hello
appended data
```

# Changing Permissions

# Initial file permissions

**On Unix-derived systems (Linux, macOS, Android, *BSD):**

- **umask = set of permissions applications cannot set on files**
  - Bitmask (octal) of bits that will be turned off
- **To disallow *read-write-execute* for everyone but the owner**
  - `umask` = 000 111 111 = 077
- **Default umask on macOS & Ubuntu is 022**
  - `022` = `000 010 010` = `--- -w- -w-`
  - This takes away ***write*** access from group & other
  - By default, new files are readable by all and writable only by the owner

**See the *umask* command and *umask* system call man pages**

# Watch out for race conditions!

```
#!/bin/bash
myapp >secretfile
chmod go-r secretfile
```

Suppose we create a file readable by all: `rwxr--r--`

`rwx, r, r`

- And then we change the permissions to `rwx------`

`rwx, -, -`

| GOOD | BAD |
|---|---|
| Create a file: `rwx-r--r-` | Create a file: `rwx-r--r-` |
| Change permissions to `rwx------` | *[Attacker opens the file for reading]* |
| *[Attacker opens the file for reading]* | Change permissions to `rwx------` |
| *Do your work* | *Do your work* |

- We don't know when the attacker will hit
- Once the attacker has the file open, changing permissions does not take access away
  - Access rights are only checked when the file is opened!

# Giving files away

- **You can change the owner of a file**

```
chown alice testfile
```

  - Changes the file's owner to alice

- **You can change the group of a file too**

```
chgrp accounting testfile
```

  - Changes the file's group to accounting

**… but you have to be the owner to do either**

# Changing user & group IDs

- **root = uid 0 = super user**
  - Access to everything
- **How do you log in?**
  - login program runs as uid=0
  - Gets your credentials
  - Authenticates you
  - Then:

```
chdir(home_directory);
setgid(group_id);
setuid(user_id);
execve(user_shell, …);
```

# Changing user ID temporarily

- **What if some files need special access?**
  - A print program needs to access the printer queue
  - A database needs to access its underlying files
- **An executable file normally runs under the user's ID**
- **A special permission bit, the "setuid bit" changes this**
  - Executable files with the setuid bit
    will run with the *effective UID* set to the owner of the file
  - Directories with the setuid bit set
    will force all files and sub-directories created in them to be owned by the directory owner
- **Same thing with groups – the setgid permission bit**
  - Executable files with this bit set will run with effective gid set to the gid of the file.

# Principle of Least Privilege

At each abstraction layer, every element (user, process, function) should be able to access ***only*** the resources necessary to perform its task

**Even if an element is compromised, the scope of damage is limited**

**Consider:**

- **Good:** You cannot kill another user's process
- **Good:** You cannot open the `/etc/hosts` file for writing
- **Good:** Private member functions & local variables in functions limit scope

- **Violation:** a compromised print daemon allows someone to add users
- **Violation:** a process can write a file even though there is no need to
- **Violation:** admin privileges set by default for any user account

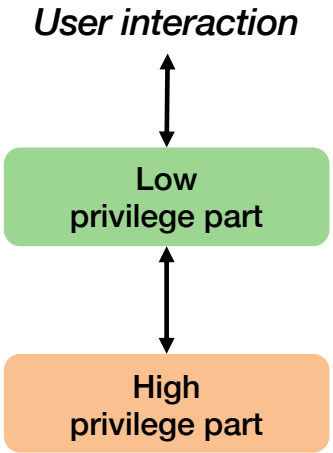
**Least privilege is often difficult to define & enforce**

# Privilege Separation

## Divide a program into multiple parts: high & low privilege components

### Example on POSIX systems

- Each process has a *real* and *effective* user ID
- Privileges are evaluated based on the effective user ID
  - Normally, `uid == euid`
- An executable file may be tagged with a *setuid bit*
  - **`chmod +sx filename`**
  - When run: uid = user's ID
    euid = file owner's ID (without setuid, runs with user's ID)
- Separating a program
  1. Run a *setuid* program
  2. Create a communication link to self (*pipe*, *socket*, shared memory)
  3. *fork*
  4. One of the processes will call `seteuid(getuid())` to lower its privilege



# Setuid can get you into trouble!

- Most *setuid* programs ran as root
- If they were compromised, the whole system was compromised
- This was one of the best attack vectors for Unix/Linux systems

# The End