

# Computer Security

## 02. Operating System Access Control

Paul Krzyzanowski

Rutgers University

Spring 2017

# Protection is essential to security

## Protection

- The mechanism that provides and enforces controlled access of resources to processes
- A protection mechanism *enforces* security policies
  
- User accounts & user authentication
  
- User privileges: access rights
  - File protection
  
- Resource scheduling
  - Priorities
  
- Quotas (sometimes)

# Co-located resources

- Early computers
  - Single-user batch processing
  - No need for access control
- Later ... timesharing systems
  - Multiple users share the same computer
  - User accounts & access control important
- Even later ... PCs
  - Back to single-user systems
  - ... but software is less trusted
- Now: PCs + mobile devices + IoT devices
  - + Shared access: cloud computing, file servers, university systems
  - + Program isolation on servers
    - Need to enforce **access control**

# Access control

---

- Ensure that authorized users can do what they are permitted to do ... *and no more*
- Real world
  - Keys, badges, guards, rules
- Computer world
  - Hardware
  - Operating systems
  - Web servers, databases & other multi-access software
  - Policies

# OS controls access to resources

---

- CPU
- Memory
- Files & devices
- Network

# Fundamental Mechanisms

---

- Protect the operating system from applications
- Protect applications from each other
- Allow the OS to stay in control

# Hardware timer

- OS kernel requests timer interrupts
  - One of several timer devices:
    - Programmable Interval Timer (PIT)
    - HPET (High Precision Event Timer),  
or APIC timer (one per CPU)
  - Most current Intel Linux systems use APIC
- Applications cannot disable this
- Ensures that the OS can always regain control

# Processes

- Timer interrupts ensure OS can examine processes while they are running
- OS Scheduler
  - Decides whether a process had enough CPU time and it is time for another process to run
  - Avoid **starvation**: ensure all processes will get a chance to run
    - This would be an **availability** attack
  - Prioritize threads
    - Based on user, user-defined priorities, interactivity, deadlines, “fairness”
    - One process should not adversely affect others

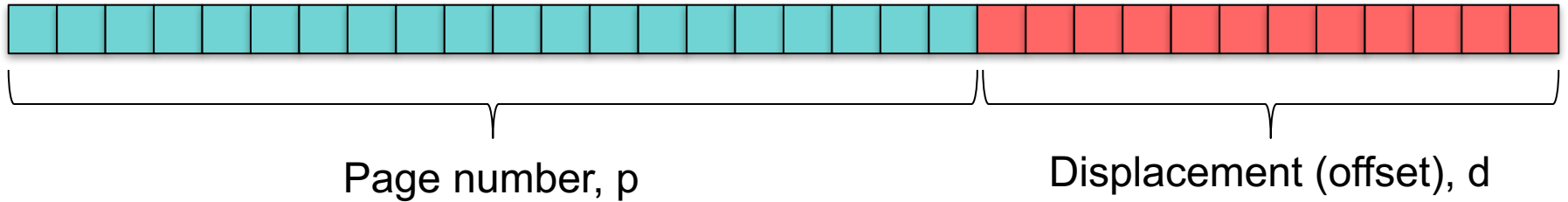


# Memory Management Unit

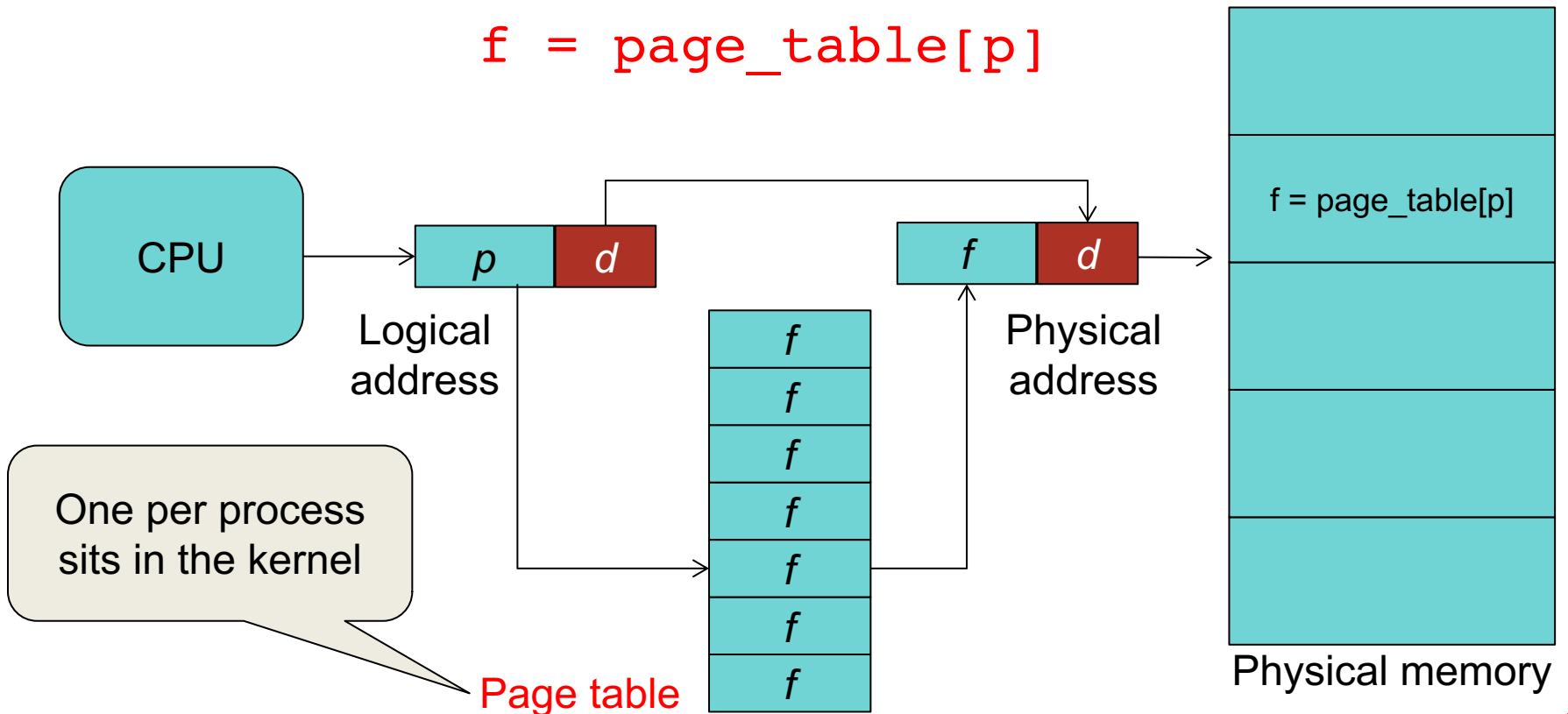
- All modern CPUs have a **Memory Management Unit (MMU)**
- OS provides each process with **virtual memory**
  - Gives each process the illusion that it has the entire address space
  - One process cannot see another process' address space
  - Enforce access rights
    - Read-only
    - Read-write
    - Execute

# Page translation

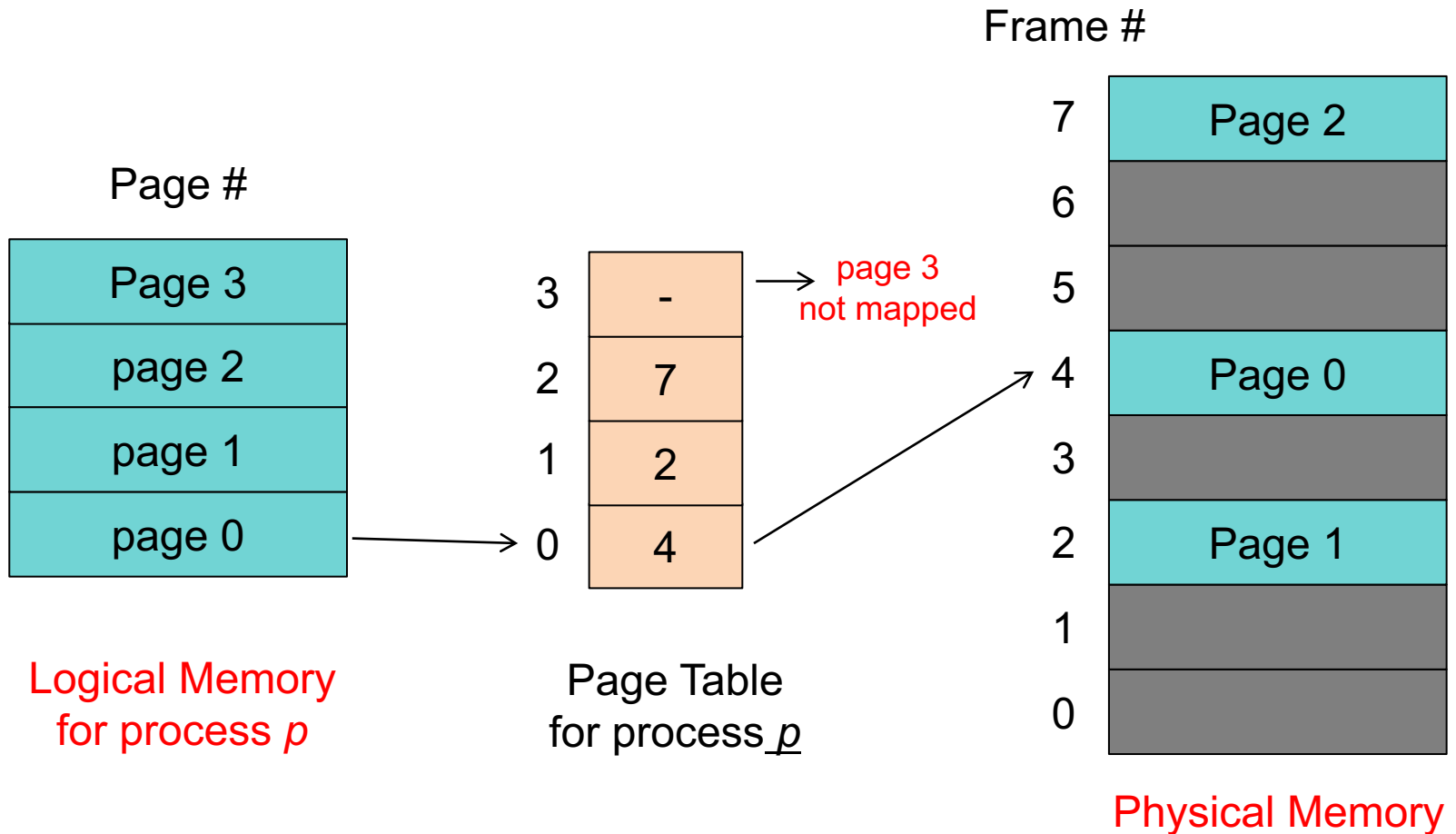
Virtual memory address



$$f = \text{page\_table}[p]$$



# Logical vs. physical views of memory



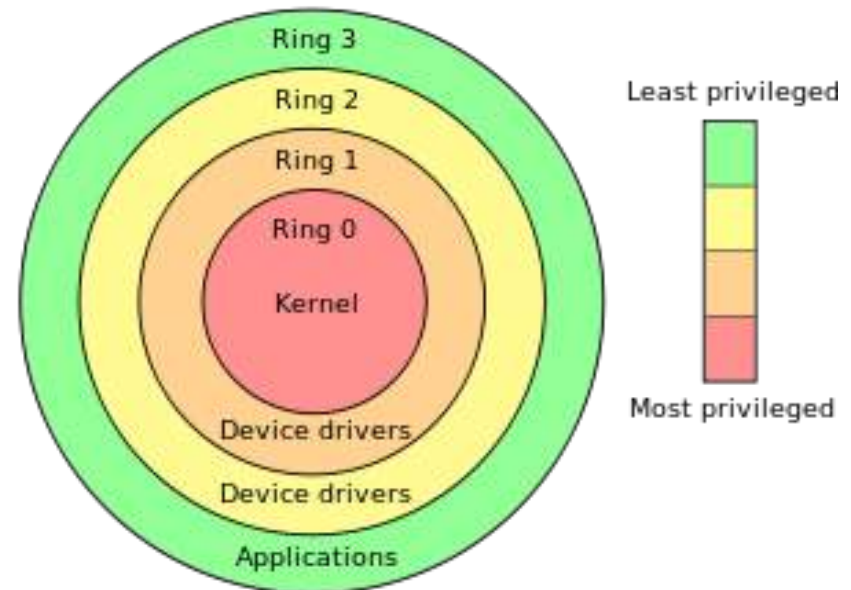
# User & kernel mode

**Kernel mode** = privileged, system, or supervisor mode

- Access restricted regions of memory
  - Modify the memory management unit (page tables)
  - Set timers
  - Define interrupt vectors
  - Halt the processor
  - Etc.
- Getting into kernel mode
    - **Trap**: explicit instruction
      - Intel architecture: *INT* instruction (interrupt)
      - ARM architecture: *SWI* instruction (software interrupt)
    - **Violation**
    - Hardware **interrupt** (e.g., receipt of network data or timer)

# Rings

- All modern operating systems use two modes or operation: **user & kernel**
- Multics defined a ring structure with 6 different privilege levels
  - Each ring is protected from higher numbered rings
  - Special call (**call gates**) to cross rings
  - Most of OS did not run in ring 0
- Intel x86 supports 4 rings
- Today's OSes only use
  - Ring 0: kernel
  - Ring 3: user



[https://en.wikipedia.org/wiki/Protection\\_ring](https://en.wikipedia.org/wiki/Protection_ring)

# User authentication

---

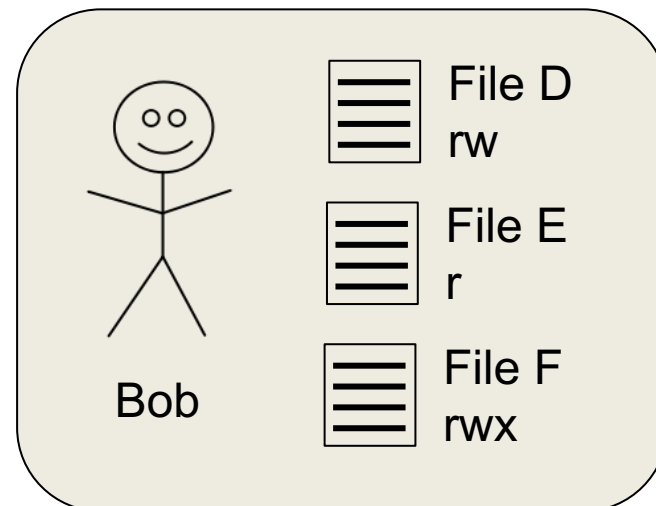
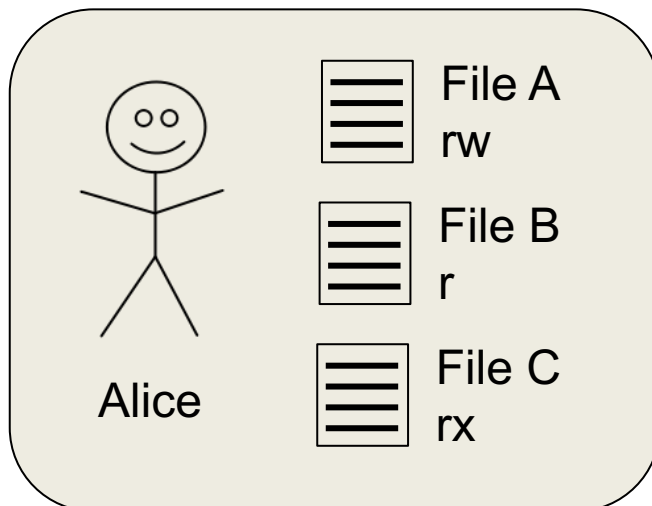
- Establish user identity
- Operating system privileges are granted by user identity
- Steps
  1. Get user credentials (e.g., name, password)
  2. Authenticate user by validating the credentials
  3. Grant further access based on user ID

# Subjects and Objects

- **Subject**: the thing that needs to access resources
  - Often the user
- **Object**: the resource the subject may access
- **Access control**
  - Define what different subjects are allowed to do  
or  
define what can be done to different objects
- Traditional operating systems:
  - Define what can be done to different objects  
(permissions are associated with each object)

# File permissions

- Access isn't all or nothing
- Objects can have different access permissions
- UNIX permission model
  - Read, write, execute access
    - All independent
  - Each file has an owner





# 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
- File access permissions are expressed as:

```
rwxrwxrwx
┌──┬──┬──┐
  user  group  other
```

```
$ ls -l /bin/ls
-rwxr-xr-x 1 root  wheel  38624 Dec 10 04:04 /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

# Windows

- Windows has more permissions
  - Read, write, execute
  - Also: delete, change permission, change ownership
- Users & resources can be partitioned into **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 contain **names** and **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

# Where are user IDs and group IDs stored?

On Linux, user ID information in the password file, [/etc/passwd](#)

- (which does not contain passwords!)

```
root:x:0:0:System Administrator:/root:/bin/sh
```

- User name
  - User ID
  - Default group ID
  - User's full name
  - Home directory
  - Login shell
- Group IDs are stored [/etc/group](#)
    - wheel:x:0:root
    - certusers:x:29:root,\_jabber,\_postfix,\_cyrus,\_calendar,\_dovecot

# Changing permissions

The **chmod** command

- 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

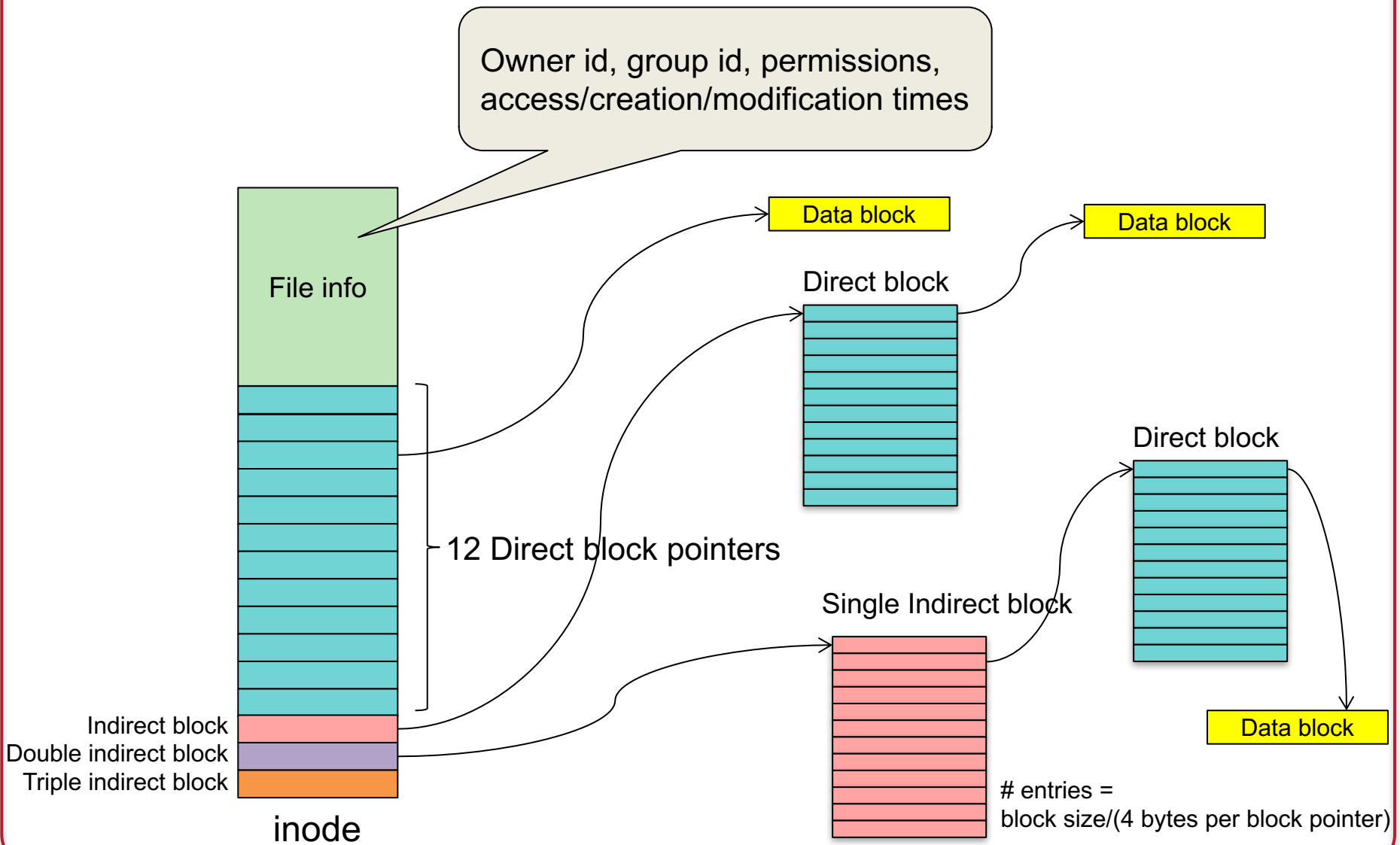
```
$ chmod 754 testfile
```

```
$ ls -l testfile
```

```
-rwxr-xr--  1 paul  localaccounts  6 Jan 30 10:37 testfile
```



# File permissions live in the inode



# Sometimes groups aren't enough

- Access Control Lists (ACL)
- Explicit list of permissions for users
- Supported by most operating systems
  - Windows  $\geq$  XP
  - macOS  $\geq$  10.4
  - Linux  $\geq$  ext3 file system + acl package

# 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
  - directory's file contents can inherit one set of ACLs
  - Directories inherit another set of ACLs
- 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
```

```
pxk.*      rwx
```

```
*.*        x
```

# 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. Search through the ACL entries to find an applicable entry
4. else other permissions apply

# 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!

- Suppose we create a file readable by all:  
rwx, r, r
- But then we change the permissions to rwx, -, -

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

Create a file: rwx-r--r-  
*[Attacker opens the file for reading]*  
Change permissions to rwx-----  
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, argv, envp);
```

# Changing user ID temporarily

- What if some programs need special access?
  - A network program needs to create raw sockets
  - 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 user ID* 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 group ID* set to the group ID 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

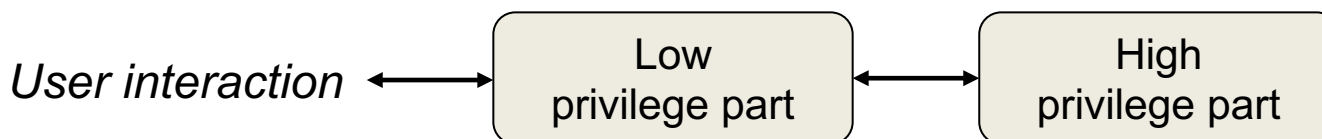
Least privilege is often difficult to define & enforce

# 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:  $uid = \text{user's ID}$   
 $euid = \text{file owner's ID (without setuid, runs with user's ID)}$
- 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/Solaris/Linux systems

# Domains of protection

- **Subjects** (users, processes) interact with **objects**
  - Objects include:
    - hardware (CPU, memory, I/O devices)
    - software: files, processes, semaphores, messages, signals
- A process should be allowed to access only objects that it is authorized to access
  - A process operates in a **protection domain**
  - Protection domain defines the objects the process may access and how it may access them

# Modeling Protection: Access Control Matrix

Rows: domains (subjects or groups of subjects)

Columns: objects

Each entry represents an access right of a domain on an object

*objects*

	<b>F<sub>0</sub></b>	<b>F<sub>1</sub></b>	<b>Printer</b>	
<i>domains of protection (subjects)</i>	D <sub>0</sub>	read	read-write	print
D <sub>1</sub>	read-write- execute	read		
D <sub>2</sub>	read- execute			
D <sub>3</sub>		read	print	
D <sub>4</sub>			print	

An Access Control Matrix is the primary abstraction for protection in computer security



# Access Transitions

Change the state of the system

- Expressed in commands
  1. **Command**
  2. Possible **condition**
  3. One or more ***primitive operations***
- **Condition** = ownership, ability to copy object, ...
- **Primitive operations** =
  - Create subject
  - Create object
  - Enter right  $r$  into  $a[s, o]$
  - Delete right  $r$  from  $a[s, o]$
  - Destroy subject  $s$
  - Destroy object  $r$

# Access Transitions

## Example

- Linux process  $p$  creates file  $f$  with owner read/write permissions
- **command**  $createfile(euid(p), f)$ 
  - create** object  $f$
  - enter** “ $own$ ” into  $a[euid(p), f]$
  - enter** “ $read$ ” into  $a[euid(p), f]$
  - enter** “ $write$ ” into  $a[euid(p), f]$

		<b><math>f</math></b>	
<i>uid</i>		own read write	

# Access Control Matrix: Domain Transfers

Switching from one domain to another is a configurable policy

A process in  $D_0$  can switch to running in domain  $D_1$

*objects*

*domains of protection*

	$F_0$	$F_1$	Printer	$D_0$	$D_1$	$D_2$	$D_3$	$D_4$
$D_0$	read	read-write	print	–	switch	switch		
$D_1$	read-write-execute	read			–			
$D_2$	read-execute				switch	–		
$D_3$		read	print					
$D_4$			print					

# Access Matrix: Additional operations

## Copy: allow delegation of rights

- Copy a specific access right on an object from one domain to another
  - Rights may specify either a copy or a transfer of rights

*objects*

*domains of protection*

	F <sub>0</sub>	F <sub>1</sub>	Printer	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>
D <sub>0</sub>	read	read-write	print	–	switch	sv		
D <sub>1</sub>	read-write-execute	read*						
D <sub>2</sub>	read-execute				switch	–		
D <sub>3</sub>		read	print					
D <sub>4</sub>			print					

A process executing in  $D_1$  can give a *read* right on  $F_1$  to another domain

# Access Matrix: Additional operations

**Owner:** allow new rights to be added or removed

- An object may be identified as being *owned* by the domain
- Owner can add and remove any right in any **column** of the object

*objects*

	F <sub>0</sub>	F <sub>1</sub>	Printer	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>
<i>domains of protection</i>	D <sub>0</sub>	read <b>owner</b>	read-write	print	–	switch	sw	
	D <sub>1</sub>	read-write-execute	read*		–			
	D <sub>2</sub>	read-execute			switch			
	D <sub>3</sub>		read	print				
	D <sub>4</sub>			print				

A process executing in  $D_0$  can give a *read* right on  $F_0$  to domain  $D_3$  and remove the *execute* right from  $D_1$

# Access Matrix: Additional operations

**Control:** change entries in a row

- If  $access(i, j)$  includes a *control* right, then a process executing in Domain  $i$  can change access rights for Domain  $j$

*objects*

	$F_0$	$F_1$	Printer	$D_0$	$D_1$	$D_2$	$D_3$	$D_4$
<i>domains of protection</i>	$D_0$	read owner	read- write	print	–	switch	swtich	
	$D_1$	read- write- execute	read*		–			control
	$D_2$	read- execute			switch	–		
	$D_3$		read	print				
	$D_4$			print				

A process executing in  $D_1$  can modify any rights in domain  $D_4$

# Implementing an access matrix

- A single table is usually impractical
  - Big size: # domains (users) × # objects (files)
  - Objects may come and go frequently
- **Access Control List**
  - Associate a column of the table with each object

# Implementing an access matrix

## Access Control List

- Associate a column of the table with each object

*domains of protection*

*objects*

	F <sub>0</sub>	F <sub>1</sub>	Printer	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>
D <sub>0</sub>	read owner	read- write	print					
D <sub>1</sub>	read- write- execute	read*			–			
D <sub>2</sub>	read- execute				switch	–		
D <sub>3</sub>		read	print					
D <sub>4</sub>			print					

ACL for file F<sub>0</sub>



# Example: Limited ACLs in POSIX systems

Problem: an ACL takes up a varying amount of space

- Won't fit in an inode

UNIX Compromise:

- A file defines access rights for three domains:
  - the **owner**, the **group**, and **everyone** else
- Permissions
  - Read, write, execute, directory search
  - Set user ID on execution
  - Set group ID on execution
- Default permissions set by the **umask** system call
- **chown** system call changes the object's owner
- **chgrp** system call changes the object's owner
- **chmod** system call changes the object's permissions

# Example: Full ACLs in POSIX systems

- *What if we really want 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

# Implementing an access control matrix

## Capability List

- Associate a row of the table with each domain

*objects*

*domains of protection*

	F <sub>0</sub>	F <sub>1</sub>	Printer	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>
D <sub>0</sub>	read owner	read- write	print	–	switch	swtich		
D <sub>1</sub>	read- write- execute	read*			–			
D <sub>2</sub>	read- execute				switch	–		
D <sub>3</sub>		read	print					
D <sub>4</sub>			print					

Capability list for domain D<sub>1</sub>

# Capability Lists

- List of objects together with the operations allowed on the objects
- Each item in the list is a *capability*: the operations allowed on a specific object
- A process presents the capability along with a request
  - Possessing the capability means that access is allowed
- A process cannot modify its capability list

# Capability Lists

- Advantages
  - Run-time checking is more efficient
  - Delegating rights is easy
- Disadvantages
  - Changing a file's permissions is hard
  - Hard to find all users that have access to a resource
- Limited implementation
  - Cambridge CAP, IBM AS/400
  - Not used in mainstream systems
- BUT
  - Used for single sign-on services and other authorization services such as OAuth and Kerberos (sort of)

# Windows Capabilities

---

- Added in Windows 2000
- Can override or complement ACLs
- Profiles permit whitelisting or blacklisting users
- Security policy defined by groups (Group Policy) for the entire system
  - Associated with sites, domains, or organizations

# What's wrong with ACLs?

- Users are in control

```
chmod o+rw secret.docx
```

- Now everyone can read and modify secret.docx
- Doesn't work well in environments where management needs to define access permissions
- No ability to give time-based or location-based permissions
- Access associated with objects
  - **Hard to turn off access for a subject** - except by locking the user
  - Otherwise have to go through each object and remove user from the ACL

# Access Control Models: MAC vs. DAC

- **DAC: Discretionary Access Control**
  - A subject (domain) can pass information onto **any** other subject
  - In some cases, access rights may be transferred  
e.g., chown
  - *Users are in charge of access permissions*
  - *Most systems use this*
  
- **MAC: Mandatory Access Control**
  - Policy is centrally controlled
  - Users cannot override the policy
  - *Administrators are in charge of access permissions*



# MLS: Multi-Level Security Systems

Handle multiple levels of classified data in one system

## Bell-LaPadula Model

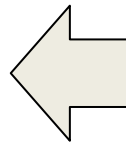
- Designed for the military
- Based on U.S. military classification levels

Top Secret

Secret

Confidential

Unclassified



### Motivation:

If one piece of software gets hacked, it will not have rights to do widespread damage by leaking data

If you have confidential clearance:

- You can access **confidential** & **unclassified** data
- You can create **confidential**, **secret**, and **top-secret** data

# Bell-LaPadula Base OS Model

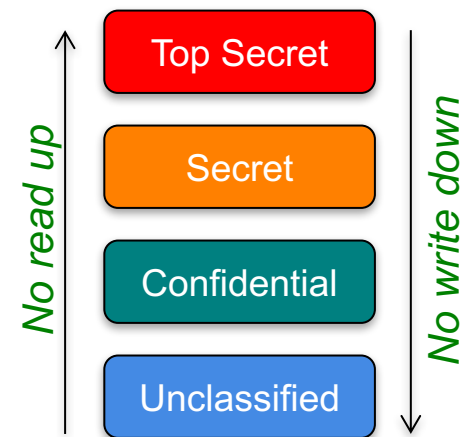
---

Designed to address security concerns in the Air Force

- **Reference Monitor**
  - Component of the OS that would manage access control decisions
- **Trusted Computing Base (TCB)**
  - Set of components whose correct functioning is sufficient to ensure the security policy is being enforced
  - If the TCB fails, the security policy could be breached

# Bell-LaPadula Access Model

- Objects are classified into a hierarchy of sensitivity levels
  - Unclassified, Confidential, Secret, Top Secret
- Each user is assigned a clearance
- “No read up; no write down”
  - Cannot read from a higher clearance level
  - Cannot write to a lower clearance level
- Assumes vulnerabilities exist and staff may be careless
- Need a “trusted subject” to declassify files



*Confidential cannot read Secret  
Confidential cannot write Unclassified*

# Bell-LaPadula Model Properties

Every subject & object gets a security **label** (e.g., confidential, secret)

## 1. The Simple Security Property

- No Read Up (NRU)  
A subject cannot read from a higher security level

## 2. \*-Property (star-property)

- No Write Down (NWD)  
A subject cannot write to a lower security level

## 3. The Discretionary Security Property

- Access control matrix can be used for **DAC** after MAC is enforced

# BLP Tranquility Principle

- **Tranquility principle**: security labels never change during operation
- **Weak tranquility principle**: labels may change but in a way that does not violate security policy
  - Implements the principle of least privilege
  - If owner has *Top Secret* clearance, a program will run at the lowest clearance level and get upgraded only when it needs to access data at a higher level
- BLP gets complicated
  - Changes in security policy in real time can result in access being revoked
    - even in the middle of an operation
- Difficult to use BLP in practice
  - Networking, servers, collaborative work

# Can *Write Up* cause problems?

- If you can write up, can a *Confidential* user overwrite *Secret* data?
  - That's an attack on availability
- Usually: allow overwriting files when the process' security labels match exactly
  - E.g., a user with *confidential* clearance can only overwrite a *confidential* file

# Type Enforcement Model (TE)

- Secondary Access Control Matrix that gives MAC priority over DAC
- Domains and Types
  - Assigns subjects to **domains**
  - Assigns objects to **types**
  - Matrix defines permitted **domain-domain** and **domain-type** interactions
- SE Linux = Security-Enhanced Linux
  - Both subjects and objects are types
  - Matrix defines allowed type pairs
  - Each process has a **security ID**, user ID, and group ID
  - Security modules may be added with rules that operate on SIDs

# Role-Based Access Control (RBAC)

- More general than Bell-LaPadula
- Designed to allow enforcement of both MAC & DAC properties
- Access decisions do not depend on user IDs but on **roles**
  - Administrators define **roles** for various job functions
  - Each role contains **permissions** to perform certain operations
  - Users are assigned one or more roles
- Roles are job functions, not permissions
  - “update customer information” is a role
  - “write to the database” is not a role
- Enables fine-grained access
  - Roles may be defined in application specific ways (e.g., “move funds”)



# RBAC Rules

---

- **Role assignment**
  - A subject can execute an operation only if the subject has been assigned a role
- **Role authorization**
  - A subject's active role must be authorized for that subject
  - Ensures that users can only take on roles for which they have been authorized
- **Transaction authorization**
  - A subject can execute a transaction only if the transaction is authorized through the subject's role membership

# Aren't roles == groups?

- **Group** = collection of users
  - Does not enable management of user-permission relationships
- **Role** = collection of permissions
  - Permissions can be associated with users and groups
- Roles have a **session**
  - Users can activate a role
- In SE Linux, RBAC is built on top of TE (type enforcement)
  - Users mapped to roles at login time
  - Roles are authorized for domains
  - Domains are given permissions to types

# RBAC Benefits

- RBAC is hugely popular in large companies
  - Driven by regulations such as HIPAA and Sarbanes-Oxley
- Makes it easy to manage movement of employees
- Makes it easy to manage “separation of duty” requirements
- Can manage complex relationships
  - Doctor X wants to view records of Patient Y
  - Doctor needs roles of “Doctor” and “attending doctor with respect to Y”
  - Roles allow specification of **only if**, **not if** or **if and only if** relations
- RBAC can simulate MAC and DAC

See <http://csrc.nist.gov/groups/SNS/rbac/faq.html>

# Biba Integrity Model

- Bell-LaPadula was designed to address confidentiality
- Biba is designed to ensure data integrity

Confidentiality = constraints on who can *read* data

Integrity = constraints on who can *write* data

- Biba model properties
  - A subject cannot read an object from a lower integrity level  
Subjects may not be corrupted by objects from a lower level  
(no read down)
  - Star integrity property = A subject cannot write to an object at a higher integrity level  
Subjects may not corrupt objects at a higher level than the subject  
(no write up)
  - A process cannot request higher access

# An example of where Biba is useful

The Biba model fits many real-world applications

- ECG device
  - Runs a calibration process, which stores a calibration file
  - And user processes, that run ECG tests
- Normal users cannot write the calibration file but can read it
  - Can read data at higher levels (calibration = higher data level)
    - User process can read calibration data – but cannot modify it
- Calibration process can write data to lower levels
  - Calibration process can write to the user process – but cannot read user data
- Works well when you need to get data from a trusted device

# Biba Problems

- Like Bell-LaPadula, it doesn't always fit the real world
- Microsoft implemented Mandatory Integrity Control
  - File objects are marked with an integrity level:
    - Critical files: *System*
    - Regular users and objects: *Medium*
    - Elevated users: *High*
    - Internet Explorer: *Low*
  - New process gets the **minimum** of the user's integrity level and the file integrity level
  - Default policy = *NoWriteUp*
    - Goal: Anything downloaded with IE can read files but cannot write them – limit damage from malware
    - Trusted subjects would have to overwrite the security model
      - Users get used to the pop-up dialog boxes asking for permission!
  - Microsoft dropped the *NoReadDown* restriction
    - Did not end up protecting the system from malicious content

# Access Models: Summary

- **Discretionary Access Control**
  - Works great when it's ok to put the user is in charge
- **Mandatory Access Control**
  - Needed when an organization needs to define policies
  - **Bell-LaPadula** (BLP)
    - Oldest & most widely studied model – synonymous with MLS
    - Designed to protect confidentiality
    - Doesn't work well outside of the DoD ... and is clunky within the DoD
  - **Type Enforcement** (TE)
    - Simple MAC model to override DAC
  - **Role-Based Access Control** (RBAC)
    - Identifies roles and assigns users to roles
    - Made popular
    - Most actively used MAC model
  - **Biba Model**
    - Opposite of Bell-LaPadula: concerned with integrity, not confidentiality

# Security Risks

- Even if the mechanisms work perfectly, policies may fail
  - DAC: you're trusting the users or a sysadmin to set everything up correctly
  - MAC
    - User or role assignment may be incorrect
    - Collaboration needs must be considered
    - Models like Bell-LaPadula and Biba require overrides to function well
- Corruption
  - Attacks may change the definition of roles or the mapping of users to roles
  - This is an attack on the Trusted Computing Base
- Users
  - Most malware is installed willingly
  - Users thus give it privileges of – at least – normal applications
  - As far as the operating system is concerned, it is enforcing defined policy



# Security Risks

- Even administrators should not be able to read all files
  - Many security systems enforce this
  - Edward Snowden should not have been able to copy sensitive documents onto a thumb drive ... even if NSA policy banned thumb drives
- General assumption has been that programs are trusted and run with the user's privileges
- Worked well for system programs
- Do you trust the game you installed on your phone?
- Need to consider better application isolation
  - Android turned Linux into a single-user system
  - User IDs are used on a per-application bases

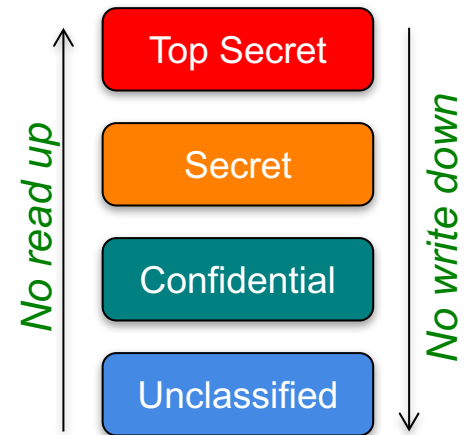
# Program-Based Control

- A lot of access decisions must be handled by programs, not the OS
  - Database users and the access each one has
  - Microsoft Exchange & Active Directory administrators
  - Mail readers
  - Web servers – especially those providing web services
  - Movement of data over a network
    - How do you send access permissions to another system?
    - Digital rights management = requires trusted players
- Programs may implement RBAC (e.g., Exchange) or other mechanisms
  - But the OS does not help

# Multi-Lateral Security

# Multi-Level Security

- Subjects and objects have assigned classification labels
- Rules control what you can read or write

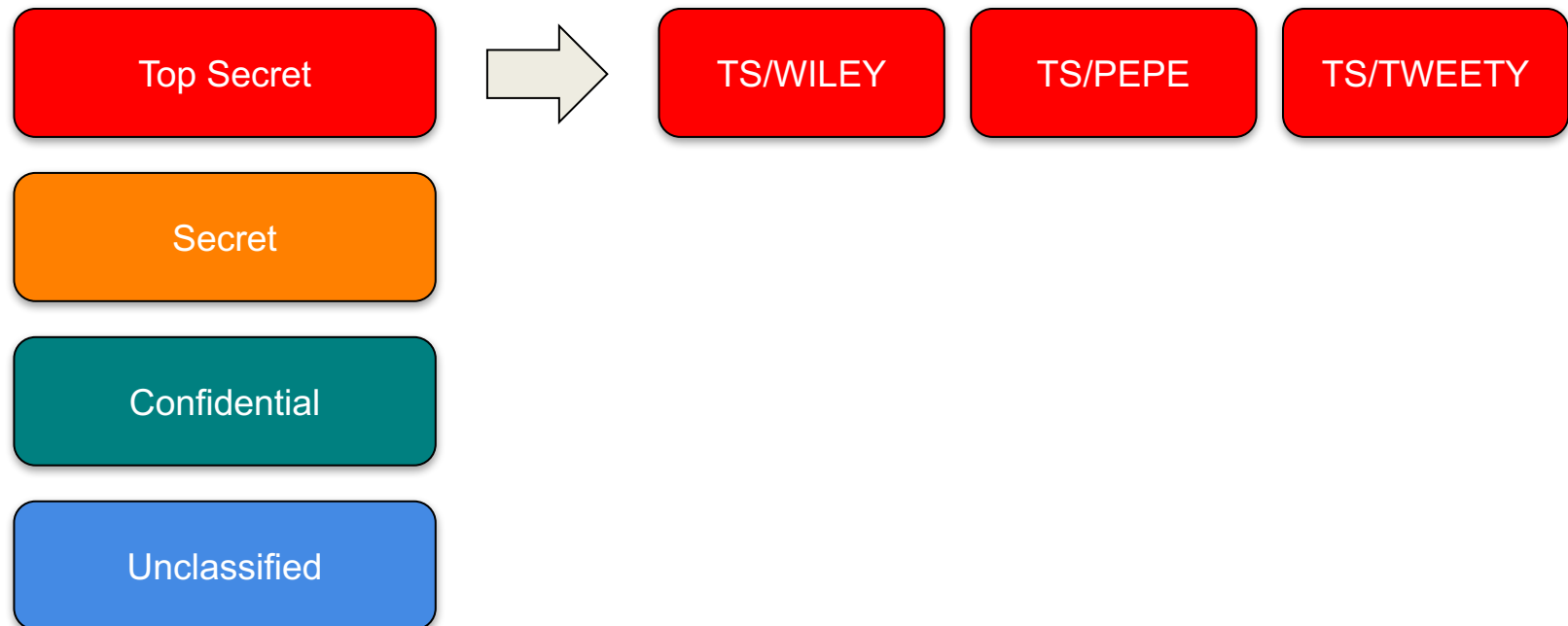


Bell-LaPadula

# Multilateral Security

Each security level may be divided into compartments

- Usually applied to the top secret level
- TS/SCI = Top Secret / Special Compartmentalized Intelligence
- You will be granted access to specific compartments
  - Formalized description of “need to know”

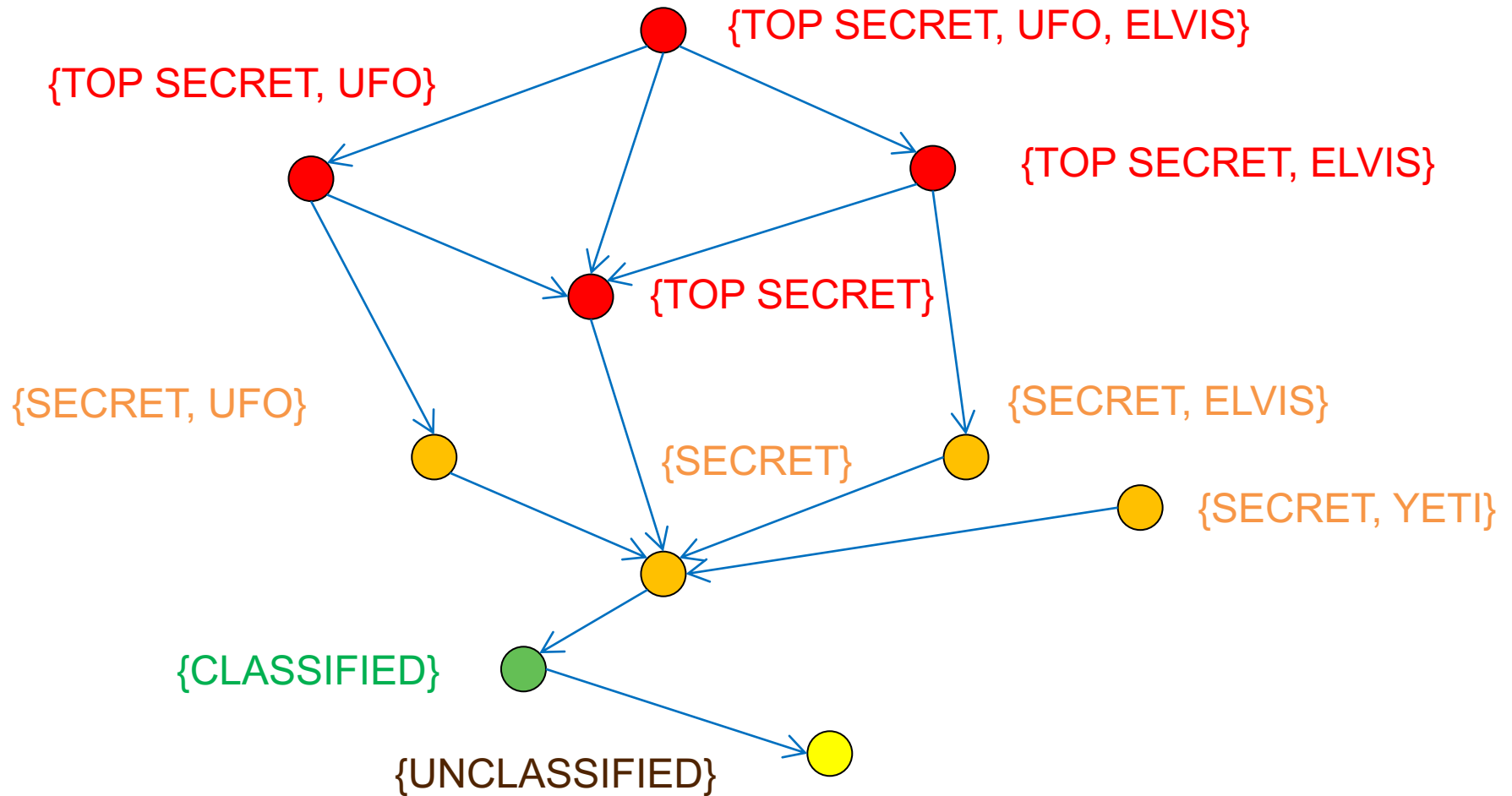


# Compartmentalization

- Subjects & objects get security labels (compartments) in addition to security classification labels
- If you do not have clearance for the label, you cannot access the data
  - {TOP SECRET, UFO} cannot be read by someone with {TOP SECRET} clearance
  - Neither can {SECRET, UFO}

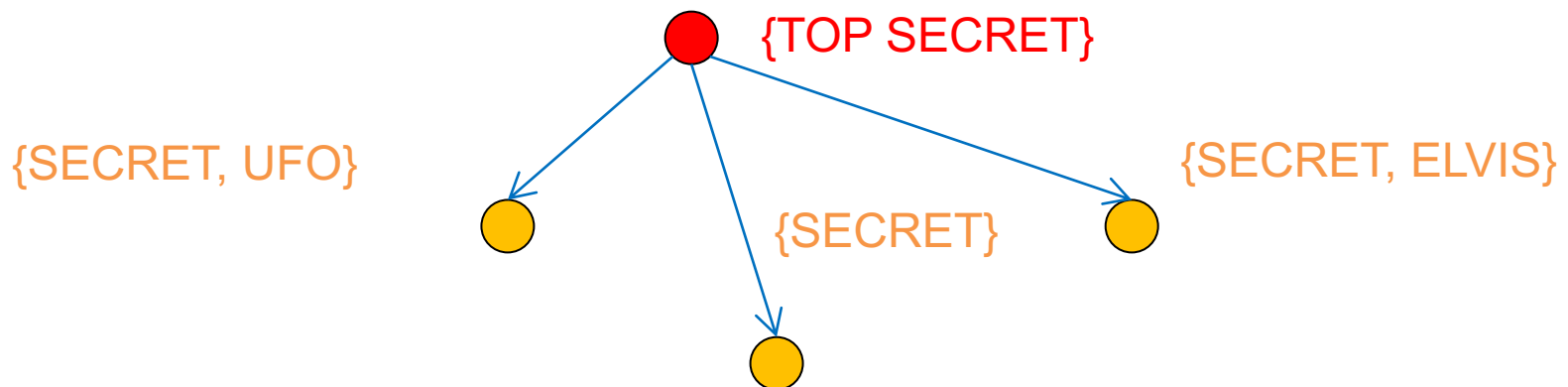
# Lattice Model

Graph representing access rights of different labels & levels



# Lattice model

- Data from two compartments => third compartment
  - Creates more isolation
  - Does not help with sharing
- One option
  - Allow multiple compartments at a lower level to be readable by a higher level





# Multi-level & Lattice models

---

- Do not help downgrading data
  - Need special roles to relabel or declassify data
- Handing searches across compartments is difficult
  - No single entity will likely have rights to everything

# Chinese Wall model

- **Chinese wall** = rules designed to prevent conflicts of interest
  - Common in financial industry
    - Separate corporate advisory & brokerage groups
  - Also in law firms and advertising agencies
- **Separation of duty**
  - A user can perform transaction *A* or *B* but not both
- Three layers of abstraction
  - **Objects**: files that contain resources about one company
  - **Company groups** = set of files relating to one company
  - **Conflict classes**: groups of competing company groups:
    - { Coca-cola, Pepsi }
    - { American Airlines, United, Delta, Alaska Air }

# Chinese Wall model

- **Basic rule**
  - A subject can access objects from a company as long as it never accessed objects from competing companies.
- **Simple Security property**
  - A subject  $s$  can be granted access to an object  $o$  only if the object
    - Is in the same company group as objects already accessed by  $s$
  - Or
    - $o$  belongs to a different conflict class
- **\*-property**
  - Write access is allowed only if
    - Access is permitted by the simple security property
  - And
    - No object can be read which is in a different company dataset than the one for which write access is requested  
*and* contains **unsanitized** information
      - **Sanitization** = disguising a company's identify

The end