CS165 – Computer Security

Access Control

March 7, 2024

Authentication and Access Control

- Authentication
  - Verifying the identity of a principal/subject
    - Passwords
    - Cryptography
    - E.g., User, process, host
- Access Control
  - Limit the accesses that a principal can perform
    - Access: Object and operation

Access Control

- Why do we need access control?

Access Control

- Why do we need access control?
  - Systems often run processes on behalf of multiple users or applications
  - May have objects with confidentiality, integrity, and/or availability concerns

Accidental Access

- What are we protecting data from?
  - Another user or application may run a process to accidentally overwrite, delete, leak your data
    - No reason for another user's errors to impact your data
- Access control can prevent another user or application from accessing your data

Malware

- What are we protecting data from?
  - Malware
  - Malicious code installed on your host may try to attack your system
    - Virus – modify binary files
    - Ransomware – encrypt data files
    - Trojan horse – steal passwords, contacts, photos, etc.
- Access control can confine malware to protect it from accessing/misusing your data

Compromised Processes

- What are we protecting data from?
  - Compromised processes
  - Adversaries may hijack a benign process
    - To exploit those permissions – advanced persistent threat
    - To escalate privileges through – local exploits
    - To compromise the host and spread – worm
- Access control can confine compromised processes to limit their impact

Protection System

- A protection system enforces access control policies
  - A program that enforces access control invokes the protection system to determine whether a subject can perform a security-sensitive operation
    - E.g., an operating system queries its protection system to determine whether a process running under a specific userid may write to a particular file
    - Lots of server programs enforce their own access control
  - The protection system checks whether the access control policy authorizes a subject (e.g., userid), object (e.g., file), and operation (e.g., write) combo

Access Matrix

- One way of viewing an access control policy is view an access matrix
  - Columns: Objects
  - Rows: Subjects
  - Cells: Operations (allowed)
- Shows:
  - Subj2 can read Obj2

	Obj1	Obj2	Obj3
Subj1	R	RWX	RW
Subj2		R	
Subj3		RW	

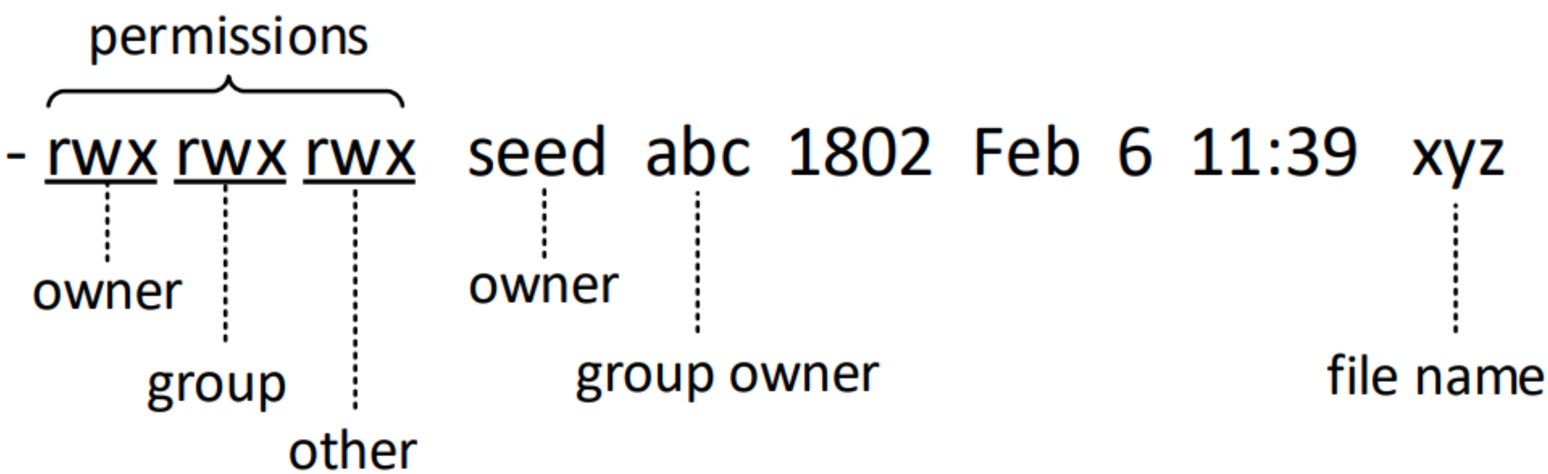
Access Matrix

- An access matrix can be interpreted from two perspectives
  - From the object's perspective
    - Access Control List (red)
    - Subjects that can access that particular object
  - From the subject's perspective
    - Capabilities (green)
    - Objects a subject can access

	Obj1	Obj2	Obj3
Subj1	R	RWX	RW
Subj2		R	
Subj3		RW	

UNIX File Permissions

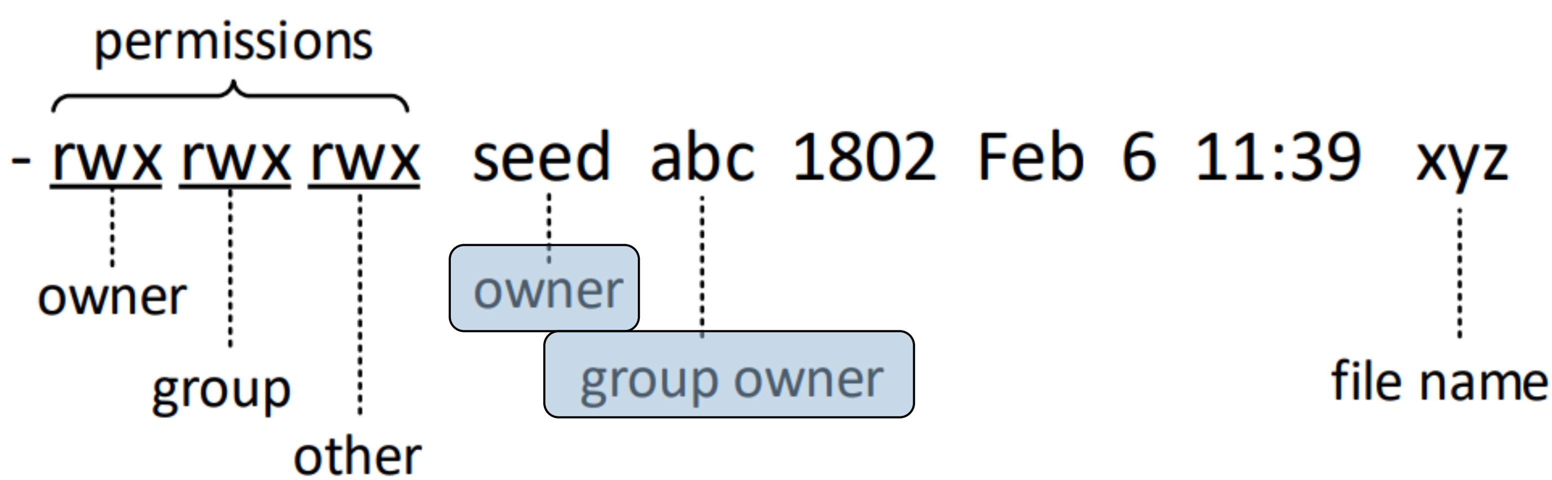
- Each file is assigned its own ACL encoding (called mode bits) of permissions to authorize subjects
  - Run `ls -la`



- What does all this mean?

UNIX File Permissions

- Each file has an owner and group owner, userids that have special permissions to a file



Users and Userids

- In Linux, each user is assigned a unique userid
- Userids are stored in /etc/passwd

```
root:x:0:0:root:/root:/bin/bash
seed:x:1000:1000:SEED,,,:/home/seed:/bin/bash
```

- Find a userid

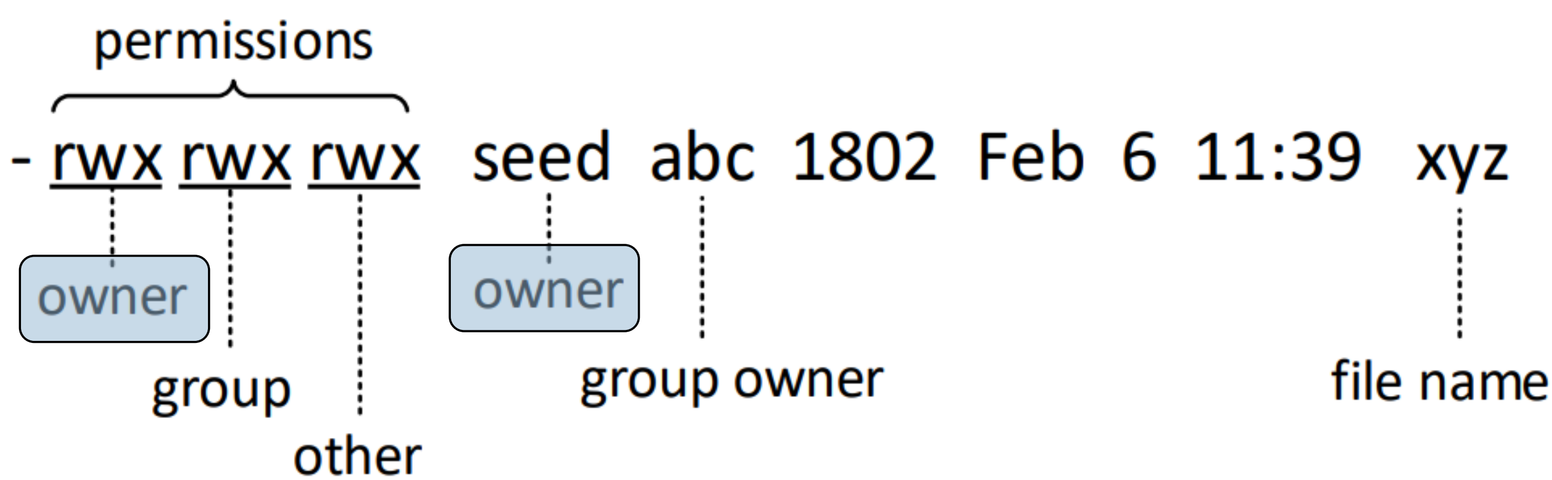
```
seed@VM:~$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed)

root@VM:~# id
uid=0(root) gid=0(root) groups=0(root)
```

- Each process run has a userid

UNIX File Permissions

- The permissions of a file designate the permissions of the file's owner
  - The only permissions are read (r), write (w), execute (x)



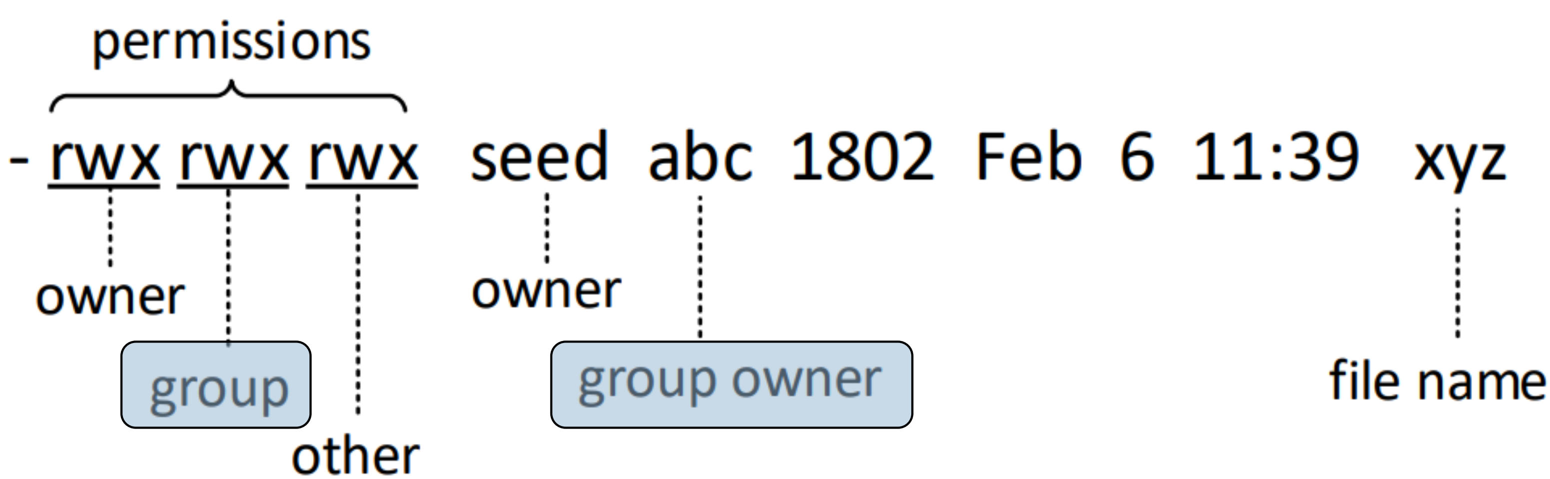
UNIX Groups

- Represents a group of userids
- Assigns permissions based on group
- A user can belong to multiple groups
- A user's primary group is in /etc/passwd

```
root:x:0:0:root:/root:/bin/bash
seed:x:1000:1000:SEED,,,:/home/seed:/bin/bash
bob:x:1001:1001:Bob,,,:/home/bob:/bin/bash
alice:x:1002:1003:Alice,,,:/home/alice:/bin/bash
```

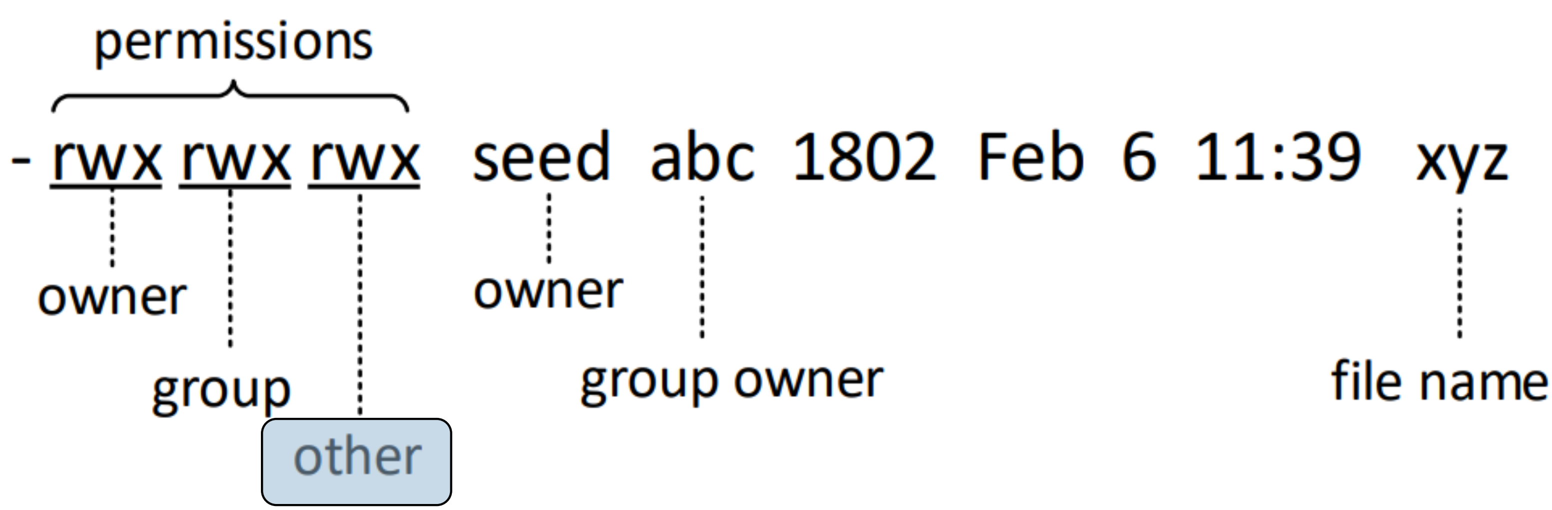
UNIX File Permissions

- The permissions of a file designate the permissions of the file's group owner too
  - A process may belong to multiple groups, so just need one to be the group owner of this file to get group



UNIX File Permissions

- What about users who are neither the file's owner nor a member of the file's group owner?
  - They are authorized using the other permissions



UNIX Permission Semantics

- Types of access on files
  - **read (r)**: user can view the contents of the file
  - **write (w)**: user can change the contents of the file
  - **execute (x)**: user can execute or run the file if it is a program or script
- Types of access on directories
  - **read (r)**: user can list the contents of the directory (e.g., files in the directory)
  - **write (w)**: user can create files and sub-directories inside the directory
  - **execute (x)**: user can enter that directory (e.g., using ‘cd’)

Default File Permissions

- umask: determines the default permissions assigned to new files
  - You probably live with these permission assignments

```
Initial (0666)          rw- rw- rw-
                        110 110 110
umask    (0022)         000 010 010
---------------------------------
Final permission        110 100 100
                        rw- r-- r--
```

Umask Example

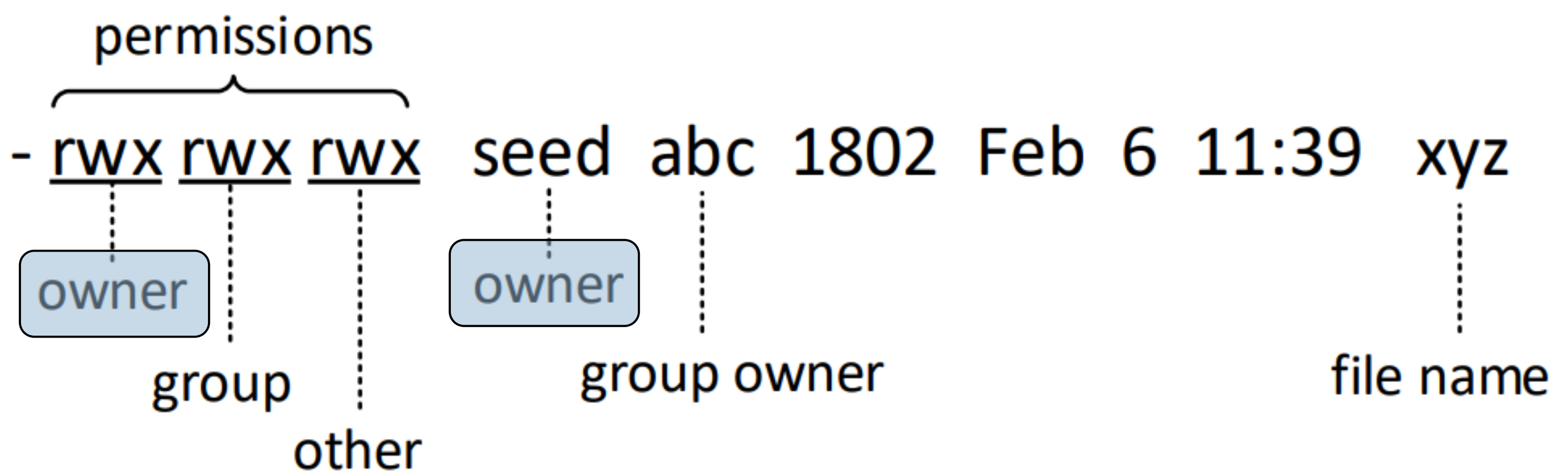
```
$ umask
0002
$ touch t1

$ umask 0022
$ touch t2
$ umask 0777
$ touch t3

$ ls -l t*
-rw-rw-r-- 1 seed seed 0 Feb  6 16:23 t1
-rw-r--r-- 1 seed seed 0 Feb  6 16:24 t2
---------- 1 seed seed 0 Feb  6 16:24 t3
```

UNIX File Permission Changes

- A file owner can change its permissions
  - chmod – change the mode bits value of a file
    - `chmod 644 xyz` or `chmod +r xyz`



File Descriptors

- After you are authorized to open a file, your process receives a form of a capability, called a file descriptor, to perform operations on the file

File Descriptors

- After you are authorized to open a file, your process receives a form of a capability, called a file descriptor, to perform operations on the file
- A file descriptor identifies the permissions that may be exercised on the file when presented on subsequent system calls (i.e., to the OS)
  - `write(fd, buffer, size)`
    - Allowed if the file descriptor `fd` has the write permission, based on opening the file read-write (`O_RDWR`)
- In a pure capability system, a file descriptor could be given to another process – more limited here

POSIX Capabilities

- Divide the root privilege into smaller privilege units
- Known as POSIX capabilities – not capabilities in the traditional sense
  - Just identifiers for sets of permissions
  - Use "man capabilities" to find all the capabilities

```
CAP_CHOWN:          Make arbitrary changes to file UIDs and GIDs.
CAP_DAC_OVERRIDE: Bypass file read/write/execute permission checks.
CAP_DAC_READ_SEARCH: Bypass file read permission checks ...
CAP_NET_RAW:        Use RAW and PACKET sockets ...
```

Does Access Control Ensure Security?

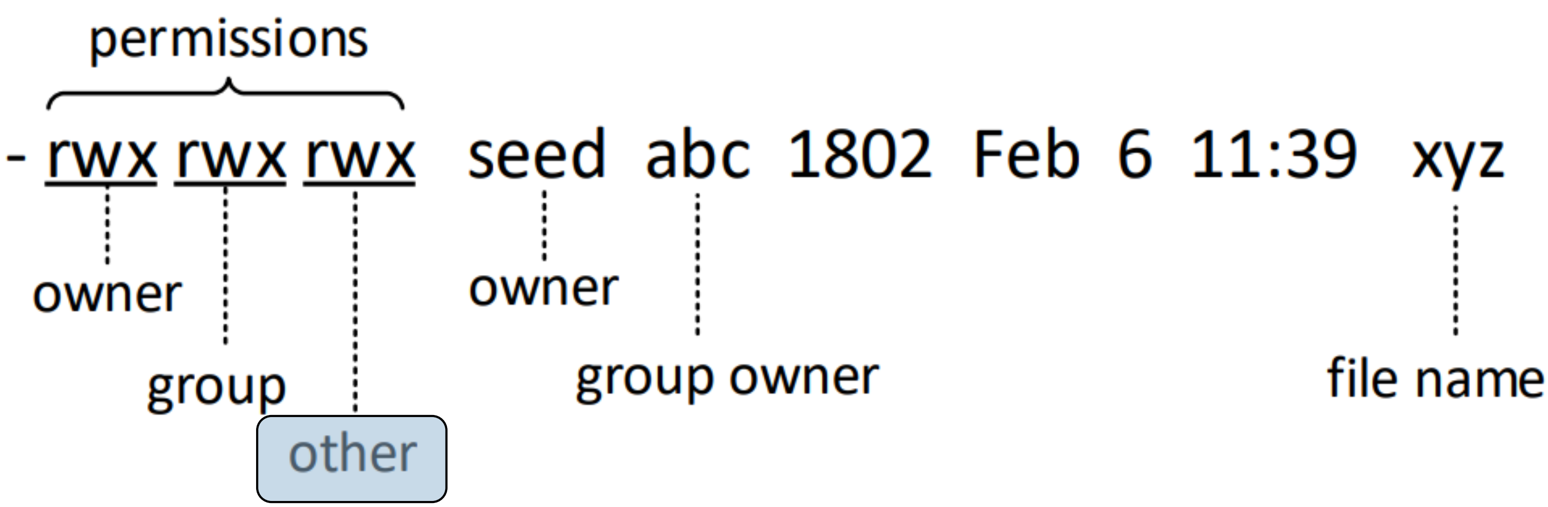
- E.g., Solve basic security problems
  - Can UNIX/Windows protection systems ensure that a particular permission is never granted to a particular subject?
  - Answer: No (proven in 1976)
- As a result, we cannot solve many security problems with UNIX protection systems
  - It can prevent accidents, but cannot enforce security

Can I Confine Malware?

- Can I define a UNIX policy that confines an untrusted program to no file access?

Can I Confine Malware?

- Can I define a UNIX policy that confines an untrusted program to no file access?
  - Answer: No
  - Remember "others" rights to files



- E.g., Malware can execute many programs (root)

Can I Prevent Secrets from Being Leaked?

- Can I define a UNIX policy that ensures that a process with access to a file cannot leak the file contents to another process?
  - The Trojan horse problem
    - You download a program and give it access to a secret file.
    - Can you ensure that the program does not leak the file?

Can I Prevent Secrets from Being Leaked?

- Can I define a UNIX policy that ensures that a process with access to a file cannot leak the file contents to another process?
  - The Trojan horse problem
  - Answer: No way
- A process can create an object (i.e., become the owner) and grant the other process read access

UNIX Defenses

- There are actually some ad hoc attempts to enable UNIX to enforce such policies
  - E.g., chroot
- But, they don't really work

Mandatory Access Control

- Consists of two goals
- (1) Provide a fixed (i.e., system-defined) access control policy to express security requirements
  - E.g., to confine processes and prevent leaks
- (2) Ensure that the access control policy is enforced correctly and comprehensively
  - To guarantee the policy enables its goals

Fixed Access Matrix

- Can still express policies as an access matrix
  - Columns: Objects
  - Rows: Subjects
  - Cells: Operations (allowed)
- But, what if the set of objects changes?
- But, what if a user runs multiple programs?
  - Trusted and untrusted

	Obj1	Obj2	Obj3
Subj1	R	RWX	RW
Subj2		R	
Subj3		RW	

Fixed Access Matrix

- But, what if the set of objects changes?
- But, what if a user runs multiple programs?
  - Trusted and untrusted
- Can fix both the same way
  - Use a fixed set of labels for subjects and objects
  - Subject labels are often program-specific (confine)

	Public	Httpd Objects	Httpd Code
httpd	R	RW	RX
sshd	R		
untrust	R		

Access Control for Security

- In practice, mandatory access control is used in two ways to express security requirements
- Least privilege
  - Confine malware
  - Confine compromised processes
    - In particular, network-facing daemons
- Multi-level Security (MLS)
  - Prevent leakage
    - Basically, a form of information flow

Least Privilege

- Only the permissions necessary to operate
  - ”Confine” by preventing use of unnecessary permissions
  - This idea is old (Multics: Saltzer & Schroeder 1975)
- How do we determine least privilege permissions for a program?

Least Privilege

- How do we determine least privilege permissions for a program?
  - Run the program
  - Log the permissions used
  - Grant only those permissions
- Linux program to do this called audit2allow

Issues with Least Privilege

- Did we find all the permissions that may be used?
  - Multiple runs
  - Multiple configurations
  - Not easy to find all uses – RHEL notes for Apache
    - https://www.serverlab.ca/tutorials/linux/web-servers-linux/configuring-selinux-policies-for-apache-web-servers/
- Did we ensure confinement of the process?
  - Access control is configured based on functionality
  - So, if the program uses a dangerous permission, an adversary may exploit that

Least Privilege and Confinement

- Suppose we run the web server and it creates files in the directory `/var/www/html`
  - Root web server directory
- And crond can execute scripts in `/var/www/html`
  - crond is a daemon to execute scheduled commands
  - crond runs as root and has a lot of uses/privileges
    - Hard to confine
- What can an adversary do?

Least Privilege and Confinement

- Suppose we run the web server and it creates files in the directory `/var/www/html`
  - Root web server directory
- And crond can execute scripts in `/var/www/html`
  - crond is a daemon to execute scheduled commands
  - crond runs as root and has a lot of uses/privileges
    - Hard to confine
- What can an adversary do?
  - Compromise the web server (network daemon) to inject code into `/var/www/html` for crond to run

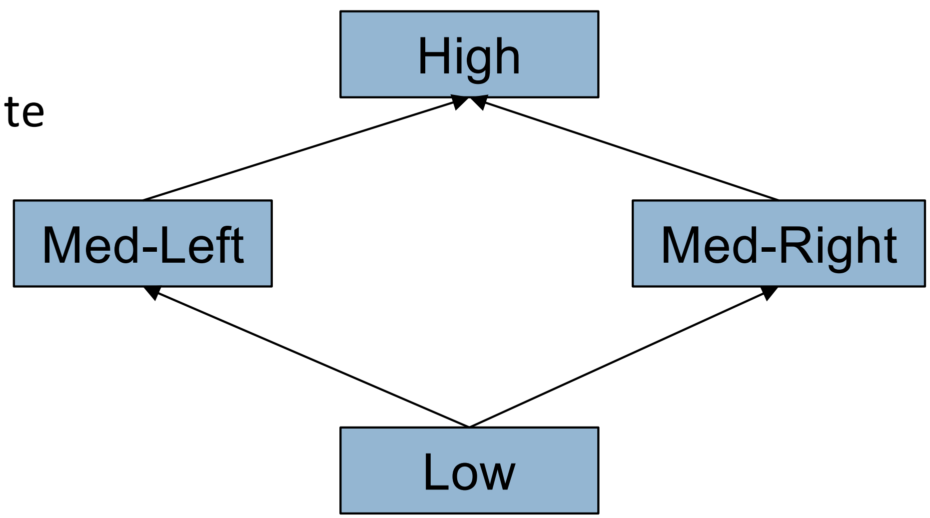
Preventing Leakage

- Classic Threat: Trojan horse
  - You download a program and give it access to a secret file.
    - The program may perform a valuable service, but also have additional function that is adversarial
  - Can you ensure that the program does not leak the file with mandatory access control? How?

Lattice Security Model (Info Flow)

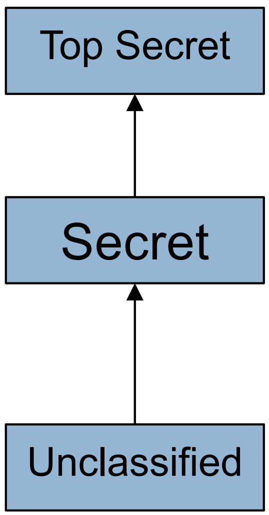
- Formalizes security based on information flow models
  - FM = {N, P, SC, /, >}
- Information flow model instances form a lattice
  - What's a lattice?
    - Graph where every node has a LUB and a GLB
- N are objects, P are processes, and each are assigned a security class SC
  - {SC, >} is a partial ordered set
  - SC, the set of security classes, is finite
  - SC has a lower bound
  - and / is a LUB operator

Subjects and Objects Are Assigned SCs



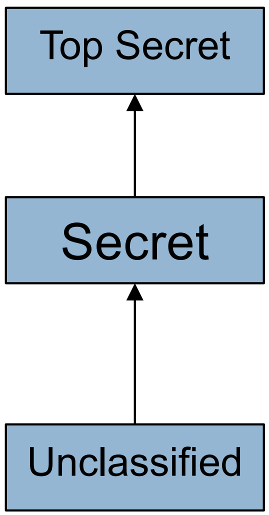
Multi-Level Security (MLS)

- An operation is only authorized if:
  - Read: $SC_{Subject} >= SC_{Object}$
  - Write: $SC_{Subject} <= SC_{Object}$
- To ensure that operations cannot leak data either by:
  - Reading up
  - Writing down



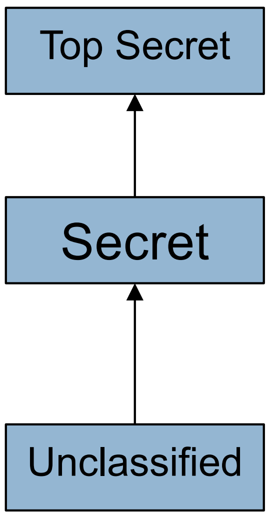
Multi-Level Security (MLS)

- An operation is only authorized if:
  - Read: $SC_{Subject} >= SC_{Object}$
  - Write: $SC_{Subject} <= SC_{Object}$
- Suppose a Trojan horse is run to access Top Secret data
  - Can it leak that data?
    - E.g., Write to an unclassified file



Multi-Level Security (MLS)

- An operation is only authorized if:
  - Read: $SC_{Subject} >= SC_{Object}$
  - Write: $SC_{Subject} <= SC_{Object}$
- Suppose a Trojan horse is run to access Top Secret data
  - Can it leak that data?
    - E.g., Write to an unclassified file



- Blocked by MLS enforcement

Issues with MLS

- Did we ensure confinement of the process?
  - Yes!
  - Access control is configured based on security
  - So, no way to leak secrets assuming subject and object labels are correct
    - And no side channels (out of scope for the course)
- Did we allow a program its least privilege permissions?
  - No!
  - Cannot even have bi-directional communication
- As a result, MLS is used in limited cases (isolation)

Access Control Enforcement

- What do we need to do to enforce access control correctly and comprehensively?
  - Comprehensive: all security-sensitive operations
  - Correctly: are checked against the expected policy

Reference Monitor Concept

- Describes the requirements for correct and comprehensive enforcement
- Complete mediation: The reference validation mechanism must always be invoked on each security-sensitive operation.
- Tamperproof: The reference validation mechanism must be tamperproof.
- Verifiable: The reference validation mechanism must be small enough to be subject to analysis and tests, the completeness of which can be assured.

Linux Security Modules

- Linux mechanism to enforce mandatory access control (reference monitor)
  - https://www.kernel.org/doc/html/v4.16/admin-guide/LSM/index.html
- Main goal: confine network-facing daemons
  - To make it difficult to compromise a host with one vulnerability
  - Main "modules" include: SELinux, AppArmor, Tomoyo
- Least privilege for root processes
  - MLS can be used to isolate processing (VMs)

Conclusions

- Access control and authentication are the two fundamental security mechanisms
  - We have seen access control throughout this course
- UNIX access control uses Access Control Lists (mode bits) per file to list authorized subjects
  - Prevents accidents but cannot enforce security
- Linux now enforces mandatory access control
  - Least privilege: Limits malware/compromised daemons
  - Multilevel security: Prevents illegal info flows

Questions