Access Control
Security Begins on the Host

• Even without a network, hosts must enforce the CIA trilogy

• Something on the host—the operating system aided by the hardware—must provide those guarantees
Access Control

- Hardware
- Software
  - Operating systems
  - Databases
  - Other multi-access programs
- Distributed (including web servers)
Hardware

• What is the *minimum* necessary?

• What do other mechanisms buy us?
Minimum Functionality

- Protect the OS from applications
- Protect applications from each other
- Change state from application to OS
- Timer interrupt
Why a Timer?

- Availability is a security feature
- Must prevent uncooperative applications from hogging CPU
- Not going to discuss this more here, but it's a major topic in W4118 (Operating Systems)
Historical Mechanisms

- Single privileged mode bit—restrict ability to execute certain instructions
- Memory protection
- Interrupts—hardware and software—cause state transition
- The two states are (today) commonly called “kernel” and “user” mode
What Are Privileged Instructions?

- Ability to do I/O without the OS’s intervention—allowing that could bypass file permission checking
- Ability to manipulate timers
- Ability to access other programs’ memory without OS intervention
Example: IBM System/360 Mainframe

- Designed in the early 1960s
- Much of the architecture still in use...
- 4-bit protection key associated with each 4K block of memory, plus read-protect bit
- Single “supervisor mode” bit
- 4-bit state key of 0 can write to anything
- But—operating systems of that time didn’t use the hardware to its full capabilities
Memory-Mapped Control

- On some machines, privileged operations work by memory access
- If applications have no access to such memory, they can’t do sensitive things
- But—must have way to enter privileged state
Multics

- Virtual memory
- “Ring” structure—8 different privilege levels (i386 has rings, too)
- OS could use rings 0-3; applications could use 4-7.
- (Original design had 64 rings!)
- Each ring is protected against higher-numbered rings
- Special form of subroutine call to cross rings
- Most of the OS didn’t run in Ring 0
What is the Advantage of Rings?

- A single bit is theoretically sufficient
- Assurance!
- Don’t need to trust all parts of the system equally
- “Principle of Least Privilege”
Assurance

- How do you know something is secure
- Much harder to provide later than features
- A trustable secure system has to be designed that way from the beginning: designed, documented, coded, and tested—and maybe proved
Underlying Principles of Privilege

- Two basic approaches to privilege: identity and attribute
- Hardware protection is *attribute*: the state of various registers controls what can and cannot be done
- Easier to manage in a single system
What is the role of the OS?

- Protect itself
- Separate different applications
- More?
Operating Systems and Hardware

- The hardware provides the minimum functionality
- The OS has to provide its own services on top of that
- This is the application’s *virtual execution environment*
- Must manage access to I/O devices as well
What Protections do Operating Systems Provide?

- User authentication (why?)
- File protection
- Process protection
- Resource scheduling (CPU, RAM, disk space, etc)
User Authentication

- (Much more on this later)
- Why authenticate users?
- Most operating system privileges are granted by identity, not attributes
- Procedure:
  - Authenticate user
  - Grant access based on userid
File Permissions

• Besides user authentication, the most visible aspect of OS security
• Read protection—provide confidentiality
• Write protection—provide integrity protection
• Other permissions as well
Classical Unix File Permissions

- All files have "owners"
- All files belong to a "group"
- Users, when logged in, have one userid and several groupids.
- 3 sets of 3 bits: read, write, execute, for user, group, other
- (512 possible settings. Do they all make sense?)
- Written rwxrwxrwx
- 111 101 001: User has read/write/exec; group has read/exec; other has exec-only
Permission-Checking Algorithm

if curr_user.uid == file.uid
    check_owner_permissions();
else if curr_user.gid == file.gid
    check_group_permissions();
else
    check_other_permissions();
fi

Note the else clauses—if you own a file, “group” and “other” permissions aren’t checked
Execute Permission

- Why is it separate from “read”?
- To permit only execution
- Cannot copy the file
- Readable only by the OS, for specific purposes
Directory Permissions

- “write”: create a file in the directory
- “read”: list the directory
- “execute”: trace a path through a directory
Example: Owner Permissions

$ id
uid=54047(smb) gid=54047(smb) groups=0(wheel),3(sys),54047(smb)
$ ls -l not_me
----r--r-- 1 smb wheel  29 Sep 12 01:35 not_me
$ cat not_me
cat: not_me: Permission denied

I own the file but don’t have read permission on it
Example: Directory Permissions

$ ls -ld oddball
dr--r--r-- 2 smb wheel 512 Sep 12 01:36 oddball
$ ls oddball
cannot_get_at
$ ls -l oddball
ls: cannot_get_at: Permission denied
$ cat oddball/cannot_get_at
cat: oddball/cannot_get_at: Permission denied

I can read the directory, but not trace a path through it to oddball/cannot_get_at
Deleting Files

- What permissions are needed to delete files?
- On Unix, you need write permission on the parent directory.
- You can delete files that you can’t write. You can also write to files that you can neither create nor delete.
- Other systems make this choice differently.
Historical Note

- Unix has *never* been fond of asking "do you really mean that?"

- That said, at least as long ago as February 1973 the original Bell Labs Unix `rm` command prompted if you tried to delete a file you couldn’t write

- In other words, the Unix model is philosophically correct but perhaps incorrect from a human factors perspective
Access Control Lists

- 9-bit model not always flexible enough
- Many systems (Multics, Windows XP and later, Solaris, Linux, MacOS) have more general Access Control Lists
- ACLs are explicit lists of permissions for different parties
- Wildcards are often used
Sample ACL

smb.*          rwx
4187-ta.*     rwx
*.faculty      rx
*.*            x

Users “smb” and ‘4187-ta” have read/write/execute permission. Anyone in group “faculty” can read or execute the file. Others can only execute it.
Order is Significant

With this ACL:

*.faculty    rx
smb.*       rwx
4187-ta.*   rwx
.*          x

I would not have write access to the file
MacOS ACLs
Windows Vista ACLs
Linux/Solaris ACLs

$ getfacl acl.pdf
# file: acl.pdf
# owner: smb
# group: smb
user::rw-
user:postfix:-w-
group::r--
group:landscape:--x
mask::rwx
other::r--

The standard Unix permissions are translated into ACL entries
Setting File Permissions

- Where do initial file permissions come from?
- Who can change file permissions?
Unix Initial File Permissions

• Unix uses “umask”—a set of bits to *turn off* when a program creates a file

• Example: if umask is 022 and a program tries to create a file with permissions 0666 (rw for user, group, and other), the actual permissions will be 0644.

• Default system umask setting has a great effect on system file security

• Set your own value in startup script; value inherited by child processes
Why Umask?

- Suppose files were always created with rw,r,r permissions
- What’s wrong with the application simply changing the file permissions after creating the file?
- Race conditions
What is a “Race Condition”?

A race condition is when two or more asynchronous processes try to access the same resource “simultaneously” but it is not possible to control or predict which will happen first.

<table>
<thead>
<tr>
<th>Sequence 1</th>
<th>Sequence 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create file mode 666</td>
<td>Create file mode 666</td>
</tr>
<tr>
<td>Change permissions to 600</td>
<td>Attacker tries to read the file</td>
</tr>
<tr>
<td>Attacker tries to read the file</td>
<td>Change permissions to 600</td>
</tr>
</tbody>
</table>

It is impossible to predict which will happen!
Multics Initial File Permissions

- Directories contain “initial access control list”—values set by default for new files

- Common setting:
  
  smb.faculty rw
  
  *.sysdaemon r
  
  *.* -

- If group “sysdaemon” doesn’t have read permission, the file can’t be backed up!

- Linux and Solaris also have default ACLs for new files
MAC versus DAC

- Who has the right to set file permissions?
- Discretionary Access Control (DAC)—the file owner can set permissions
- Mandatory Access Control (MAC)—only the security officer can set permissions
- Enforce site security rules
- Note: viruses and other malware change DAC permissions, but not MAC permissions
Implementing MAC

- Often side-by-side with DAC: system has both
- Processes need to pass both sets of permissions to access files
- Or—can have a special ACL-changing attribute in an ACL:
  ```
  security_officer.wheel p
  ```
- But—can security_officer give him/herself privileges?
- In reality, MAC is often used for classification levels (next class), rather than ACLs
Privileged Users

- Root or Administrator can override file permissions
- This is a serious security risk—there is no protection if a privileged account has been compromised
- There is also no protection against a rogue superuser...
- Secure operating systems do not have the concept of superusers
Database Access Control

• Often have their own security mechanisms
• Permit user logins, just like operating systems
• Some have groups as well
• Permissions are according to database concepts: protect rows and columns
• Different types of operations: select, insert, update, delete, and more
Databases versus OS Security

- The database has many objects in a single OS file
- The OS can control access to the file
- The DBMS has to control access to objects within the file
- The set of database users is not the same as the set of OS users
Web Server ACLs

• Can restrict access based on web login (*not* OS login!)
• Can also use network concepts, e.g., IP address or cryptographic criteria
• More on this later in the semester