Access Control

Security Begins on the Host

- Computer systems must enforce the confidentiality/integrity/availability 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, e.g., servers for mobile apps
- 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: the IBM Z computers
- 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 capabilites

Memory-Mapped Control

- On some machines, e.g., the PDP-11s on which early Unix development was done, 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—why do we need rings?

What is the Advantage of Rings?

- A single bit is theoretically sufficient—why do we need rings?
- Assurance!
- Don't need to trust all parts of the system equally

What is the Advantage of Rings?

- A single bit is theoretically sufficient—why do we need rings?
- 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

Rings Today

- Modern x86 hardware supports 4 rings
- There's also Ring -1, for the hypervisor
- Why isn't the hypervisor Ring 0? Maximum compatibility with guest operating systems, with as few traps as possible to the hypervisor
- But rings aren't used. . .

Why Not?

- Ring-crossing is expensive—dividing a kernel or an application into multiple rings would hurt performance
- Rings don't play as well as one would like with the virtual memory system
- The kernel—on Windows, Linux, and MacOS—runs in Ring 0; applications run in Ring 3
- It would be nice if applications started in Ring 2, to allow them to protect themselves more

What is the Role of the OS?

- Protect itself
- Separate different applications
- More?

App Stores

- Today's commercial operating systems (including for phones) are linked to an "app store"
- The OS can ensure that all applications are digitally signed with a certificate from the proper app store
- Can protect against malware—and for iOS, can protect Apple's revenue stream
- Is the OS protecting the user, the applications—or protecting the system *from* the user?

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

- 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, the 1971 Bell Labs Unix Programmer’s Manual said

```
BUGS    rm probably should ask whether a read-only file
        is really to be removed
```

  and by 1973 that had been implemented
- 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, 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
4181-ta.*     rwx
*.faculty     rx
*.*           x
```

Users “smb” and ‘4181-ta” have read/write/execute permission. Anyone in group “faculty” can read or execute the file. Others can only execute it.

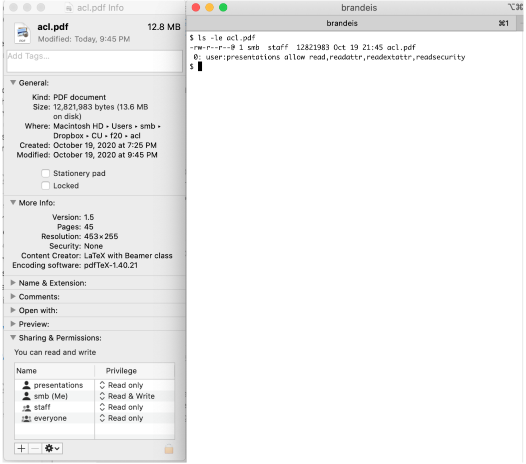
Order is Significant

With this ACL:

```
*.faculty    rx
smb.*        rwx
4181-ta.*    rwx
*.*          x
```

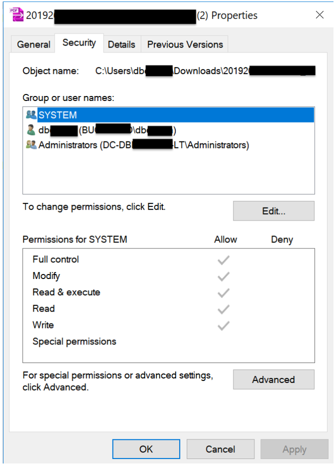
I would not have write access to the file

MacOS ACLs



acl.pdf Info
acl.pdf
12.8 MB
Modified: Today, 9:45 PM
Add Tags...
General:
Kind: PDF document
Size: 12,821,983 bytes (13.6 MB on disk)
Where: Macintosh HD ▸ Users ▸ smb ▸ Dropbox ▸ CU ▸ f20 ▸ acl
Created: October 19, 2020 at 7:25 PM
Modified: October 19, 2020 at 9:45 PM
Stationery pad
Locked
More Info:
Version: 1.5
Pages: 45
Resolution: 453×255
Security: None
Content Creator: LaTeX with Beamer class
Encoding software: pdfTeX-1.40.21
Name & Extension:
Comments:
Open with:
Preview:
Sharing & Permissions:
You can read and write
Name
Privilege
presentations
Read only
smb (Me)
Read & Write
staff
Read only
everyone
Read only
brandeis
brandeis
⌘1
$ ls -le acl.pdf
-rw-r--r--@ 1 smb staff 12821983 Oct 19 21:45 acl.pdf
0: user:presentations allow read,readattr,readextattr,readsecurity
$

Windows 10 ACLs



Linux 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 permssions 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.

Sequence 1	*Sequence 2*
Create file mode 666	Create file mode 666
Change permissions to 600	Attacker tries to read the file
Attacker tries to read the file	Change permissions to 600

It is impossible to predict which will happen!

Race Conditions

- Made easier by multicore CPUs—there really is true simultaneity now
- Extremely hard to find by testing, unless you tune your tests specifically for each such situation
- Many different variants

TOCTTOU: Time of Check to Time of Use

TOCTTOU races: a program tries checking file permissions itself instead of relying on the OS

```
fn = getfilename();
if (check_user_perms(fn))
    /* Attacker changes the
       file here! */
    process_file(fn);
```

```
$ touch file attacker-file
$ system_cmd attacker-file &
$ rm attacker-file
$ ln (system-file) attacker-file
```

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 also have default ACLs for new files

Privilege-Setting Privilege

- Who has the right to set file permissions?
- Generally, the file owner can set its permissions
- Note: viruses and other malware can change permissions on behalf of some user
- A user cannot use file permissions to protect their own account from malware executing with their privileges

Privileged Users

- (We'll discuss privilege next week)
- `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/Mobile App Server Security

- Similar issues arise for other multi-user applications
- Most obvious example: gmail and other big mail systems
- *Application* users are not *OS* users—which means that the operating system's file protections can't be used
- Query: are there race condition attacks?

Why Use File Permissions?

- File permissions were designed for multi-user computers—do we still need them?

Why Use File Permissions?

- File permissions were designed for multi-user computers—do we still need them?
- Yes—prevent privilege escalation
- Protect the OS from applications (and users, and malware)
- Protect parts of applications from each other, e.g., in web servers
- Protect resources such as keys

System Design Principles

- Essentially nothing should be world-writable
- Making things world-readable, unless there's a strong reason not to, generally simplifies design and coding
- You can't easily protect intellectual property by making things read-protected—the attacking users generally have full permissions on the system

Questions?

(American robin eating a berry, Morningside Park, October 19, 2020)