CSE 120
Principles of Operating Systems

Fall 2024

Lecture 14: Protection

Geoffrey M. Voelker

On Protection

- OS textbooks can be somewhat cryptic when it comes to some aspects of protection
  - Access control lists in the file system make sense
  - But capabilities often remain mysterious, why we have them, how OSes actually use them, and how they relate to ACLs
- Goal is to make this more concrete, and to explain why
  - You will never look at “opening a file” the same way again…

Multics

- Historically very important operating system
  - Large research project at MIT started in the 60s
  - Not a commercial operating system, but…
- Unix drew heavily upon ideas from Multics
  - Unix tended to avoid the more complex aspects of Multics
    - Multiple reasons (lack of hardware support, design philosophy)
- Famous seminal paper on Multics protection
  - Jerome H. Saltzer, "Protection and the Control of Information Sharing in Multics", CACM 1974
- Describes the design and mechanisms for protection, and reasoning behind the design choices (the "why")
  - Modern OSes (Unix, Windows, MacOS) follow these footsteps

Protection Principles

1) Permission rather than exclusion
   - Default is no access (will immediately discover if wrong)
2) Check every access to every object
   - Including every instruction and memory reference
3) Design is not secret
   - Linux is open source, and that should not make it insecure
4) Principle of least privilege
   - Only execute with the privileges you need (avoids mistakes)
5) User interface to protection must be easy to use
   - If it is hard for users to use the protection system, they will not use it and instead find ways around it

We will see how these principles manifest in OSes today

Users

- Protection starts with the concept of user
- Which user you are completely defines…
  - What programs you can run (execute)
  - Which files you can access, and how (read, write)
- Cannot do anything on the system until you login
- Once you login, everything you do on the system is performed under your user ID (UID)
  - Every process runs under a user ID
  - The user ID is the basis for protection checks
- Can a process open a file? → Does the user ID associated with the process have permission to open the file?

Root & Administrator

- The user "root" is special on Unix
  - It bypasses all protection checks in the kernel
  - Administrator is the equivalent on Windows
- Recall "Principle of least privilege"
- Always running as root can be dangerous
  - A mistake (or exploit) can harm the system
    - "rm" will always remove a file
  - Why we create user accounts even if you have root access
    - You only run as root when you need to modify the system
  - If you have Administrator privileges on Windows, then you are effectively always running as root (unfortunately)
    - Need additional protection mechanisms (User Account Control)

setuid

- OSes provide a mechanism to enable you to run programs with the privileges of other users
  - Unix: setuid, setgid (on executable files)
  - Windows: runas, CreateProcessAsUser (on process creation)
- Normally a process runs with your user privileges

```
10:52 (6) /bin> ls -l ls
-rwxr-xr-x 1 root root 110080 Mar 10  2016 ls*
```

- By running a setuid program, the process runs with the privileges of the user or group associated with the file

```
10:53 (7) /bin> ls -l mount
-rwsr-xr-x 1 root root 94792 Sep  2  2015 mount*
```

su & sudo

- The su command runs a shell with root privileges
  - Authenticate using the password for the root user
  - Effectively logging in as root
  - All child processes (commands) run with root privs
- The sudo command runs a process with root privileges
  - Authenticate using the user’s password
  - User must be in the sudo group (/etc/group)
  - Effectively running the process as setuid root
  - More precise than su since it is per-process

Android

- Android uses Linux as the underlying operating system
  - Linux has a protection model designed for many users
  - But smartphones are single-user personal devices
- Instead, Android uses the user abstraction for apps
  - Each app has its own user ID (UID)
  - All the mechanisms for isolation, protection, and sharing implemented for users now applies to apps
  - Provides a user-based sandbox for each app

File System Protection

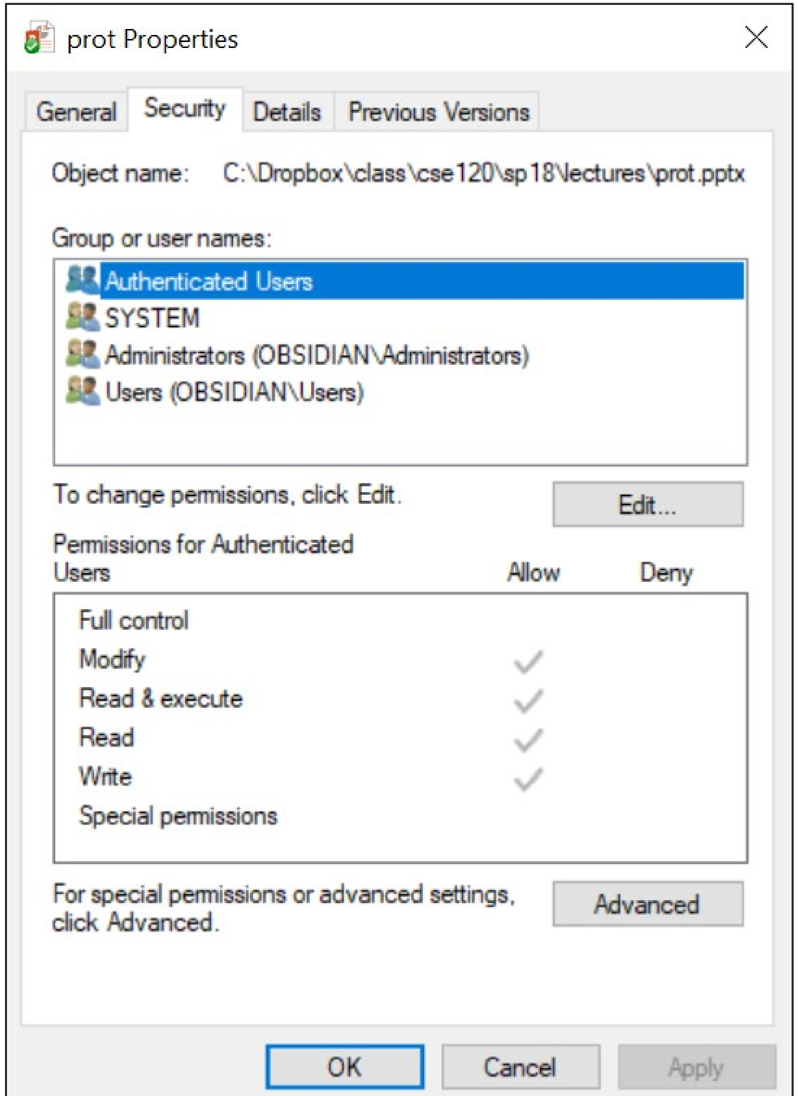
- The file system stores the permissions on all objects (files, directories, executables, devices, …)
  - It is the static representation of permissions
- The mechanism used to represent static permissions is the access control list (ACL)
  - Recall "Permission, not exclusion"
- For each object (file), which users have access to the object, and what rights do they have?
  - Can be compact: Unix's owner/group/other, read/write/execute
  - Can be flexible: an arbitrary list of user+rights entries
    - Windows' explicit ACLs
    - Linux extended file attributes (xattrs)

Unix Access Control List

- Completely familiar to you

```
10:52 (6) /bin> ls -l ls
-rwxr-xr-x 1 root root 110080 Mar 10  2016 ls*
```

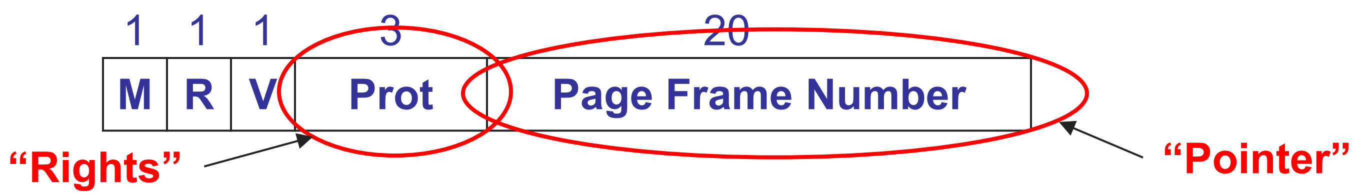
Windows Access Control List



Virtual Memory Protection

- The address space defines permissions for a process under execution
  - It is the dynamic representation of permissions
- The mechanism used to represent dynamic permissions for using an address space are capabilities
- Capabilities are pointers (references) + rights
  - Also known as descriptors, tokens, etc.
  - Pointer/reference identifies an object
  - Rights determine what you can do with an object
- Page table entries are our VM capabilities
  - Every PTE determines what the process can do with that page

Page Table Entries (PTEs)



- Page table entries control mapping
  - The Modify bit says whether or not the page has been written
    - It is set when a write to the page occurs
  - The Reference bit says whether the page has been accessed
    - It is set when a read or write to the page occurs
  - The Valid bit says whether or not the PTE can be used
    - It is checked each time the virtual address is used
  - The Protection bits say what operations are allowed on page
    - Read, write, execute
  - The page frame number (PFN) determines physical page

PTEs as Capabilities

- Recall "Check every access"
- When it comes to memory, this means:
  - Check every instruction execution
  - Check every load/store
- The TLB uses PTEs to check every memory access
  - When the CPU loads the next instruction to execute, the TLB verifies that the instruction comes from a page that has the execute bit set
  - When the CPU stores a value onto a page, the TLB verifies that the process has write-access to that page (not read-only)

Protection Model

- More formally…
  - Objects are “what”, subjects are “who”, actions are “how”
  - Logging in determines the subject (“who”)
  - Objects in the file system are the “what” (also processes)
  - Permissions are the actions (“how”)
- A protection system dictates whether a given action performed by a given subject on a given object should be allowed
  - You can read and/or write your files, but others cannot
  - You can read “/etc/motd”, but you cannot write it

Representing Protection

Access Control Lists (ACL)

- For each object, maintain a list of subjects and their permitted actions

Capability Lists

- For each subject, maintain a list of objects and their permitted actions

Objects

Subjects

	/one	/two	/three
Alice	rw	-	rw
Bob	w	-	r
Charlie	w	r	rw

Capability List

ACL

(Table not actually materialized)

ACLs and Capabilities

- Approaches differ only in how the table is "represented"
  - Have different tradeoffs, so we use them in different ways
- Capabilities are easier to transfer
  - They are like keys, can handoff, does not depend on subject
  - Very fast to check
    - TLB uses PTEs to check every memory reference
- In practice, ACLs are easier to use
  - Object-centric, easy to grant, revoke
    - To revoke capabilities, have to keep track of all subjects that have the capability – a challenging problem
  - Easier for users to express their protection goals
  - But, ACLs slow to check compared to capabilities

Why Have Both?

- OSes use ACLs on objects in the file system
  - These are what users manipulate to express protection
- OSes use capabilities when checking access frequently
  - Checking every memory reference needs to be fast
  - Checking protection bits in PTEs can be done by hardware
- So the OS uses both, and they are directly related
  - Capabilities are in fact derived from ACLs
  - Let users express protection with ACLs
  - ACLs are slow to check, so bootstrap from ACLs into capabilities
  - Capabilities are much faster to check, can check frequently
- Two examples to make this more concrete

Checking File Permissions

- Recall the principle of "check every access"
- For reading/writing a file, that means that the OS needs to verify on every read()/write() that the process has permission to perform the read/write syscall
- But, checking file permissions is expensive
  - Scanning ACLs on every read/write is slow
- So how do we optimize the permissions check?
  - Open!

Opening a File

- Ever since we started learning how to program, we learned that to read/write a file we first had to open it
  - Open seems completely natural to us
- "Opening a file" is actually a subtle, but crucial step in bootstrapping protection from the file system (static) to executing in a process (dynamic)
  - It bootstraps from an ACL to a capability

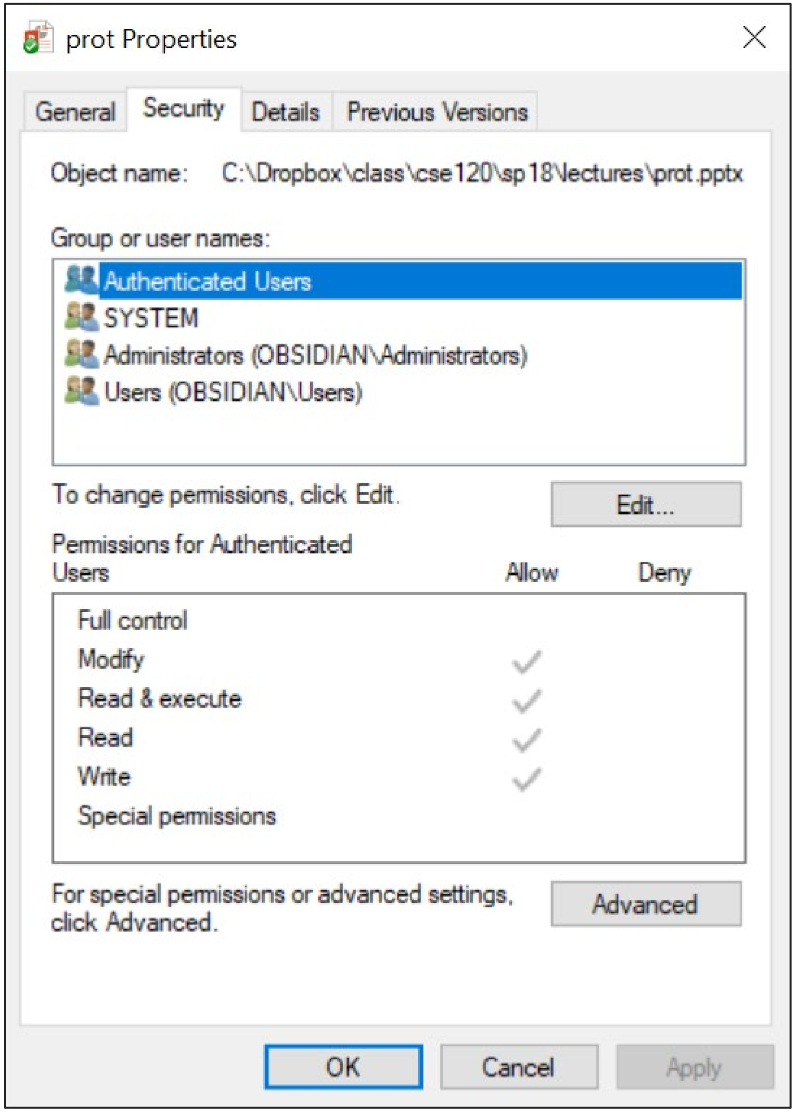
File Descriptors

- When a process calls open, the OS checks the user ID for the process against the ACL for the file
  - The process wants to open the file for writing, does the ACL say that the process user ID has write permission for the file?
  - Checking an ACL is slow, so we only want to do it once
- What does open return? A file descriptor
  - This descriptor is a capability
  - It is passed to every call to read/write
- OS checks the descriptor on every read/write to verify:
  - That the descriptor is valid (the file was opened)
  - That the process can perform the action on the file
    - Calling write on a file opened read-only will fail
    - OS doesn't check the ACL, it checks the descriptor (capability)

PTEs Once Again

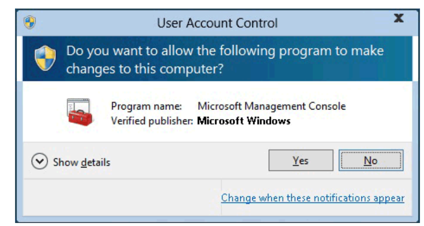
- We said PTEs are capabilities
  - So where are they derived from?
- Recall loading a program into an address space
- When creating the address space
  - For the pages containing code, we set the PTE protection bits to read-only and execute (if the hardware supports it)
  - For pages containing data, we set the PTE protection bits to read/write, but not execute
  - For memory-mapped files, we set the PTE protection bits to read/write or read-only depending on the file ACL
    - If the ACL says that the user ID for the process only has read access to a file, can only map it read-only in the address space

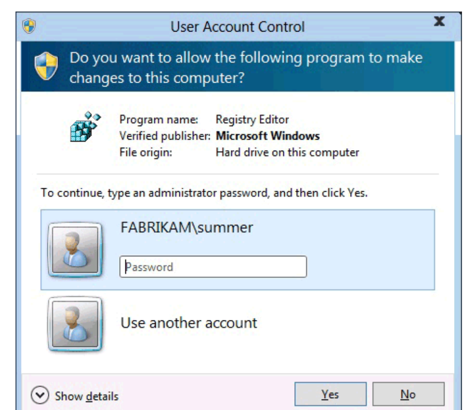
“Ease of Use” Principle



User Access Control

- Windows personal user accounts typically in the Administrators group
  - Effectively always running as "root"
  - Malware that exploits a user process now has root privileges
- Windows now has a second level of authorization / authentication: User Access Control
  - Prompt for authorizing certain tasks
  - Prompt to authenticate as Administrator for other tasks (similar to "su")
  - Require user interaction as a guard against malware





Next time

- Read Appendix B