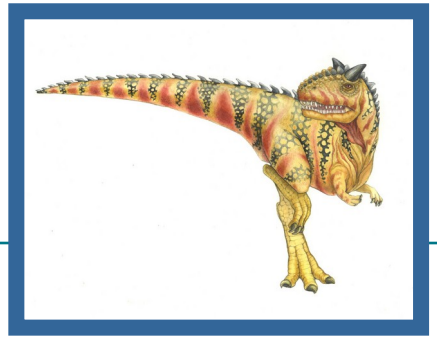
Operating System Concepts

(slides adapted from 10th ed. Silberschatz, Galvin and Gagne)

Chapter 17

Protection



Objectives

- Discuss the goals and principles of protection in a modern computer system
- Explain how protection domains combined with an **access matrix** are used to specify the resources a process may access
- Examine **capability** and language-based protection
- Describe how protection mechanisms can mitigate system attacks

Goals of Protection

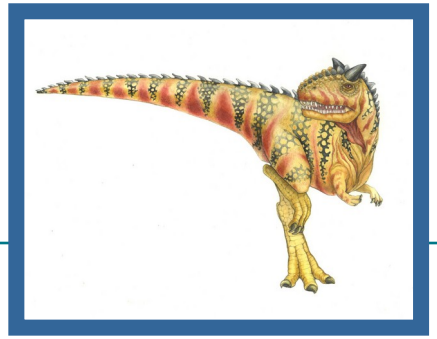
- In one protection model, computer consists of a collection of **objects (resources)**, hardware or software
- Each object has a unique name and can be accessed through a well-defined set of **operations**
- *Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so*

Mechanism vs. policy: “how” vs. “what”

Principles of Protection 1

- Guiding principle – **principle of least privilege**
  - Programs, users and systems should be given just enough **privileges** to perform their tasks
  - Properly set **permissions** can limit damage if entity has a bug, gets abused
  - Can be static (during life of system, during life of process)
  - Or dynamic (changed by process as needed) – **domain switching**, **privilege escalation**
  - **Compartmentalization** a derivative concept regarding access to data
    - Process of protecting each individual system component through the use of specific permissions and access restrictions

17.3-17.4 Protection Domains

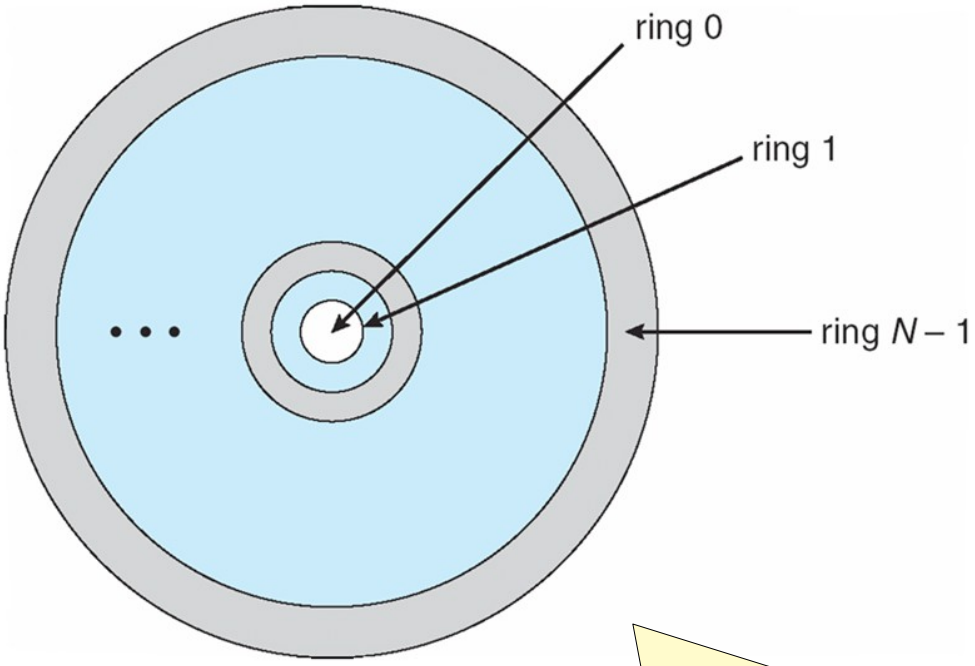


Protection Rings 3

- Components ordered by amount of privilege and protected from each other
  - For example, the kernel is in one ring and user applications in another
  - This privilege separation requires **hardware support**
  - Instructions to transfer between levels, e.g., RISC-V ecall, sret
  - Also traps and interrupts
  - **(Hypervisors** introduced the need for yet another ring)

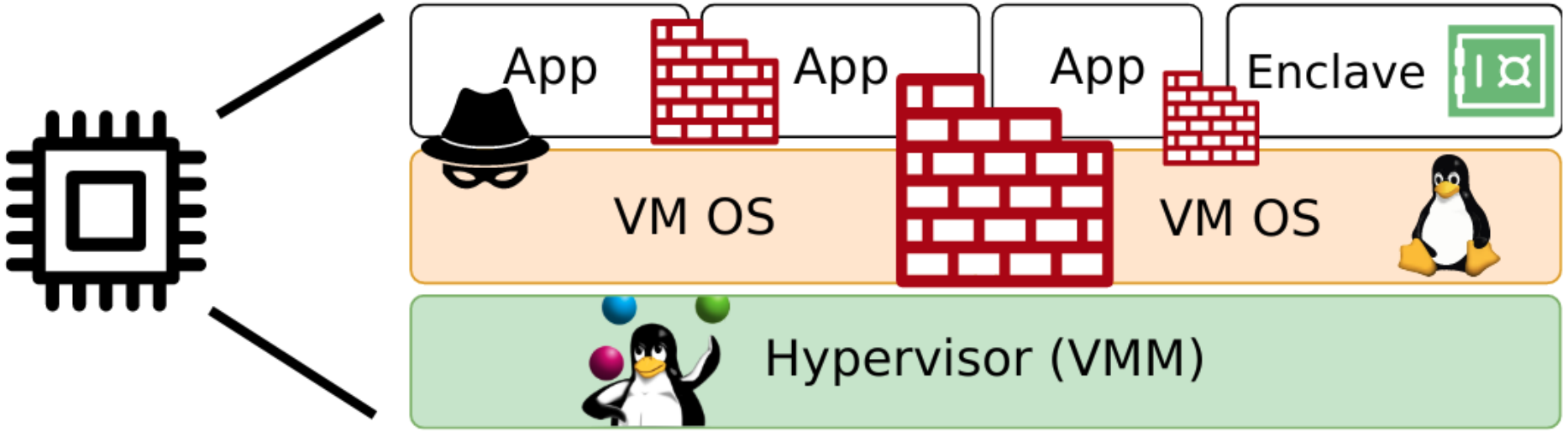
Essential building block for operating system platform protection!

Protection Rings (MULTICS)



e.g., **RISC-V** privileges *machine* > *kernel* > *user* mode

The Big Picture: Software Isolation



OS (and hypervisor, similar but not covered here) uses **CPU privilege rings** and **virtual memory** to build “walls” for memory isolation between applications

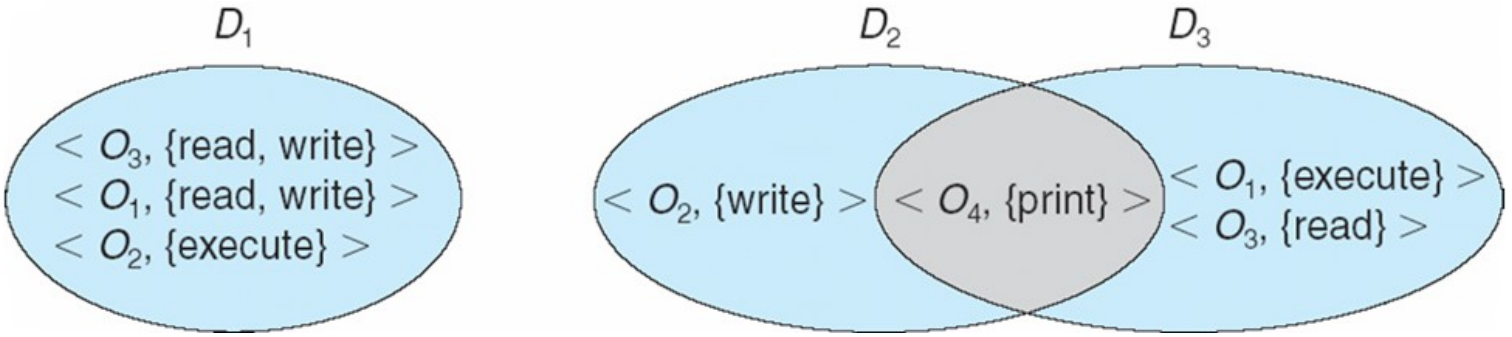
Generalization: Domains of Protection [1]

- OS controls access between domains and objects
  - **Hardware objects** (such as memory, devices) and **software objects** (such as files, programs, semaphores)
- Domain can be e.g., user, process, procedure
- Process for example should only have access to objects it currently requires to complete its task – the **need-to-know** principle
- Controlled **domain switches** to change access rights

== principle of least privilege

Domain Structure

- Access-right = <*object-name*, *rights-set*>
  where *rights-set* is a subset of all valid operations that can be performed on the object
- Domain = set of access-rights



Domain Implementation (UNIX)

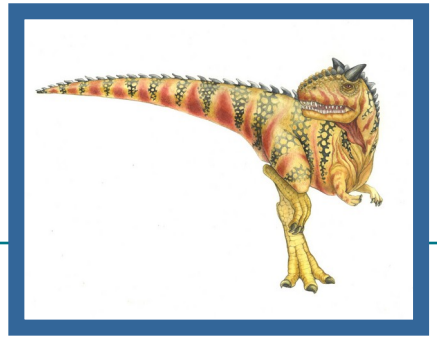
- **Domain = user-id**
- Domain switch accomplished via file system
  - Each file has associated with it a domain bit (**setuid** bit)
  - When file is executed and setuid = on, then user-id is set to owner of the file being executed
  - When execution completes user-id is reset
- Domain switch accomplished via passwords
  - `su` command temporarily switches to another user's domain when other domain's password provided
- Domain switching via commands
  - `sudo` command prefix executes specified command in another domain (if original domain has privilege or password given)

Least privilege principle: don't run everything as root(!)

Domain Implementation (Android App IDs)

- In Android, distinct user IDs are provided on a **per-application** basis
- When an application is installed, the installd daemon assigns it a **distinct user ID** (UID) and group ID (GID), along with a **private data directory** (/data/data/<appname>) whose ownership is granted to this UID/GID combination alone.
- Applications on the device enjoy the same level of protection provided by UNIX systems to separate users
- A quick and simple way to provide isolation, security, and privacy.

17.5-17.7 Access Control Matrix



Access Matrix

Policy vs. mechanism: matrix can be implemented in different ways

- View protection as a matrix (**access matrix**)
- Rows represent domains
- Columns represent objects
- **`Access(i, j)`** is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Use of Access Matrix 1

- If a process in Domain D_i tries to do "op" on object O_j, then "op" must be in the access matrix
- User who creates object can define access column for that object
- Can be expanded to dynamic protection
  - Operations to add, delete access rights
  - Special access rights:
    - *owner of* O_i
    - *copy op from* O_i *to* O_j *(denoted by "*")*
    - *control –* D_i *can modify* D_j *access rights*
    - ***transfer – switch from domain*** D_i ***to*** D_j
  - *Copy* and *Owner* applicable to an object
  - *Control* applicable to domain object

Access Matrix of Figure A with Domains as Objects

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Use of Access Matrix 1

- If a process in Domain D_i tries to do "op" on object O_j, then "op" must be in the access matrix
- User who creates object can define access column for that object
- Can be expanded to dynamic protection
  - Operations to add, delete access rights
  - Special access rights:
    - *owner of* O_i
    - ***copy op from*** O_i ***to*** O_j ***(denoted by "*")***
    - *control –* D_i *can modify* D_j *access rights*
    - *transfer – switch from domain* D_i *to* D_j
  - *Copy* and *Owner* applicable to an object
  - *Control* applicable to domain object

Access Matrix with *Copy* Rights

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Use of Access Matrix 1

- If a process in Domain D_i tries to do "op" on object O_j, then "op" must be in the access matrix
- User who creates object can define access column for that object
- Can be expanded to dynamic protection
  - Operations to add, delete access rights
  - Special access rights:
    - ***owner of O_i***
    - *copy op from O_i to O_j (denoted by "*")*
    - *control – D_i can modify D_j access rights*
    - *transfer – switch from domain D_i to D_j*
  - *Copy* and *Owner* applicable to an object
  - *Control* applicable to domain object

Access Matrix With *Owner* Rights

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Implementation of Access Matrix 1

- Generally, a *sparse matrix*
- Option 1 – Global table
  - Store ordered triples **<domain, object, rights-set>** in table
  - A requested operation M on object O_j within domain D_i -> search table for $< D_i, O_j, R_k >$
    - with M $\in$ R_k
  - But table could be large -> won't fit in main memory
  - Difficult to group objects (consider an object that all domains can read)

Conceptual, but not used in practice!

e.g., (Alice: read,write; Bob: read)

Implementation of Access Matrix 2

- Option 2 – Access control lists (ACL) for objects
  - Each column implemented as an access list for one object
  - Resulting per-object list consists of ordered pairs **<domain, rights-set>** defining all domains with non-empty set of access rights for the object
  - Easily extended to contain default set -> If M ∈ default set, also allow access

Implementation of Access Matrix 3

- Each **column = Access-control list** for one object
  Defines who can perform what operation

  Domain 1 = Read, Write
  Domain 2 = Read
  Domain 3 = Read

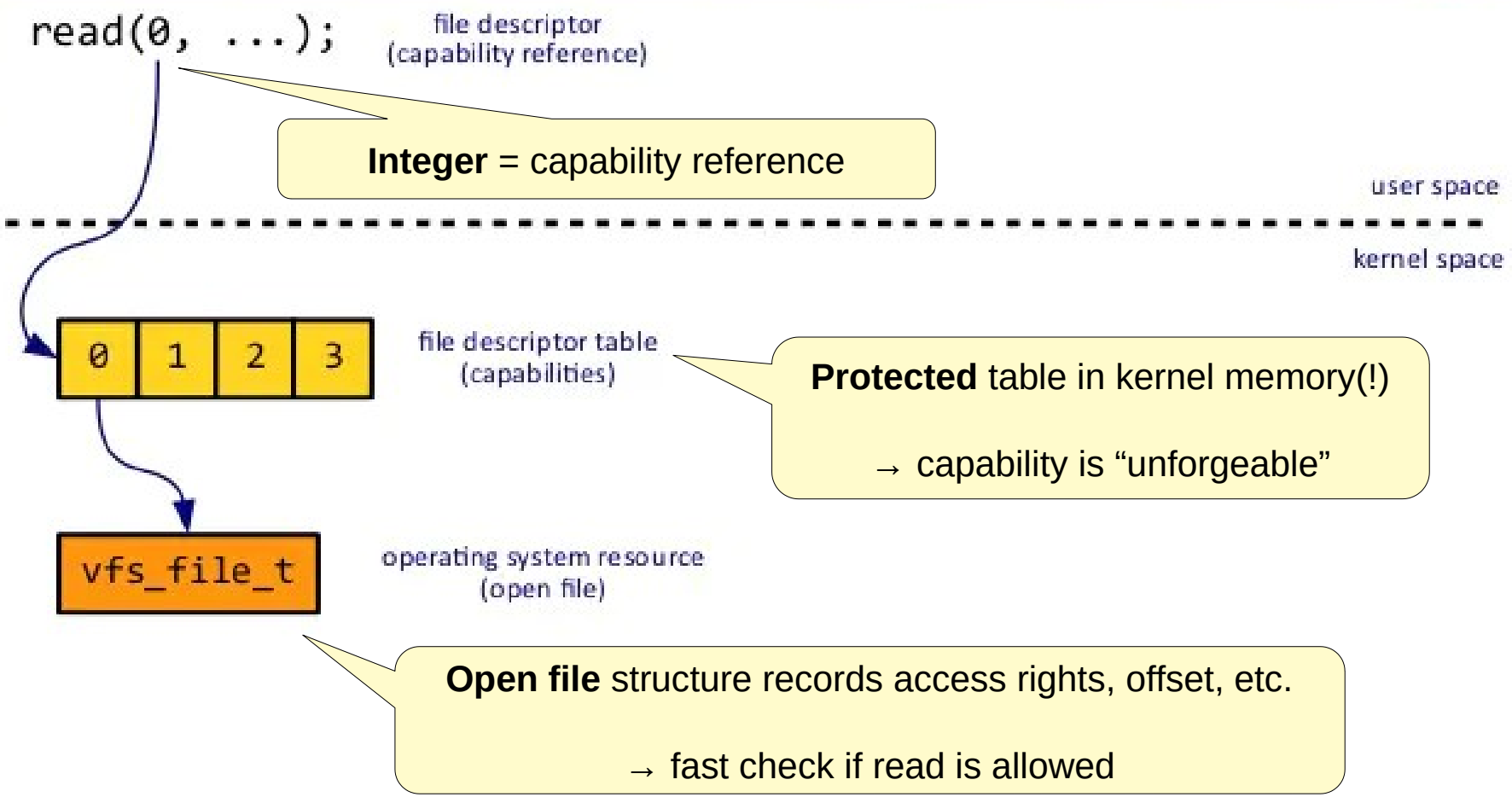
- Each **Row = Capability List** (like a key)
  For each domain, what operations allowed on what objects

  Object F1 – Read
  Object F4 – Read, Write, Execute
  Object F5 – Read, Write, Delete, Copy

Implementation of Access Matrix 4

- Option 3 – Capability list for domains
  - Instead of object-based, list is domain based
  - **Capability list** for domain is list of objects together with allowed operations
  - Object represented by its name or address, called a **capability**
  - Execute operation M on object O_j, process requests operation and specifies capability as parameter
    - **Possession of capability means access is allowed**
  - Capability list associated with domain but **never directly accessible by domain**
    - Rather, protected object, maintained by OS and accessed indirectly
    - Like a "secure pointer"
    - Idea can be extended up to applications

Capability example: File descriptors



Comparison of Implementations [1]

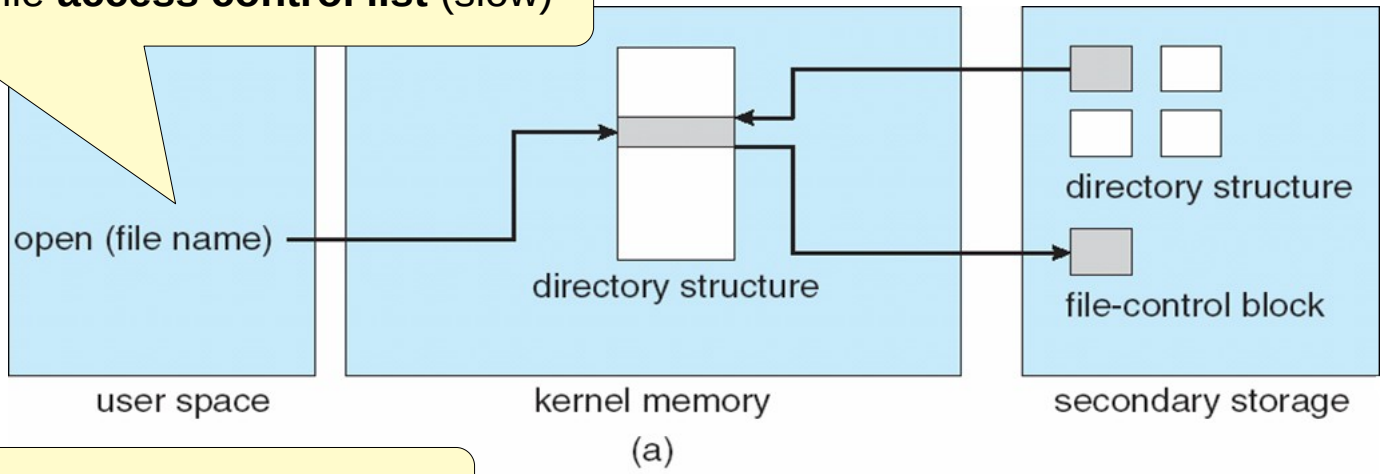
- Many **trade-offs** to consider
  - Global table is simple, but can be large
  - Access lists correspond to needs of users
    - Determining set of access rights for domain non-localized so difficult
    - Every access to an object must be checked
      - Many objects and access rights -> slow
  - Capability lists useful for localizing information for a given process
    - But revocation capabilities can be inefficient

Comparison of Implementations 2

- Most systems use **combination of access lists and capabilities**
  - First access to an object -> access list searched
    - If allowed, capability created and attached to process
      - Additional accesses need not be checked
    - After last access, capability destroyed
    - Consider file system with ACLs per file

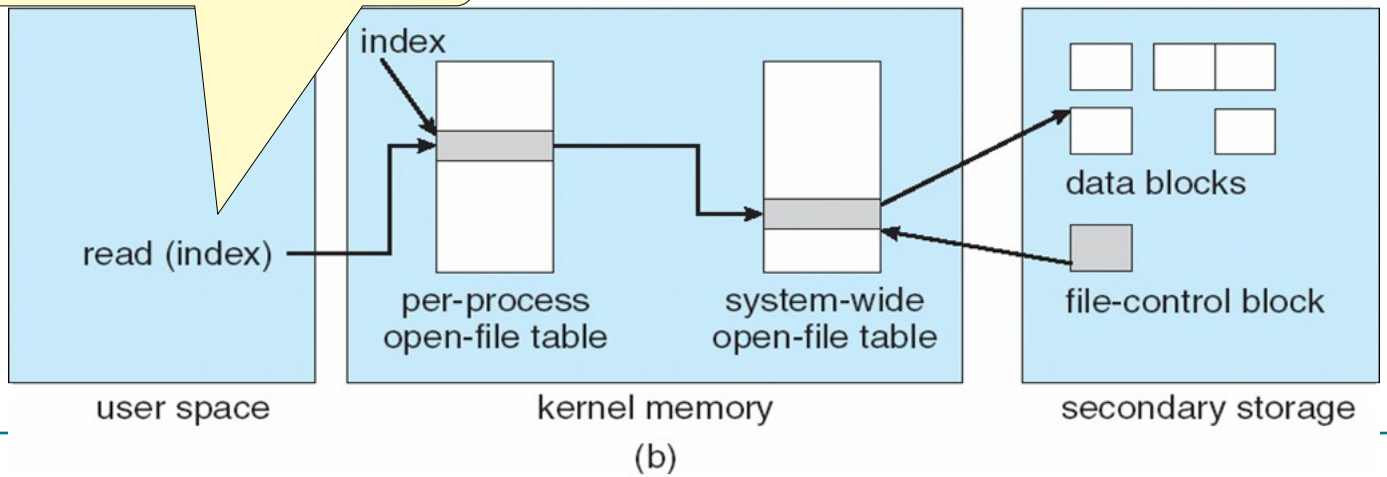
Hybrid example: File access control

1/ Per-file **access control list** (slow)



(a)

2/ File descriptor **capability** (fast)

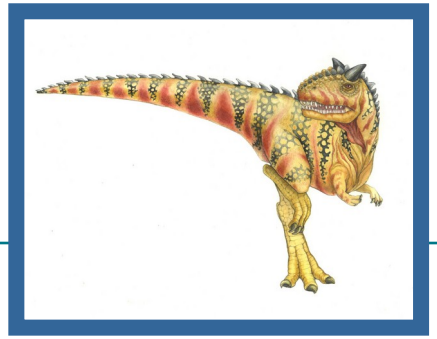


(b)

Revocation of Access Rights 2

- **Capability List** – Scheme required to locate capability in the system before capability can be revoked
  - **Reacquisition** – periodic delete, with require and denial if revoked
  - **Back-pointers** – set of pointers from each object to all capabilities of that object (Multics)
  - **Indirection** – capability points to global table entry which points to object – delete entry from global table
  - **Keys** – unique bits associated with capability, generated when capability created
    - Master key associated with object, key matches master key for access
    - Revocation – create new master key
    - Policy decision of who can create and modify keys – object owner or others?

17.8-9 Access Control Types



Discretionary Access Control (DAC) in UNIX (user,group,other)

```
jo@librem:/tmp/demo$ ls -l
total 8
-rw-rw-r-- 1 jo   jo   3 Nov 24 17:17 jo.txt
-rw-r--r-- 1 root root 5 Nov 24 17:18 root.txt
jo@librem:/tmp/demo$ cat jo.txt root.txt
j0
r00t
jo@librem:/tmp/demo$ chmod go-rw jo.txt root.txt
chmod: changing permissions of 'root.txt': Operation not permitted
jo@librem:/tmp/demo$ ls -l
total 8
-rw------- 1 jo   jo   3 Nov 24 17:17 jo.txt
-rw-r--r-- 1 root root 5 Nov 24 17:18 root.txt
jo@librem:/tmp/demo$ sudo chmod go-rw root.txt
jo@librem:/tmp/demo$ ls -l
total 8
-rw------- 1 jo   jo   3 Nov 24 17:17 jo.txt
-rw------- 1 root root 5 Nov 24 17:18 root.txt
jo@librem:/tmp/demo$ cat root.txt
cat: root.txt: Permission denied
```

Rules specified by and for **users**!

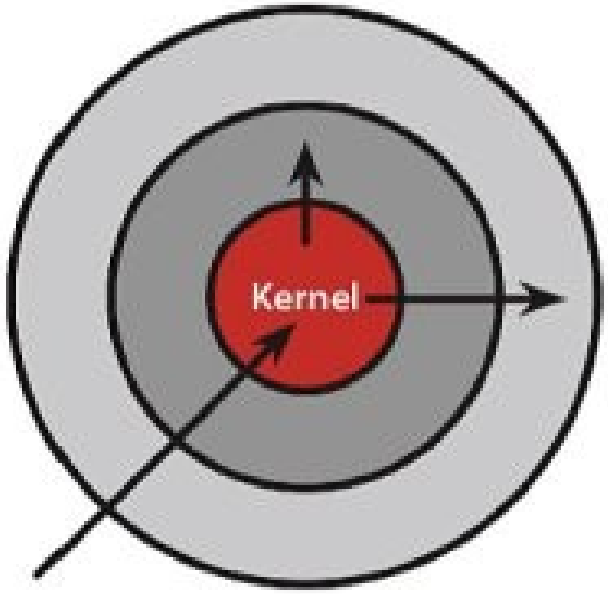
Mandatory Access Control (MAC)

- Operating systems traditionally had **discretionary access control (DAC)** to limit access to files and other objects (for example UNIX file permissions and Windows access control lists (ACLs))
  - Discretionary is a weakness – users / admins need to do something to increase protection
- Stronger form is **mandatory access control**, which *even root user can't circumvent*
  - Makes resources inaccessible except to their intended owners
  - Modern systems implement both MAC and DAC, with MAC usually a more secure, optional configuration (Trusted Solaris, TrustedBSD (used in macOS), SELinux/AppArmor), Windows Vista MAC)
- At its heart, **labels assigned to objects and subjects (including processes)**
  - When a subject requests access to an object, policy checked to determine whether or not a given label-holding subject is allowed to perform the action on the object

e.g., unclassified < secret < top-secret

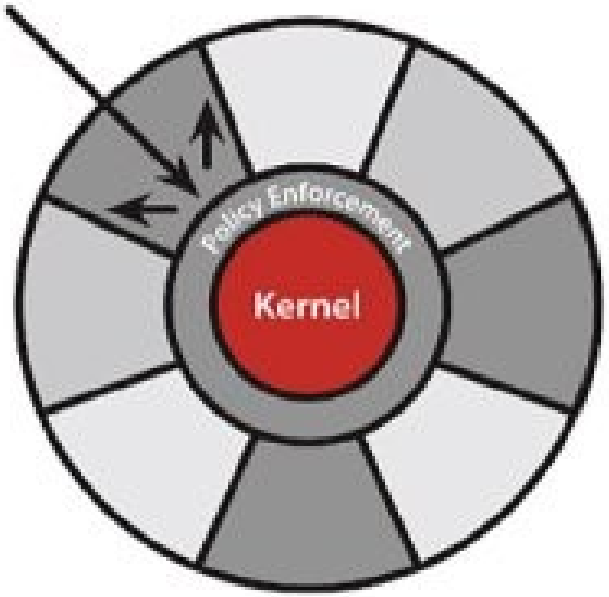
Important: **finer-grained**!
→ user (DAC) vs. process (MAC)

Discretionary vs. Mandatory Access



Discretionary Access Control

Once a security exploit gains access to priveleged system component, the entire system is compromised.



Mandatory Access Control

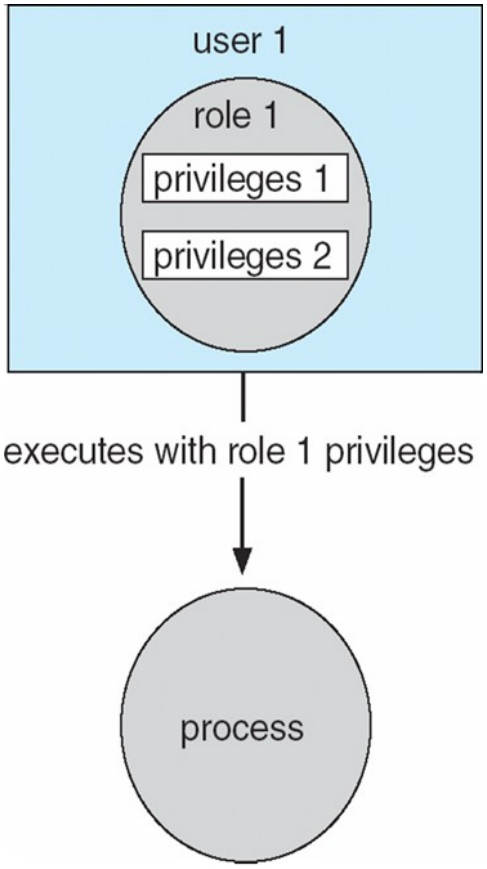
Kernel policy defines application rights, firewalling applications from compromising the entire system.

MAC better enforces **principle of least privilege!**

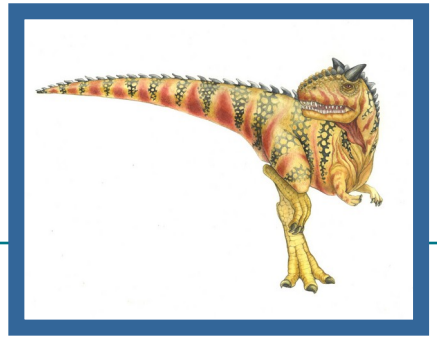
http://magazine.redhat.com/2007/05/04/whats-new-in-selinux-for-red-hat-enterprise-linux-5/

Role-based Access Control

- Protection can be applied to non-file resources (e.g., system calls)
- Oracle Solaris 10 provides **role-based access control** (**RBAC**) to implement least privilege
  - ***Privilege*** is right to execute system call or use an option within a system call
  - Can be assigned to processes
  - Users assigned ***roles*** granting access to privileges and programs
    - Enable role via password to gain its privileges
  - Similar to access matrix



17.10-11 Further confinement



“Capability”-Based Systems

Unfortunate naming, not really “capabilities”…

→ Rather **least-privilege** confinement

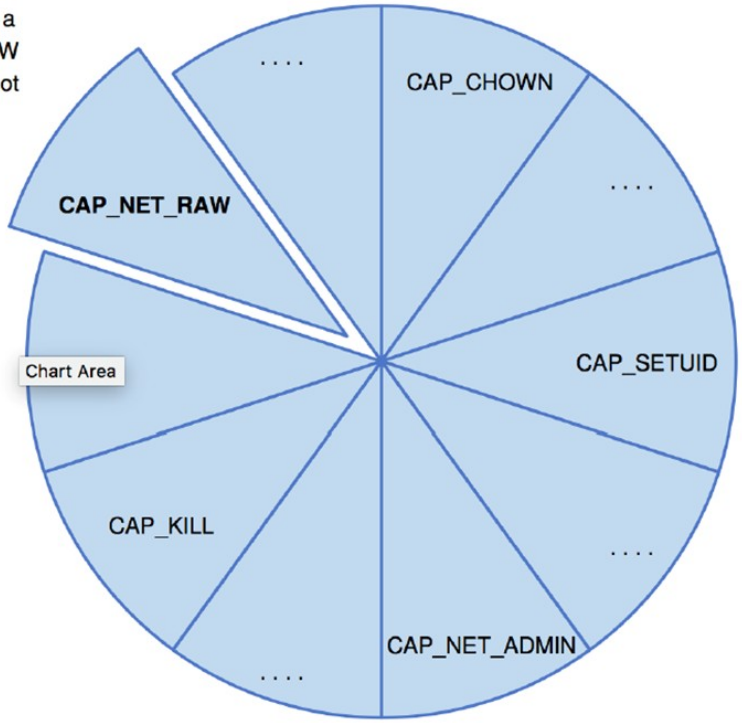
- (Hydra and CAP were first true capability-based systems)
- Now included in Linux, Android and others, based on POSIX.1e (that never became a standard)
  - Essentially slices up root powers into distinct areas, each represented by a bitmap bit
  - Fine grain control over privileged operations can be achieved by setting or masking the bitmap
  - Three sets of bitmaps – permitted, effective, and inheritable
    - Can apply per process or per thread
    - Once revoked, cannot be reacquired
    - Process or thread starts with all privs, voluntarily decreases set during execution
    - Essentially a direct implementation of the principle of least privilege
- An improvement over root having all privileges but inflexible (adding new privilege difficult, etc)

Capabilities in POSIX.1e

in the old model, even a simple `ping` utility would have required root privileges, because it opens a raw (ICMP) network socket

capabilities can be thought of as "slicing up the powers of root" so that individual applications can "cut and choose" only those privileges they actually require

with capabilities, ping can run as a normal user, with CAP_NET_RAW set, allowing it to use ICMP but not other extra privileges



Generalization: Sandboxing

- Running process in **limited environment**
- Impose set of irremovable restrictions early in startup of process (e.g., at fork(), before `main()`)
- Process then unable to access any resources beyond allowed set
- Java and .NET implement at a virtual machine level
- Other systems implement with **MAC** (e.g., SELinux, AppArmor)

Demo: Webserver containment with SELinux/Apparmor

jo@librem: ~/Documents/doc/presentations/os21/srv-example

```
jo@librem:~/Documents/doc/presentations/os21/srv-example$ cat /etc/apparmor.d/my.srv.example
#include <tunables/global>

/home/jo/Documents/doc/presentations/os21/srv-example/cgi-bin/hello.py {
  #include <abstractions/base>
  #include <abstractions/python>

  network inet,

  /home/jo/Documents/doc/presentations/os21/srv-example/* r,
  /home/jo/Documents/doc/presentations/os21/srv-example/cgi-bin/* r,
}
jo@librem:~/Documents/doc/presentations/os21/srv-example$ curl http://0.0.0.0:8000/cgi-bin/hello
.py?file=../../../../../../../../../../../../../../etc/passwd

<html><body>
<p>serving contents for file '../../../../../../../../../../../../../../etc/passwd':</p>
<code>
jo@librem:~/Documents/doc/presentations/os21/srv-example$ dmesg | tail | grep passwd
[ 4884.608929] audit: type=1400 audit(1637786835.009:725): apparmor="DENIED" operation="open" pr
ofile="/home/jo/Documents/doc/presentations/os21/srv-example/cgi-bin/hello.py" name="/etc/passwd
" pid=6934 comm="hello.py" requested_mask="r" denied_mask="r" fsuid=1000 ouid=0
jo@librem:~/Documents/doc/presentations/os21/srv-example$
```

Example only, no need to know commands!

Further Confinement Mechanisms

- **System-call filtering**
  - Like a firewall, for system calls
  - Can also be deeper –inspecting all system call arguments
  - Linux implements via SECCOMP-BPF (Berkeley packet filtering)
- **System integrity protection (SIP)**
  - Introduced by Apple in macOS 10.11
  - Restricts access to system files and resources, *even by root*
  - Uses extended file attribs to mark a binary to restrict changes, disable debugging and scrutinizing
  - Also, only code-signed kernel extensions allowed and configurably only code-signed apps

Code signing

Used by all major OSs nowadays (e.g., Debian SecureApt, etc.)

Code signing allows a system to trust a program or script by using **crypto hash to have the developer sign the executable**

- So code as it was compiled by the author
- If the code is changed, signature invalid and (some) systems disable execution

Can also be used to disable old programs by the operating system vendor (such as Apple) cosigning apps, and then invalidating those signatures so the code will no longer run

Two-edged sword: User protection vs. user freedom...

Not-covered: 17.12
“Language-based protection”

Wrap-up Part 2: Protection

- Software isolation rooted in the CPU
  - → protection rings (+ virtual memory)
- Good protection **mechanisms** allow for versatile **policies**
  - → e.g., expressed in access control matrix, implement with
    - Access Control Lists (ACL)
    - Capabilities
- General: strive for **principle of least privilege!**
  - → e.g., sandboxing with MAC, system call filtering, etc.