CHAPTER 17 - SYSTEM 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 systems
- Protection to mitigate attacks

GOALS OF PROTECTION

GOALS OF PROTECTION

- In one protection model, computer consists of a collection of objects, 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

mechanisms are distinct from **policies**.

- Mechanisms determine how something will be done
- Policies decide what will be done.

The separation of policy and mechanism is important for flexibility.

PRINCIPLES OF PROTECTION

PRINCIPLES OF PROTECTION

- Guiding principle – **principle of least privilege**
  - Programs, users and systems should be given just enough privileges to perform their tasks
  - Limits 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
  - “Need to know” similar concept with access to data

PRINCIPLES OF PROTECTION

- Must consider “grain” aspect
  - Rough-grained privilege management easier, simpler, but least privilege now done in large chunks
    - For example, traditional Unix processes either have abilities of the associated user, or of root
  - Fine-grained management more complex, more overhead, but more protective
    - File ACL lists, RBAC
- Domain can be user, process, procedure

PROTECTION RINGS

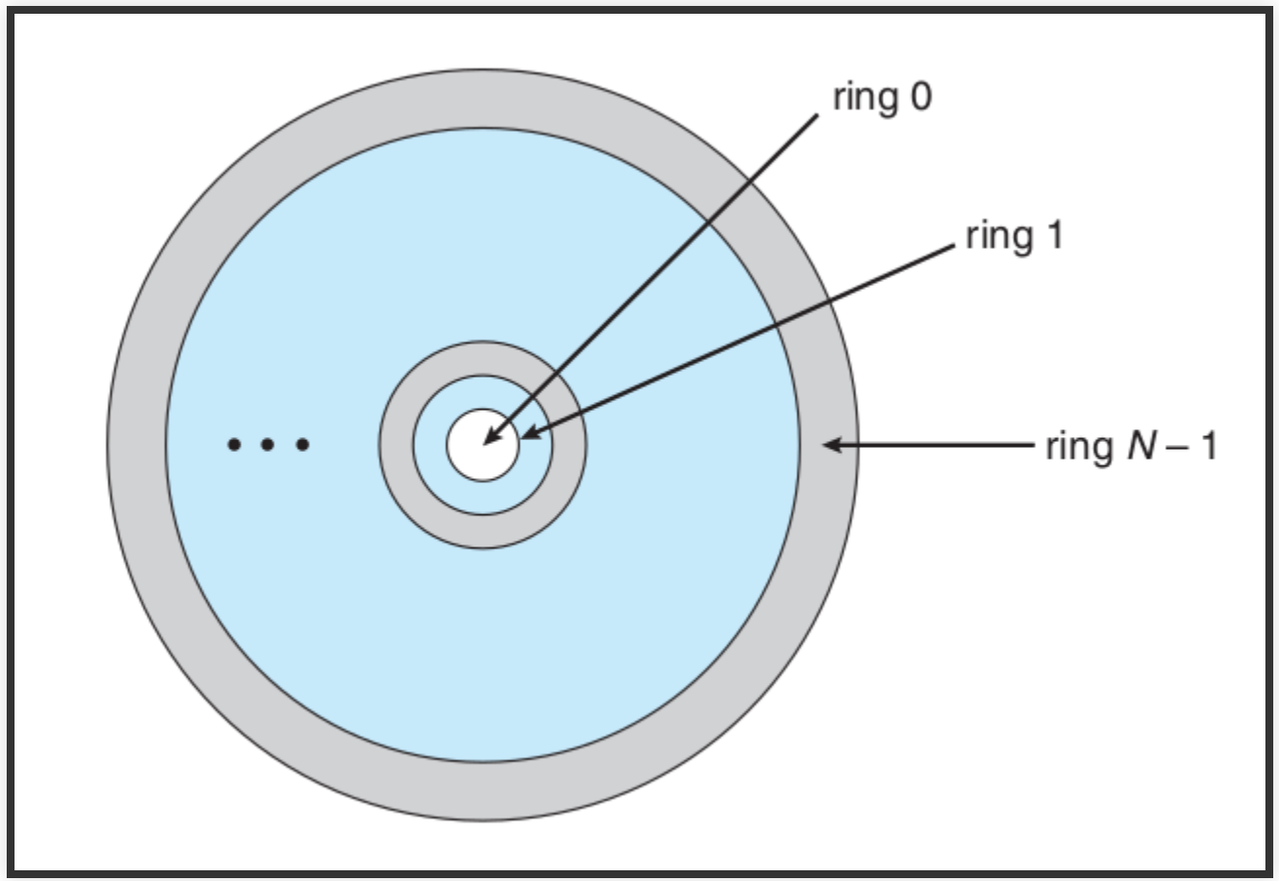
PRIVILEGE SEPARATION

- Kernel manages access to system resources and hardware
  - By definition: must run with a higher level of privileges than user processes.

To carry out this privilege separation, hardware support is required.

PRIVILEGE SEPARATION

Popular model: Protection Rings



INTEL ARCHITECHTURE

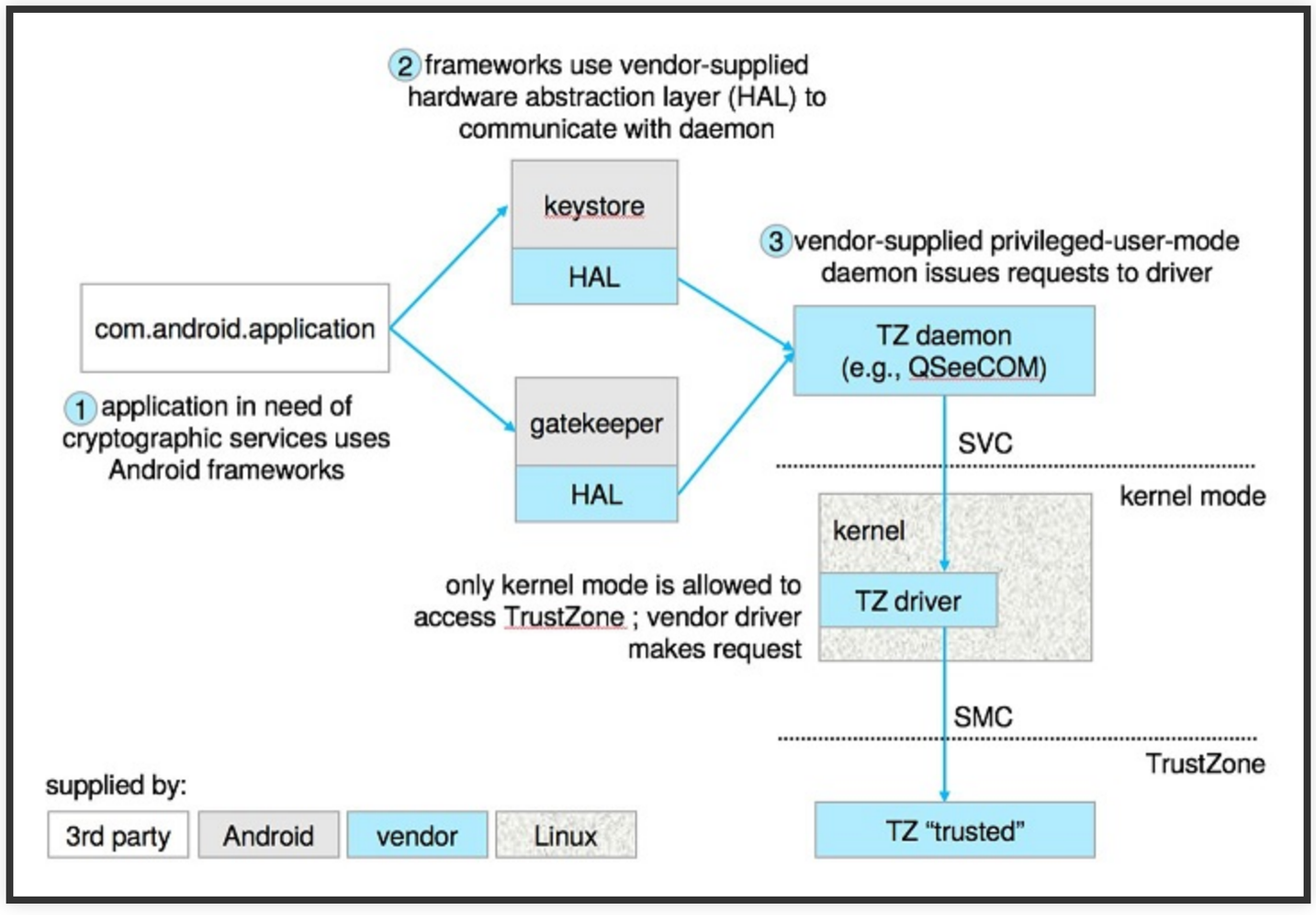
Intel architectures follow this model, placing user mode code in ring 3 and kernel mode code in ring 0.

Distinction is made by two bits in the special EFLAGS register.

ARM V7 ARCHITECHTURE

- Initially allowed only USR and SVC mode
- In ARM v7 processors, TrustZone (TZ) was introduced providing an additional ring.
- Highly restricted access - even the kernel itself has no access to the on-chip crypto key
  - Access through `Secure Monitor Call` (SMC), which is only usable from kernel mode.

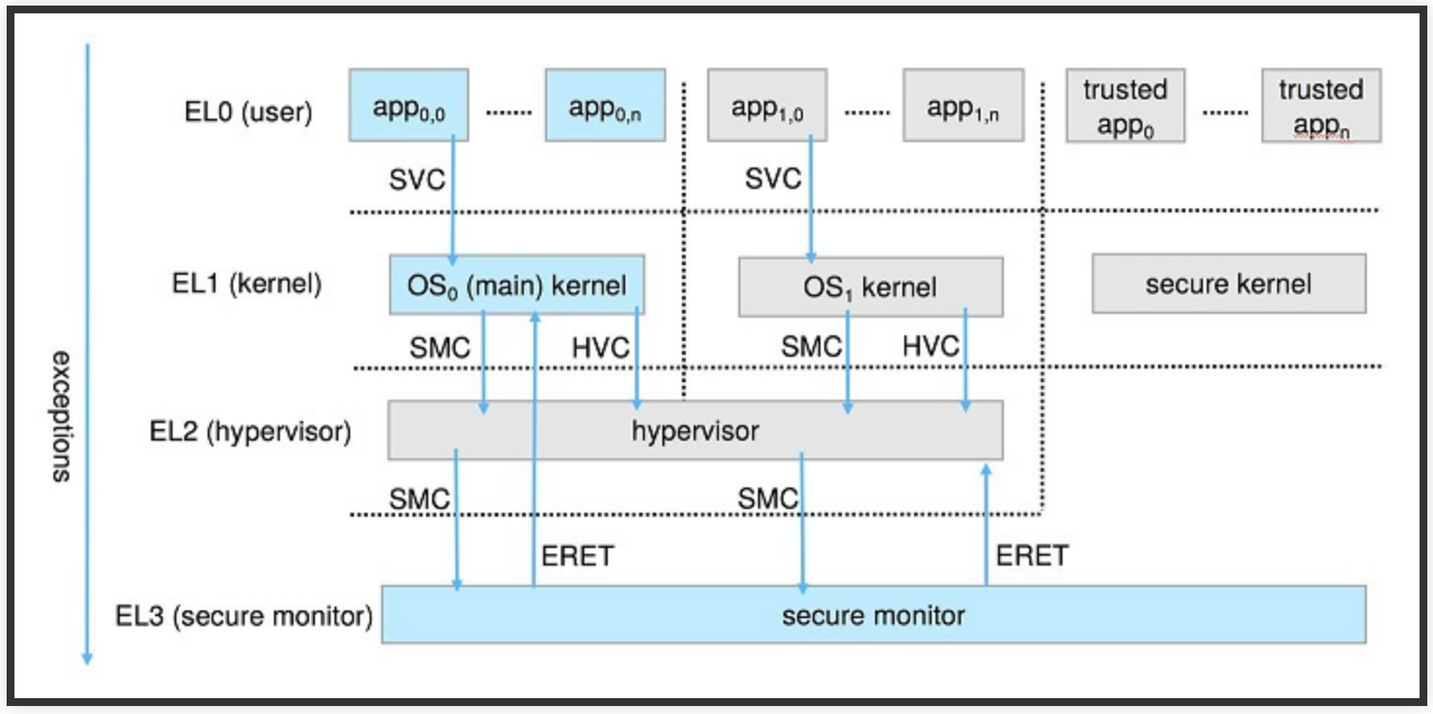
ARM TRUSTED ZONES



ARM V8 ARCHITECTURE

- Now 4 levels called “exception levels,” numbered EL0 through EL3.
  - User mode runs in EL0
  - kernel mode in EL1
  - EL2 is reserved for hypervisors
  - EL3 (the most privileged) is reserved for the secure monitor (the TrustZone layer)

ARM V8 ARCHITECTURE



DOMAIN OF PROTECTION

PROCESSES AND OBJECTS

- Processes
- Objects
  - Hardware objects (CPU, Memory segments, printers, disk, etc)
  - Software objects (files, programs, semaphores)

Operations depends on object (Fx CPU can only execute, mem words write or read)

RULE OF THUMB

- A process should be allowed to access only those resources for which it has authorization

> **Need to know principle**: At any time, a process should be able to access only those resources that is currently required to complete its task

NEED TO KNOW PRINCIPLE - EXAMPLE

When process p invokes procedure A ( ) it should be allowed to access only

- its own variables and the formal parameters passed to it

Not allowed to access:

- all the variables of process p.

NEED TO KNOW PRINCIPLE - EXAMPLE

When process p invokes a compiler to compile a particular file. It should be able to access:

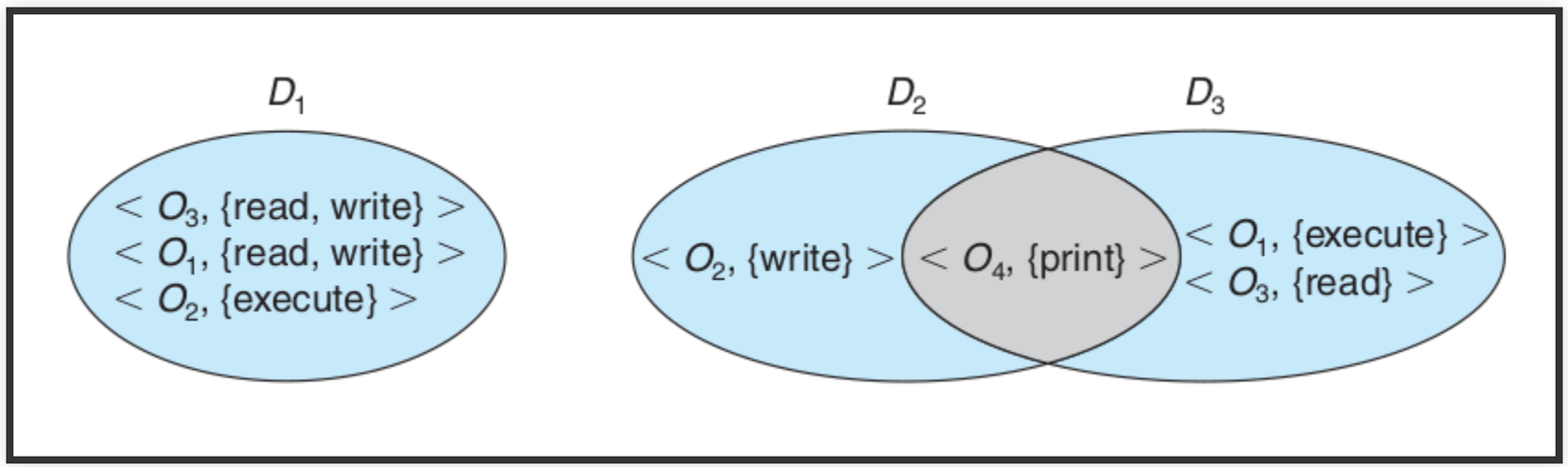
- Only a well-defined subset of files (such as the source file, output object file, and so on)

Not allowed to access:

- Arbitrarily files

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 IMPLEMENTATION (UNIX)

- 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)

ANDROID APPLICATION IDS

- Distinct user ID s are provided on a per-application basis.
- On installation, the installd daemon assigns app
  - a distinct user ID (UID) and group ID (GID)
  - a private data directory (`/data/data/<app-name>`) 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.

ACCESS MATRIX

ACCESS MATRIX

- 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 $Object_j$

ACCESS MATRIX

object / domain	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

- 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

USE OF ACCESS MATRIX

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

ACCESS MATRIX 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

- Copy and Owner applicable to an object
- Control applicable to domain object

ACCESS MATRIX W. 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)

ACCESS MATRIX W. 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)

ACCESS MATRIX W. CONTROL

object / domain	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 control
D_3		read	execute					
D_4	write		write		switch			

USE OF ACCESS MATRIX

- **Access matrix** design separates mechanism from policy
  - **Mechanism**
    - Operating system provides access-matrix + rules
    - If ensures that the matrix is only manipulated by authorized agents and that rules are strictly enforced
  - **Policy**
    - User dictates policy
    - Who can access what object and in what mode

GENERAL CONFINEMENT PROBLEM

General confinement problem: Guaranteeing that that no information initially held by an object can migrate outside of its execution environment

General unsolvable

IMPLEMENTATION OF THE ACCESS MATRIX

IMPLEMENTATION OF AM

Generally, a sparse matrix

We will consider 4 ways

- Global table
- Access lists for objects
- Capability list for domains
- Lock-key

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)

ACCESS LISTS 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

ACCESS LISTS FOR OBJECTS

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
```

CAPABILITY LIST FOR DOMAINS

- Instead of object-based, list is domain based
- Capability list for domain is list of objects together with operations allows on them
- 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 FOR DOMAINS

- 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

LOCK-KEY

- Compromise between access lists and capability lists
- Each object has list of unique bit patterns, called locks
- Each domain as list of unique bit patterns called keys
- Process in a domain can only access object if domain has key that matches one of the locks

COMPARISON OF IMPLEMENTATIONS

- 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

COMPARISON OF IMPLEMENTATIONS

- Capability lists useful for localizing information for a given process
  - But revocation capabilities can be inefficient
- Lock-key effective and flexible, keys can be passed freely from domain to domain, easy revocation

COMPARISON OF IMPLEMENTATIONS

- 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

REVOCATION OF ACCESS RIGHTS

REVOCATION OF ACCESS RIGHTS

- Various options to remove the access right of a domain to an object
  - Immediate vs. delayed
  - Selective vs. general
  - Partial vs. total
  - Temporary vs. permanent

REVOCATION OF ACCESS RIGHTS

- Access List – Delete access rights from access list
  - Simple – search access list and remove entry
  - Immediate, general or selective, total or partial, permanent or temporary

REVOCATION OF ACCESS RIGHTS

- 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
  - **Indirection** – capability points to global table entry which points to object – delete entry from global table, not selective

REVOCATION OF ACCESS RIGHTS

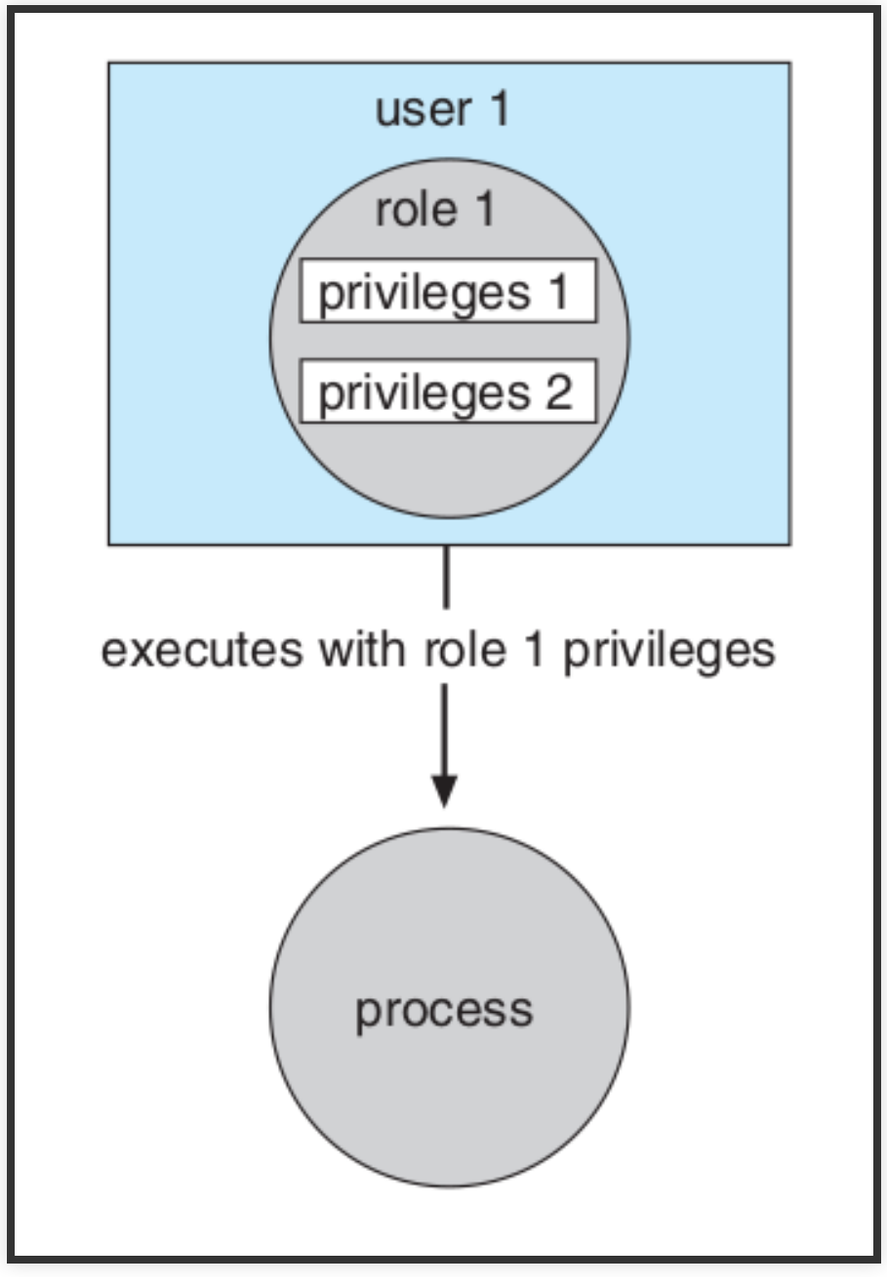
- Keys – unique bits associated with capability, generated when capability created
  - Master key associated with object, key matches master key for access
  - Revocation – create new master key
  - Policy decision of who can create and modify keys – object owner or others?

ACCESS CONTROL

ACCESS CONTROL

- Protection can be applied to non-file resources
- 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

RBAC IN SOLARIS 10



MANDATORY ACCESS CONTROL (MAC)

TRADITIONALLY

OS's traditionally used *discretionary access control* (DAC)

- Access is controlled based on the identities of individual users or groups
- Example: File permissions in Unix

Key weaknesses

- Allows the owner of a resource to set or modify its permissions
- Unlimited access allowed for the administrator or root user.

MANDATORY ACCESS CONTROL

MAC is enforced as a system policy that even the root user cannot modify

The restrictions imposed by MAC policy rules are more powerful than the capabilities of the root user and can be used to make resources inaccessible to anyone but their intended owners.

MANDATORY ACCESS CONTROL

Base idea: **Labels**

- A label is an identifier (usually a string) assigned to an object (files, devices, and the like).
- May also be applied to subjects (actors, such as processes).

  When a subject request to perform operations on the objects:

- OS checks requests defined in a policy, which dictates whether or not a given label holding subject is allowed to perform the operation on the labeled object.

MAC EXAMPLE

Labels: "unclassified", "secret", and "top secret".

A user with "secret" clearance will be able to create similarly labeled processes, which will then have access to "unclassified" and "secret" files, but not to "top secret" files.

"top secret" files the operating system would filter out of all file operations

CAPABILITY-BASED SYSTEMS

CAPABILITY-BASED SYSTEMS

Introduced in 1970, in Hydra and CAP (reseach systems)

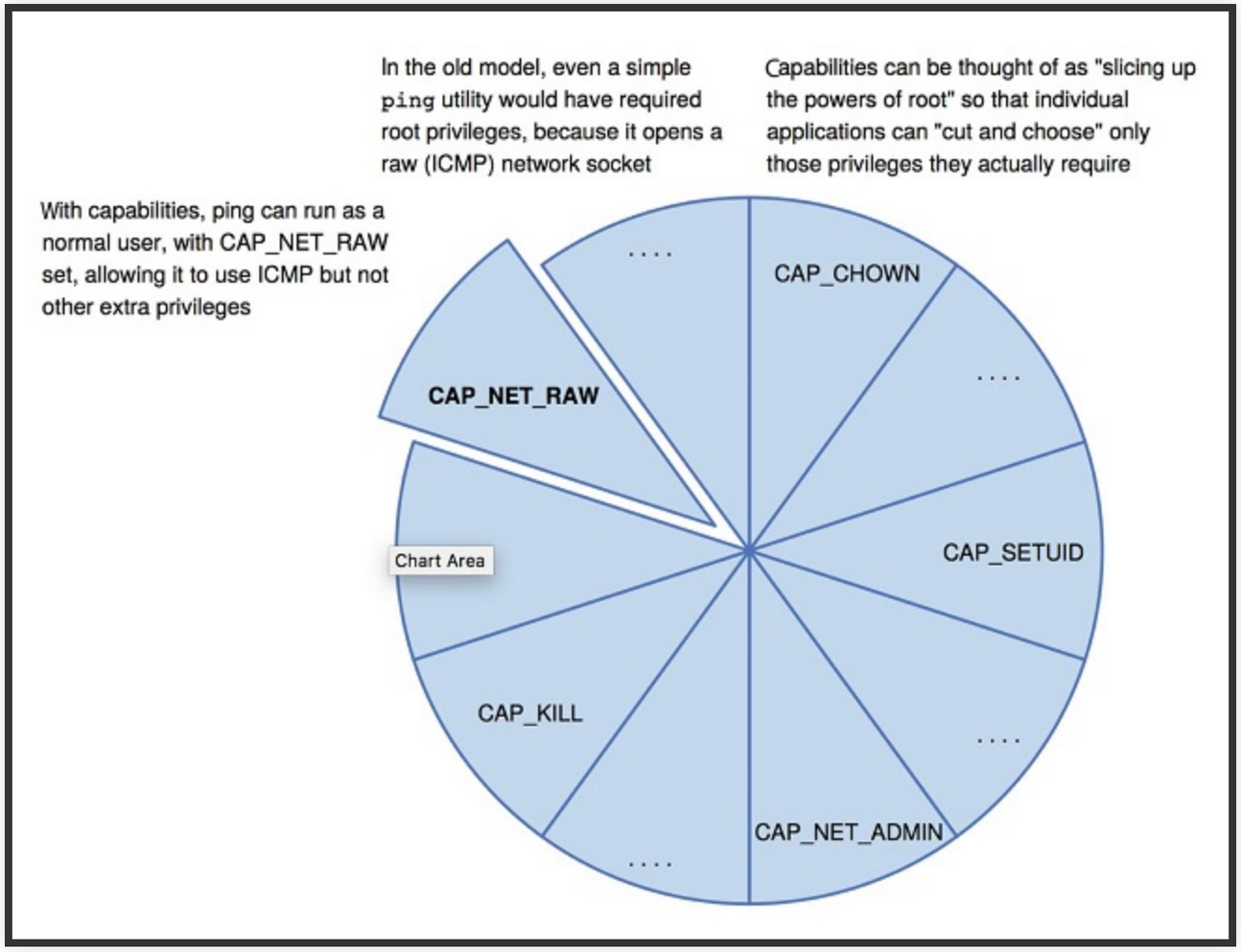
Here, two more modern approaches

LINUX CAPABILITIES

- Linux uses capabilities to address the limitations of the UNIX model
- Adopted POSIX standard

- Linux's capabilities "slice up" the powers of root into distinct areas, each represented by a bit in a bitmask
- In practice, three bitmasks are used—denoting the capabilities **permitted**, **effective**, and **inheritable**.
- Bitmasks can apply on a per-process or a per-thread basis.

LINUX CAPABILITIES



DARWIN ENTITLEMENTS

- Apple's system protection takes the form of entitlements.
- Entitlements are declaratory permissions—XML property list stating which permissions are claimed as necessary by the program
- To prevent programs from arbitrarily claiming an entitlement, Apple embeds the entitlements in the code signature
- Once loaded, a process has no way of accessing its code signature.
- Verifying an entitlement is a simple string-matching operation.

OTHER PROTECTION IMPROVEMENT METHODS

SYSTEM INTEGRITY PROTECTION

Apple introduced in macOS 10.11 a new protection mechanism called **System Integrity Protection** (SIP).

Darwin-based OSs use SIP to restrict access to system files and resources - even the root user cannot tamper with them.

SIP uses extended attributes on files to mark them as restricted and further protects system binaries so that they cannot be debugged/scrutinized/tampered with.

Root user can still manage other users' files, install and remove programs, but not in any way that would replace or modify operating-system components.

SYSTEM-CALL FILTERING

Add code to the kernel to perform an inspection at the system-call gate, restricting a caller to a subset of system calls deemed safe or required for that caller's function.

Alternative: inspects the arguments of each system call

challenge encountered with both approaches is keeping them as flexible as possible while at the same time avoiding the need to rebuild the kernel when changes or new filters are required

⇒ decouple the filter implementation from the kernel itself.

The kernel need only contain a set of callouts, which can then be implemented in a specialized driver (Windows), kernel module (Linux), or extension (Darwin).

SANDBOXING

Sandboxing involves running processes in environments that limit what they can do.

Rather than give that process the full set of system calls its privileges would allow, we impose an irremovable set of restrictions on the process in the early stages of its startup.

The process is then rendered unable to perform any operations outside its allowed set.

CODE SIGNING

At a fundamental level, how can a system “trust” a program or script?

Code signing is the digital signing of programs and executables to confirm that they have not been changed since the author created them.

It uses a cryptographic hash to test for integrity and authenticity. Code signing is used for operating-system distributions, patches, and third-party tools alike.

LANGUAGE-BASED PROTECTION

COMPILER-BASED ENFORCEMENT

Specifying the desired control of access to a shared resource in a system is making a declarative statement about the resource.

This kind of statement can be integrated into a language by an extension of its typing facility.

COMPILER-BASED ENFORCEMENT

1. Protection needs are simply declared, rather than programmed as a sequence of calls on procedures of an operating system.
2. Protection requirements can be stated independently of the facilities provided by a particular operating system.
3. The means for enforcement need not be provided by the designer of a subsystem.
4. A declarative notation is natural because access privileges are closely related to the linguistic concept of data type.

RUN-TIME-BASED ENFORCEMENT - PROTECTION IN JAVA

STACK INSPECTION

protection domain:	untrusted applet	URL loader	networking
socket permission:	none	*.lucent.com:80, connect	any

class:	gui:
	...
	get(url);
	open(addr);
	...
		get(URL u):
		...
		doPrivileged {
		open('proxy.lucent.com:80');
		}
		<request u from proxy>
		...
			open(Addr a):
			...
			checkPermission
			(a, connect);
			connect (a);
			...

QUESTIONS

BONUS

Exam question number 10: **Protection**