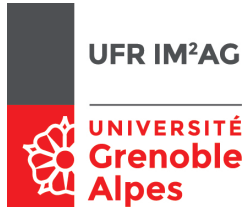


Software security, secure programming

Access Control in a Nutshell . . .

Master M2 Cybersecurity

Academic Year 2024 - 2025

Access Control

Given a set of

- subjects: (human) users,
  SW/HW entities (process, application, “tab”, device, etc.)
- objects: SW entities (file, application, data base, software component, method)
  HW entities (device, peripheral, memory area, etc.)

Specify and enforce an **access contyrol policy** telling

*which **actions** a **subject** can perform over an **object***

Where

- action: an access primitive (open, close, read, write, execute), a more specific operation (method call, etc.), etc.

Access Control

Given a set of

- subjects: (human) users,
  SW/HW entities (process, application, “tab”, device, etc.)
- objects: SW entities (file, application, data base, software component, method)
  HW entities (device, peripheral, memory area, etc.)

Specify and enforce an **access contyrol policy** telling

*which **actions** a **subject** can perform over an **object***

Where

- action: an access primitive (open, close, read, write, execute), a more specific operation (method call, etc.), etc.

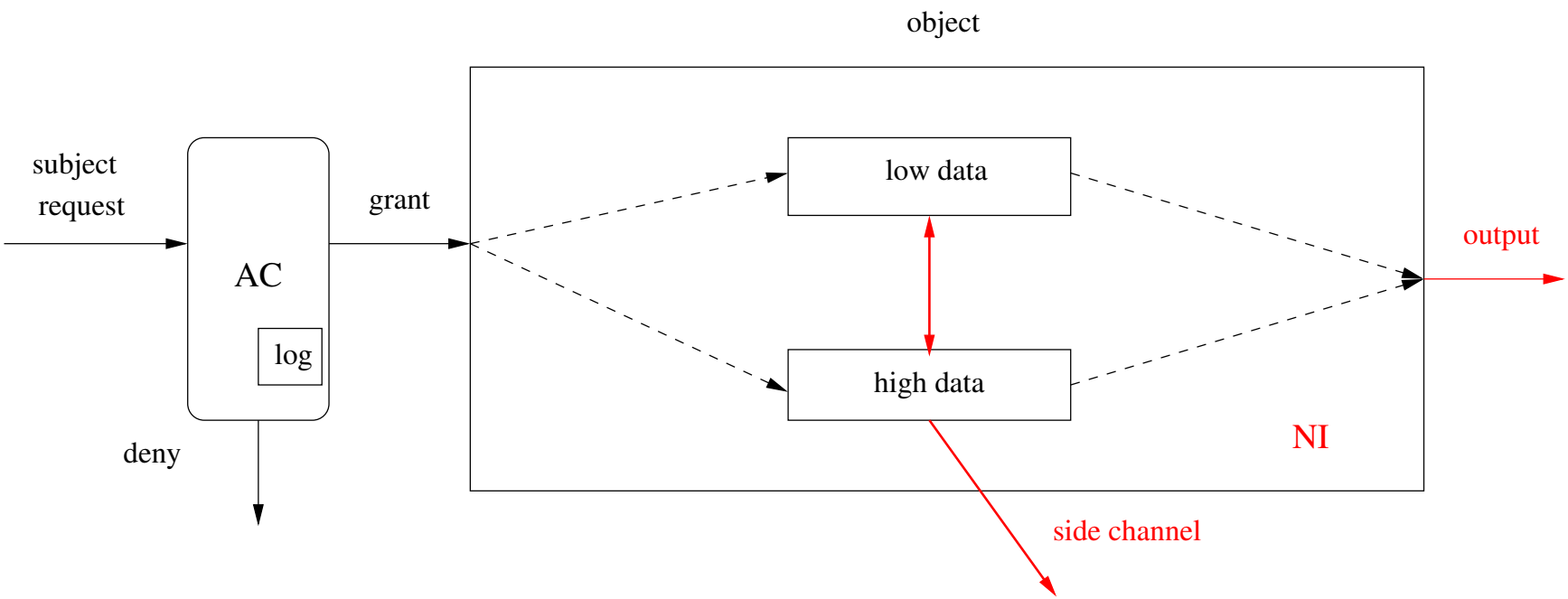
Access Control implies/encompasses

- identification + authentication (recognize and proof subject identity)
- audit logs, accountability (keep tracks of access granted/refused)

Access Control (AC) *vs* Non-Interference (NI)

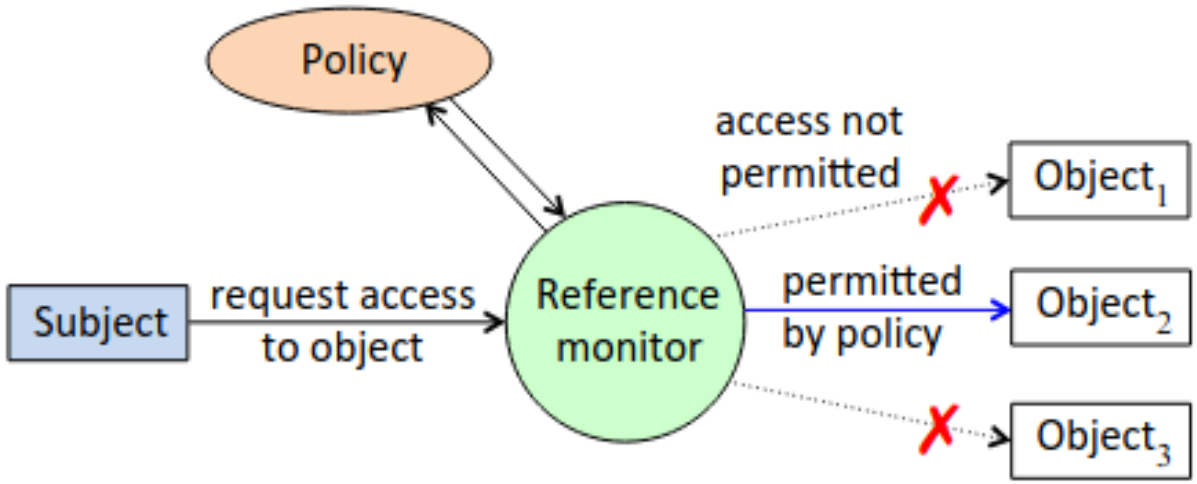
NI : how information flows **within** an object (once its access has been granted to a subject)

AC : which **operations** are granted to an object from a subject (consider only the *borders* of the objects)



Reference Monitor

An abstract mechanism to enforce an **access control policy**:
allows/deny **subjects** to perform **operations** on **objects**



Expected key properties

- non **bypassable** by an attacker
- **evaluable** w.r.t. soundness and completeness
- always **invoked**
- **tamper-proof** (cannot be hijacked)

Rk: assumes a reliable authentication system . . .

Some related notions

Trusted Computing Base (TCB)

Any AC enforcement mechanisms should rely on a trusted subset of hardware/code/data . . . the TCB

A good design practice:

Keep the TCB as small as possible!

Some related notions

Trusted Computing Base (TCB)

Any AC enforcement mechanisms should rely on a trusted subset of hardware/code/data . . . the TCB

A good design practice:

Keep the TCB as small as possible!

Principle of Least Priviledge

*Every program and every privileged user of the system should operate using the **least amount of privilege necessary** to complete the job. [J Saltzer, 1974]*

Some related notions

Trusted Computing Base (TCB)

Any AC enforcement mechanisms should rely on a trusted subset of hardware/code/data . . . the TCB

A good design practice:

Keep the TCB as small as possible!

Principle of Least Priviledge

*Every program and every privileged user of the system should operate using the **least amount of privilege necessary** to complete the job. [J Saltzer, 1974]*

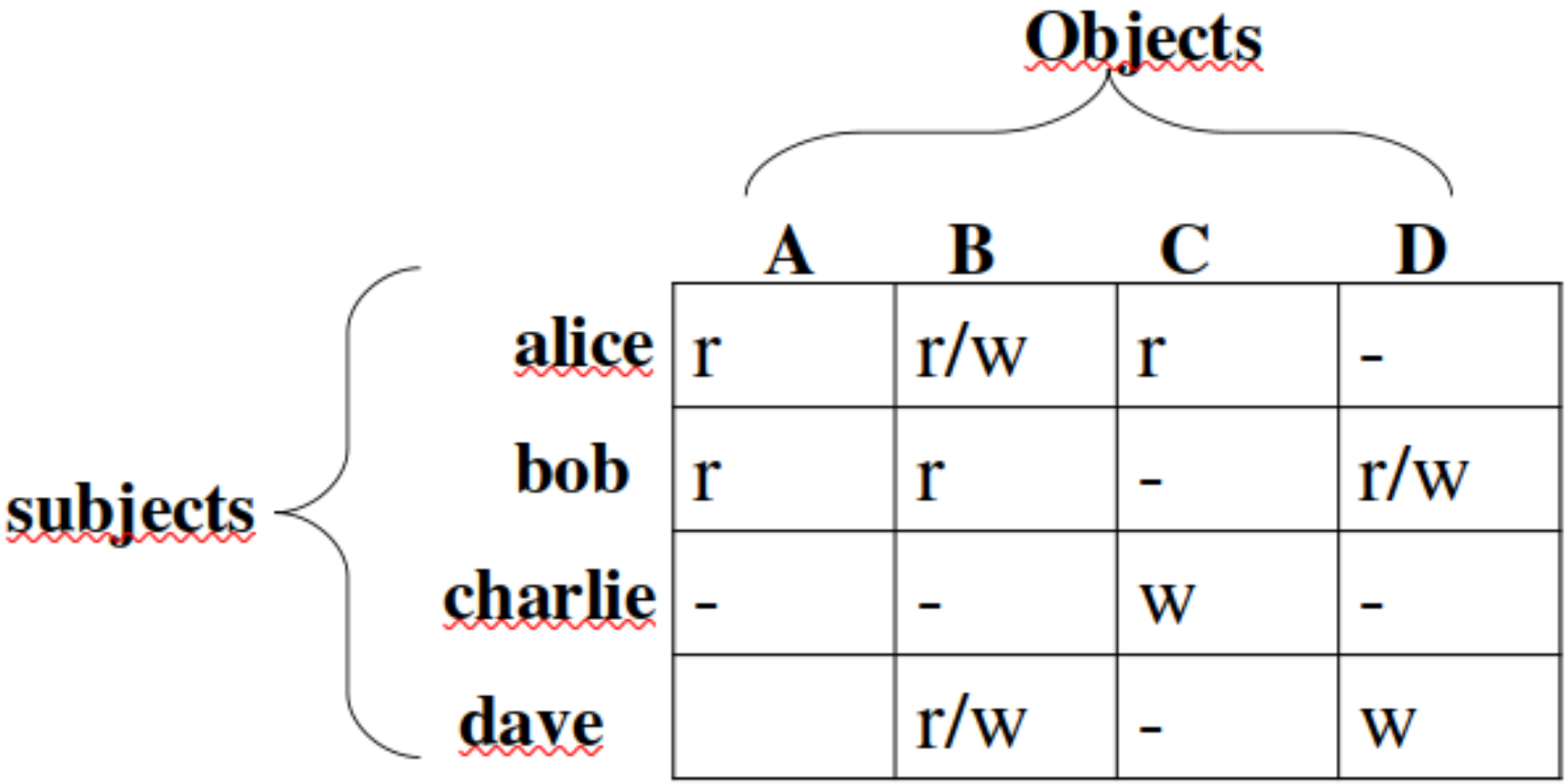
Sandboxing

A tightly controlled set of resources for guest programs to run in.

- an effective AC mechanism . . .
- rather coarse-grained (the *object* is the sandbox)
- not well-adaptaded for **sharing** (limited) sets of permissions over multiples objects . . .

Access Control Matrix

A classical model to specify rights and permissions



subjects	A	B	C	D
alice	r	r/w	r	-
bob	r	r	-	r/w
charlie	-	-	w	-
dave		r/w	-	w

Remark: users can be gathered to form **groups** . . .

Access Control Lists (ACLs)

Break down the AC matrix by columns:

- each objet gets a set of (user, right) pairs
- ex: object $A = \{(bob, r/w), (alice, w)\}$

Properties

- well adapted for numerous applications (e.g, filesystems)
- lists may become large in practive
- does not easily support **delegation** and sharing ...

Capabilities

Break down the AC matrix by rows:

- each user gets a set of (object, right) pairs
- ex: $Alice = \{(A, r/w), (B, w), (C, r)\}$

Properties

- **capability** = communicable "token" associated to objects ($\sim$ *handler*)
- well adapted for delegation: rights are **associated** to objects

Capabilities

Break down the AC matrix by rows:

- each user gets a set of (object, right) pairs
- ex: $Alice = \{(A, r/w), (B, w), (C, r)\}$

Properties

- **capability** = communicable "token" associated to objects ($\sim$ *handler*)
- well adapted for delegation: rights are **associated** to objects

Remark: subjects should be prevented to forge capabilities

- store them in a protected address space
- use special tags or HW supports
- ciphering/hashing with crypto primitives
- etc.

ID-based vs capability based access control?

Capabilities

Owning a "ticket" allows to perform some action on an objet

- The owner's id is not relevant
- tickets can be transfered for delegation
- tickets must be unforgeable

IDs

Having an authorized ID allows to perform some action on an objet

- A list of "per ID" authorization is available and maintened
- IDs must be unspoofable

Discretionary Access Control (DAC)

No central authority to grant/deny access rights

- permissions are “owned” by users
- users are able to transfer permissions to each others

Rarely implemented as a whole . . . (quite often combined with MAC)

Mandatory Access Control (MAC)

$\exists$ a central security policy controller

- only the central authority may transfer/modify permissions
- well adapted to multi-level security rules (lattice of information domains)

Numerous implementations within operating systems . . .
(sometimes combined with DAC)

- Unix: user rights (DAC) + *su* mode (MAC)
- SELinux, AppArmor (Ubuntu), Microsoft MIC, TrustBSD (BSD, MacOS), etc.

Role-base Access Control (RBAC)

$\exists$ Define the access control policy based on (subject) **roles**

1. roles can be assigned to subjects, according to some authorizations
2. object accesses (i.e., permissions) are granted to roles
3. a subject can exercise a permission only if is granted to its active role

- can be extended with **role hierarchies** and **constraints** (permission inheritances, restricted by constraints, e.g, *separation of duties*)
- flexible, allows to combine MAC and DAC
- well adapted for large organisations/companies/administrations . . .

Access Control and (programming) languages

Specification languages for AC

- numerous logic-based formalisms, allow to prove AC properties!
- some XML extensions (XACLM, XrML) to “implement” AC policy descriptions
- etc.

Access Control and (programming) languages

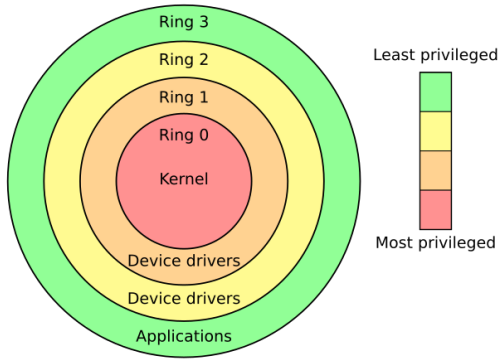
Specification languages for AC

- numerous logic-based formalisms, allow to prove AC properties!
- some XML extensions (XACLM, XrML) to “implement” AC policy descriptions
- etc.

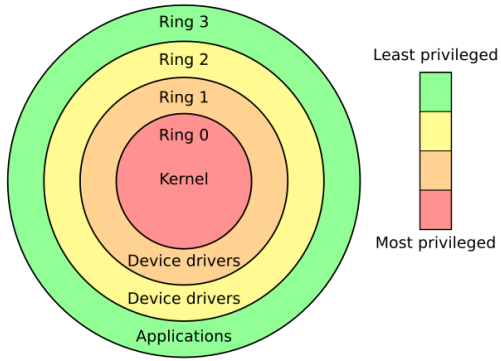
Using AC primitives in a program

1. use the primitives available at the OS level
   - (very) coase-grained, only inter-process AC
   - relies on a huge TCB (the OS itself!)
2. use dedicated primitives (when available in the PL)
   - (basic) attributes to restrain code/data access (private, protected, etc.)
   - Java: allows to mitigate access to a class/method the class “origin” (JPSA), or wrt the “user” (JAAS)
   - some available libraries un Python . . .
   - fine-grained AC primitives available in Swift (Apple)

HW based AC mechanisms: protection rings



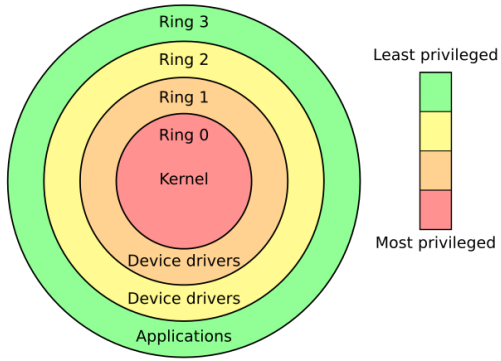
HW based AC mechanisms: protection rings



Multics Operating System

- Linux precursor, introduced in 1964 . . .
- 8 rings, each memory segment associated to one ring
- access allowed to current ring and higher ones
- predefined ring intervals, with lightweight domain switching

HW based AC mechanisms: protection rings



Multics Operating System

- Linux precursor, introduced in 1964 . . .
- 8 rings, each memory segment associated to one ring
- access allowed to current ring and higher ones
- predefined ring intervals, with lightweight domain switching

More recently

- still available in modern processors ($\sim$ 4 rings)
- X86 /OS2 : kernel / privileged code / unprivileged code
- ARM V8: secure monitor/firmware / OS / hypervisor / applications