Lec 19: Discretionary Access Control

CSED415: Computer Security

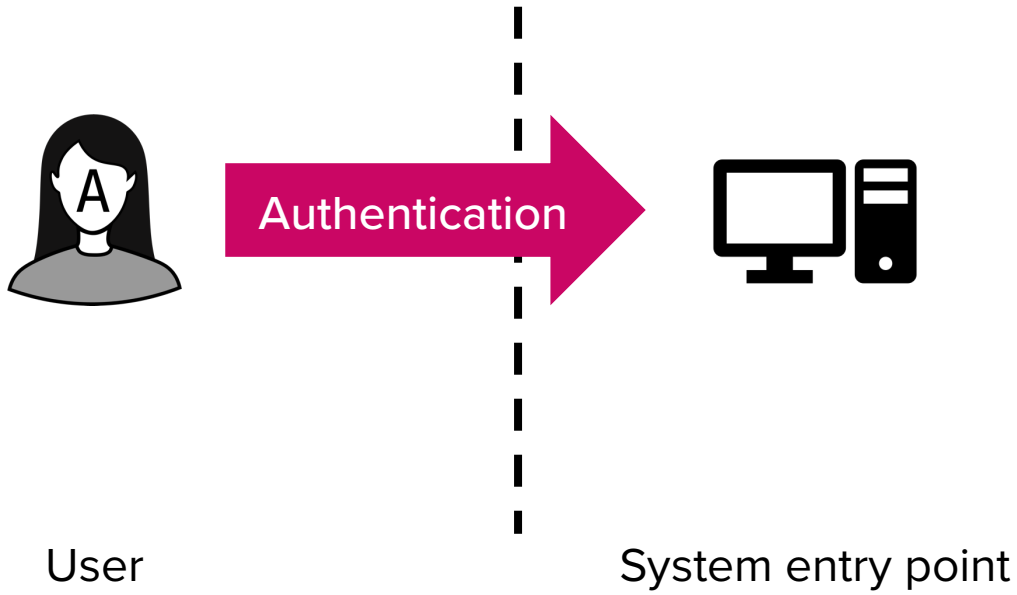
Spring 2026

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

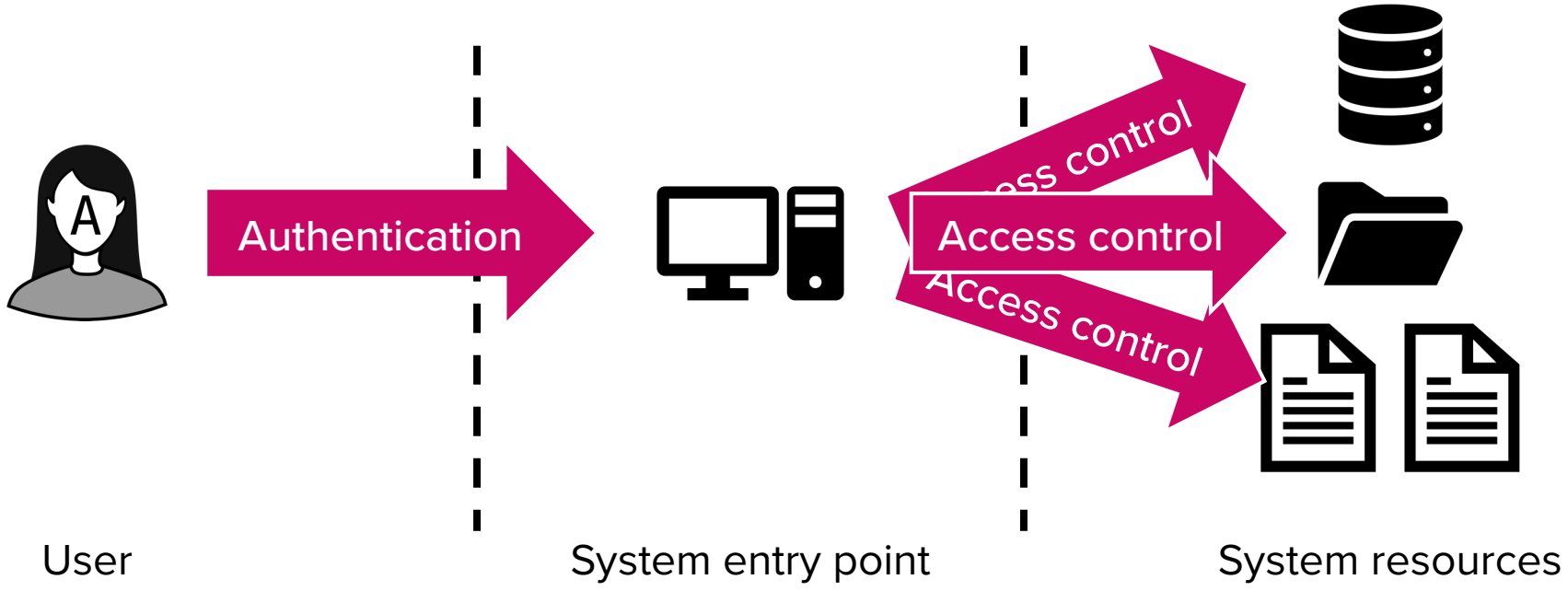
Recap

- User authentication
  - Coarse-grained gatekeeper for the entire system
    - Makes a binary decision: Grant or deny access



Today's topic: Access control

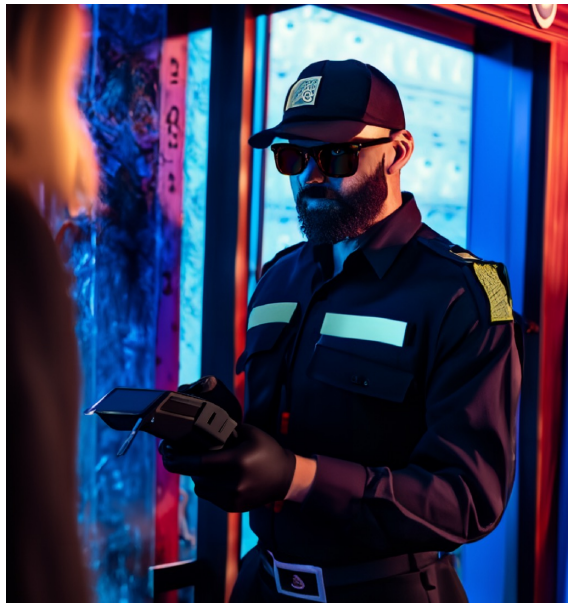
- Definition: Process of deciding whether to grant or deny a specific request to use an information or resource



A nightclub analogy

- Authentication:
  - ID check at the gate
- Access control:
  - Over 18: May enter the club
  - Over 21: May purchase/consume alcohol
  - On artist list: May access backstage and perform
  - On VIP list: May access the VIP lounge

  → Defines what an authenticated user is **allowed** to do



Models for access control

- Core entities
  - **Subject**: An entity that requests access
    - Owner: Creator of a resource
      - Note: Ownership is exclusive; a resource cannot be co-owned
    - Group: Named collection of users who can exercise access rights
    - Others: Users who are not the owner nor in group
  - **Object**: A resource to which access is controlled
    - Files, processes, memory, …

Models for access control

- Access rights: Defines what operation a subject may perform on an object
  - File access rights:
    - Read: View file contents
    - Write: Add, modify, or delete file contents
    - Execute: Run the file as a program
  - Directory access rights:
    - Read: List directory entries
    - Write: Create, delete, or rename directory entries
    - Execute: Use anything in or change working directory to the directory

Two access control policy families

- DAC: Discretionary Access Control
  - Controls access based on the identity of the requestor and per-resource rules
  - Owner of the resource determines access rights (hence discretionary)
- MAC: Mandatory Access Control
  - Controls access based on security labels and clearances enforced by the system
  - Users cannot override the system-wide policy (hence mandatory)

Discretionary Access Control (DAC)

Naïve approaches

- Naïve approach #1: Authentication == Access control
  - Allow access to all objects to all authenticated subjects
  - Drawback:
    - Only applicable to a single-subject, single-object setting
      - e.g., A physical safe
        - If you can open it (authentication),
          then you can access everything inside (access control)

Naïve approaches

- Naïve approach #2: Blacklists and whitelists
  - Blacklist: Allow by default, only deny blacklisted subjects
  - Whitelist: Deny by default, only allow whitelisted subjects
  - Drawbacks:
    - Only available for multiple-subject, single-object environments
      - e.g., Email spam filters
        - Everyone except those who are blacklisted (subjects) can send you (object) email
    - Both lists can grow quite large

→ How can we extend these naïve approaches
for modern systems with multiple subjects and objects?

Access control matrix (ACM)

- Allow multi-subject, multi-object access control
- Control:
  - access(subject, object) = 1 or 0
    - 1 (true): access granted
    - 0 (false): access denied

		Objects			
		A	**B**	**C**	**D**
Subjects	Alice	1	0	0	1
	Bob	0	1	1	1
	Claire	1	0	0	0
	Dave	0	1	1	1

Access control matrix (ACM)

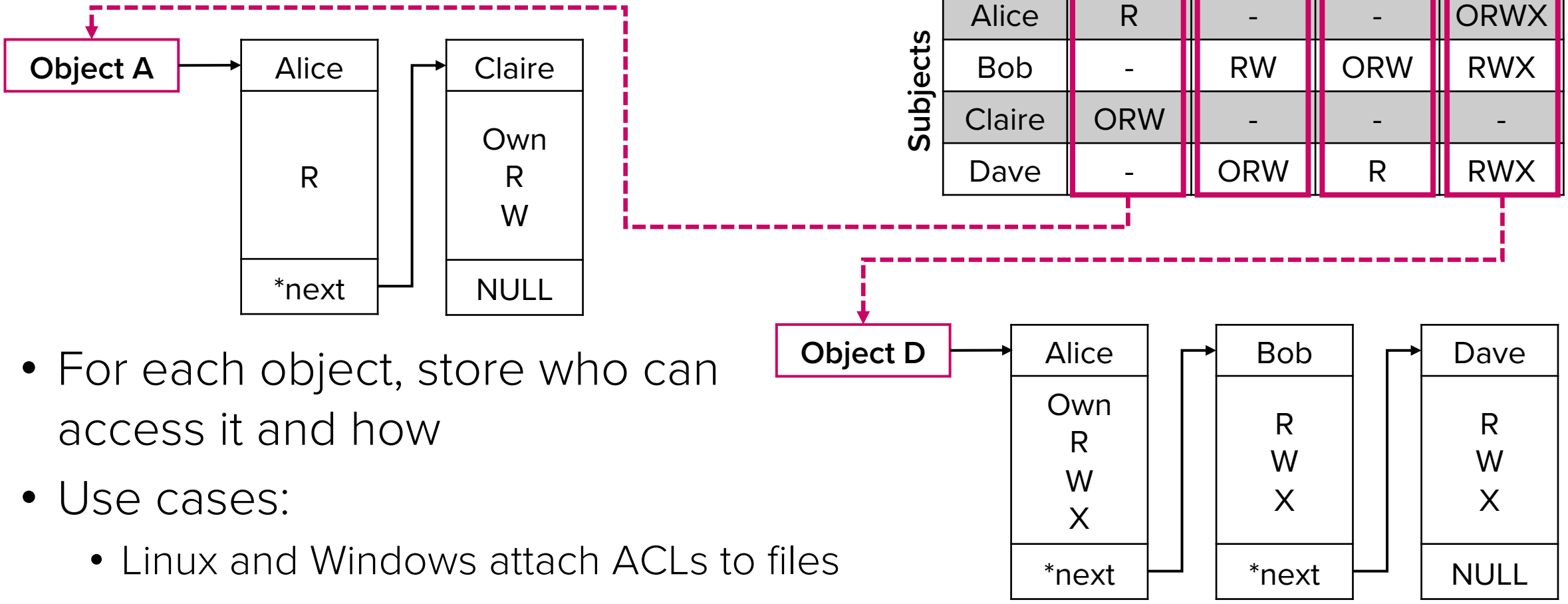
- ACM can be generalized to allow finer-grained access controls using access rights:
  - None (-), Own (O), Read (R), Write (W), Execute (X)
- Problems
  - ACM is a “sparse matrix” by nature
    - Incurs high storage overhead
  - Size of ACM grows significantly as the number of subjects and objects increases

Subjects \ Objects	A	B	C	D
Alice	R	-	-	ORWX
Bob	-	RW	ORW	RWX
Claire	ORW	-	-	-
Dave	-	ORW	R	RWX

- Solution: Store (subject, object) relations in a dense list!

Access control lists (ACL)

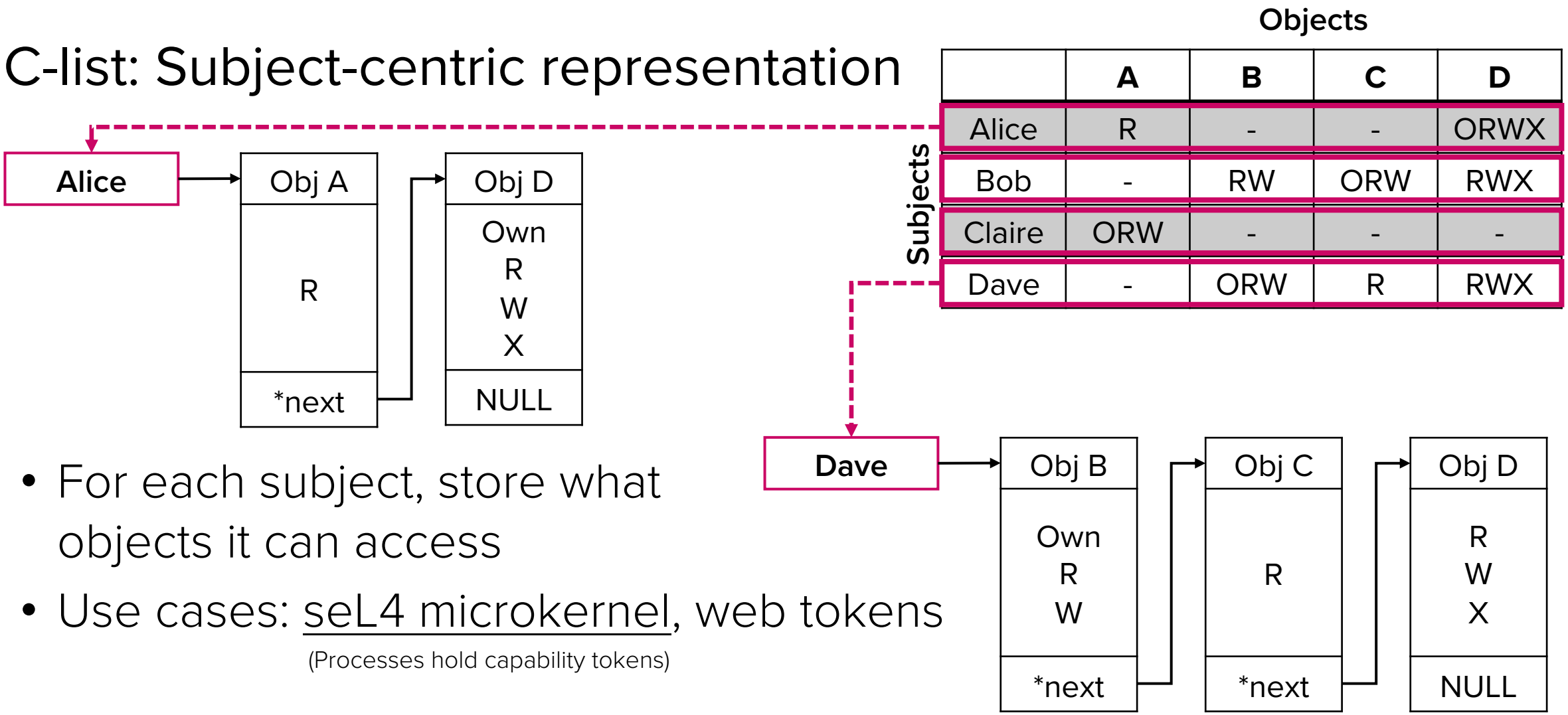
- ACL: Object-centric representation



  - For each object, store who can access it and how
  - Use cases:
    - Linux and Windows attach ACLs to files

Capability lists (C-list)

- C-list: Subject-centric representation

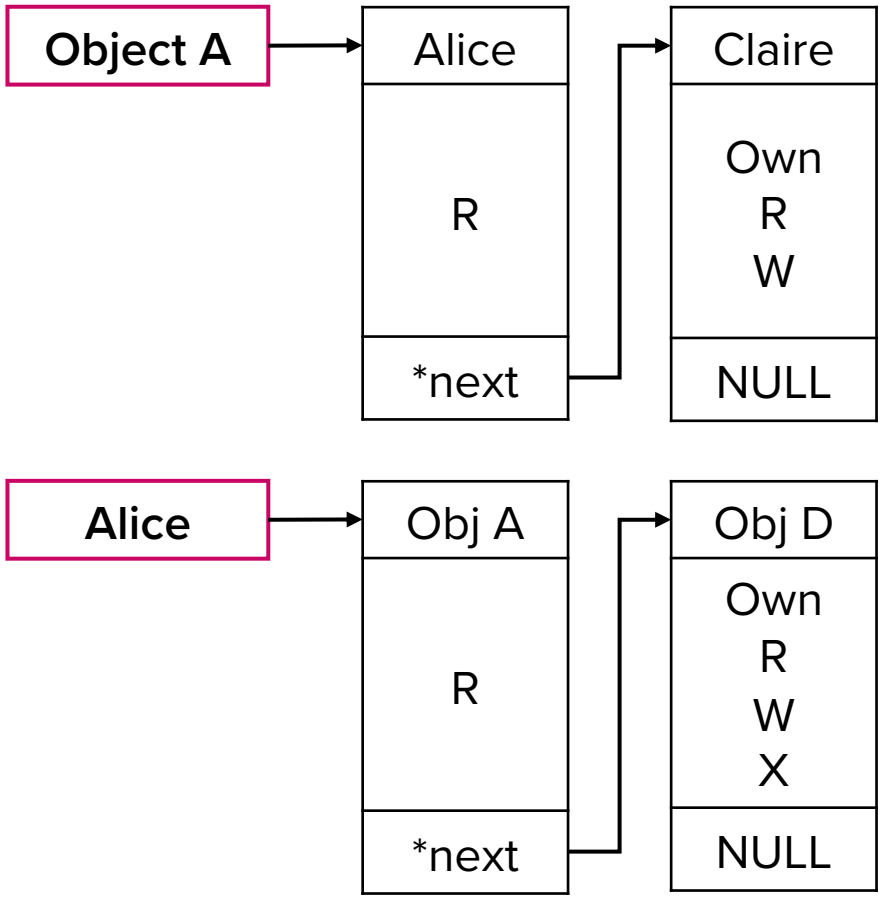


  - For each subject, store what objects it can access
  - Use cases: seL4 microkernel, web tokens

(Processes hold capability tokens)

ACL vs C-list

- Which one is better?
  1. Checking efficiency
     - ACL
       - Fast: Who can access Object A?
         - Single iteration
       - Slow: Which objects can Alice access?
         - Need to scan all ACLs
     - C-list
       - Fast: Which objects can Alice access?
         - Single iteration of Alice's C-list
       - Slow: Who can access Object A?
         - Need to search everyone's C-lists

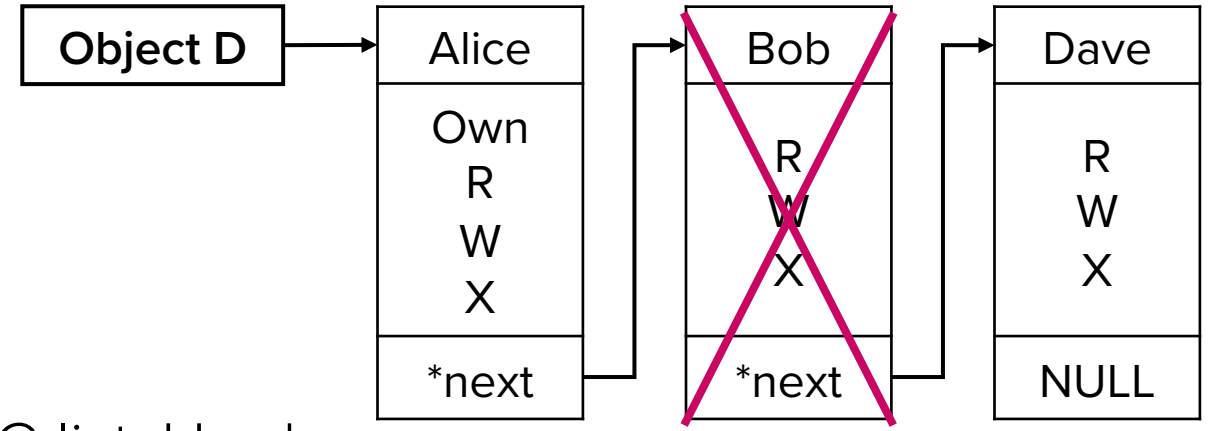


ACL vs C-list

- Which one is better?
  2. Delegation (Alice wants Bob to take her permissions)
     - ACL: Slow
       - Because ACL is slow at finding “Which objects can Alice access?”
     - C-list: Easy
       - A capability can be copied, delegated, and passed to other subjects

ACL vs C-list

- Which one is better?
  3. Revocation (removing a subject's access to an object)
     - ACL: Straightforward
       - Alice (owner) can remove Bob's permissions from the ACL of object D



     - C-list: Hard
       - Capabilities can be distributed (e.g., due to delegation). System needs to track all C-lists and remove the object

ACL vs C-list

- Which one is better?
  4. Accountability (e.g., a sensitive file has been accessed, and you want to find potential subject)
     - ACL: Easy
       - All information is available in one place, i.e., the ACL of the file
     - C-list: Hard
       - Need system-wide capability checks to investigate all subjects

Authority and delegation

- Access control using ACLs vs C-list
  - ACL-based systems rely on “who you are” at the time of access
    - ACLs: Objects decide who can access them
  - Capability systems rely on “what you have”
    - C-list: Subjects carry tokens of authority

What happens when a program acting on a user’s behalf has more privilege than the user invoking it?

Confused deputy problem

- Setting
  - Scenario: A pay-per-use compiler service
  - Command: `$ compiler input_filename output_filename`
  - The service charges users on each compiler invocation
  - The compiler records usage by updating a file named `billing`
    - The compiler has RW access to `billing`,
    - Alice (and other normal users) do not have access to `billing`
  - Alice wants to use the compiler
    - Alice has RX permissions for compiler

Subjects \ Objects	compiler	billing
Alice	RX	-
compiler	RX	RW

Access Control Matrix

Confused deputy problem

- Malicious behavior of Alice
  - Alice first uses the compiler normally:
    - Invokes it multiple times to compile programs
    - The compiler correctly updates the **`billing`** file
  - Then, Alice executes a malicious command:
    - **`$ compile input_filename billing`**
  - The compiler overwrites and corrupts **`billing`**
    - The compiler, a deputy acting on behalf of Alice, is confused!
    - Alice can walk away without paying anything

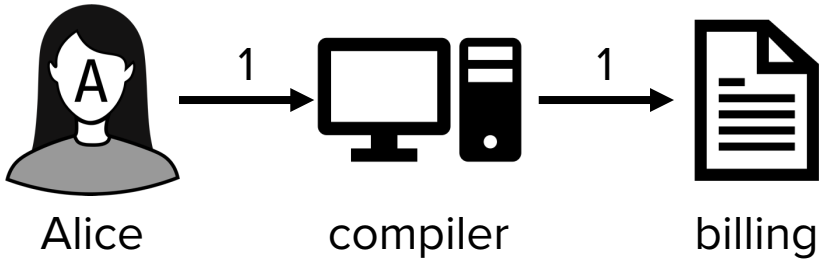
Objects

Subjects	compiler	billing
Alice	RX	-
compiler	RX	RW

Access Control Matrix

Confused deputy problem

- What's the issue with ACL?
  - access(Alice, compiler, execute) = 1
  - access(compiler, billing, write) = 1 → The compiler uses its own authority, not Alice's, when accessing files



is considered okay by ACL

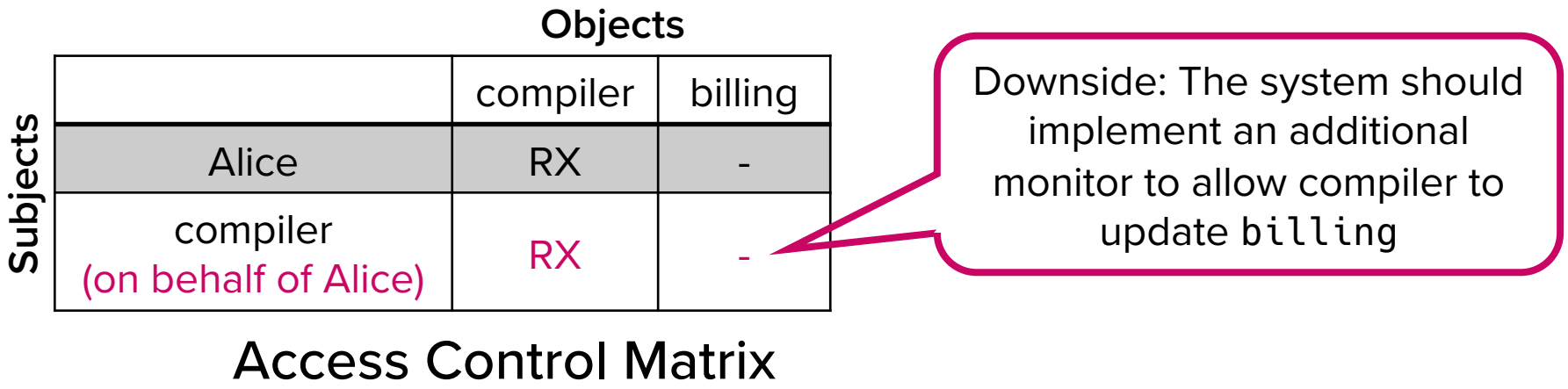
Objects

Subjects	compiler	billing
Alice	RX	-
compiler	RX	RW

Access Control Matrix

Confused deputy problem

- C-list can solve this problem through explicit delegation
  - Alice does not have a capability to write to the `billing` file
  - Alice must delegate her C-list to the compiler when executing it
  - The compiler, when acting on behalf of Alice:
    - Uses only the capabilities explicitly provided by Alice
    - Therefore, it cannot access or modify `billing`



Access Control Matrix

DAC in Practice

Unix-like systems use ACL

- Background
  - In Unix, every access-controlled resource is represented as a file
    - Regular files
    - Directories
    - Memory
    - Device drivers
    - Named pipes
    - Sockets
    - etc.

Unix-like systems use ACL

- Background
  - Each file has an <owner (UID), group (GID), others>
    - Owner is the primary controller, represented by UID
    - Group is a list of user accounts, represented by GID
    - Others is everyone else
    - User's details are in `/etc/passwd`
      - `csed415-lab04:x:2104:2104::/home/csed415-lab04:/bin/bash`

        username: csed415-lab04 | uid: 2104 | gid: 2104 | home directory: /home/csed415-lab04 | login command: /bin/bash
    - Group details are in `/etc/group`
      - `csed415-lab04:x:2104:`

        group name: csed415-lab04 | gid: 2104 | member list (empty)

Unix-like systems use ACL

- ACL permissions
  - Available permissions are read (`r`), write (`w`), and execute (`x`)
  - Original implementation: 9-bit representation
    - 3 bits for the owner, 3 bits for the group, 3 bits for the others
    - e.g., `rwxrw-r--` :
      - The owner can read, write, and execute
      - Members in the group can read and write
      - Everyone else can only read

Unix-like systems use ACL

- When applied on directories:
  - Read: List contents of directory
  - Write: Create or delete files in directory
  - Execute: Use anything in or change working directory to directory

```
mkdir /tmp/perm
cd /tmp/perm
mkdir kkk
stat kkk | grep Access
cd kkk
cd ..
chmod a-x kkk
stat kkk | grep Access
cd kkk
```

→ temporary directory for testing

→ shows (0775/drwxrwxr-x)

→ can cd (change directory) to kkk

→ remove x permission from all (user, group, others)

→ shows (0664/drw-rw-r--)

→ access denied

Unix-like systems use ACL

- Extended permissions: Available on modern Linux/macOS
  - SetUID: If set, program runs as the owner no matter who executes it
  - SetGID: If set, program runs as a member of the group
    - "Runs as" == Runs with the same privileges as
  - Examples:
    - Lab target binaries

```
$ stat /home/csed415-lab03/target | grep Access
Access: (4750/-rwsr-x---)  Uid: (21003/lab03-solved)  Gid: (2103/csed415-lab03)
```

SetUID

    - sudo

```
$ stat /usr/bin/sudo | grep Access
Access: (4755/-rwsr-xr-x)  Uid: (    0/    root)   Gid: (    0/    root)
```

SetUID

Unix-like systems use ACL

- Extended permissions: Available on modern Linux/macOS
  - Sticky bit
    - Originally: To keep files in memory (obsolete)
    - Now: Used on directories to restrict deletion and renaming
    - If sticky bit is set on a directory:
      - Only file owner, directory owner, or root can delete or rename files
      - Even with directory write permission, others cannot delete arbitrary files
    - Example

```
cd /tmp/perm
mkdir mmm
chmod +t mmm
stat mmm | grep Access
```

→ temporary directory for testing

→ shows (`1775/drwxrwxr-t`)

Unix-like systems use ACL

- Representing permissions
  - Numeric representation of permission bits consists of four digits
    - Last three digits: Permissions for user, group, and others

Bit position	2	1	0
Permission	Read	Write	Execute

r: $2^2 = 4$ w: $2^1 = 2$ x: $2^0 = 1$

→ rwx: read + write + execute = 4 + 2 + 1 = 7
→ rw: read + write = 4 + 2 = 6

    - First digit: Special permissions

Bit position	2	1	0
Permission	setuid	setgid	sticky bit

setuid: $2^2 = 4$ setgid: $2^1 = 2$

Represent full permission with 4 digits:
special owner group others

Q) what does 4750 mean?

Q) what does 3000 mean?

Unix-like systems use ACL

- When does ACL check happen?
  - Creating
    - `creat(filename, mode);`
    - `open(filename, flags, mode);` // specify `O_CREAT` in flags to create file
  - Opening
    - `int fd = open(filename, flags);`
      - flags: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`
      - OS returns a file descriptor (`fd`) if the file exists
    - ACL check happens at this stage!
      - System traverses the file's ACL and checks whether the permission in the open flags match the subject's access rights
    - Afterwards, the file can be accessed through the file descriptor (`fd`)

Unix-like systems use ACL

- When does ACL check happen?
  - Reading
    - `read(fd, buf, count);`
      - read **`count`** bytes and store in **`buf`** from the open file referred to by the **`fd`**
  - Writing
    - `write(fd, buf, count);`
      - write **`count`** bytes from **`buf`** to the open file referred to by the **`fd`**
  - Closing
    - `close(fd);`
      - Closes a file descriptor (invalidates the reference)

Reading and writing does not require any permission check → performance!

Unix-like systems use ACL

- Interacting with files in Unix-like systems through syscalls
  - Example: **`open( )`**'s permission check

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(void) {
  int fd = open("myfile", O_RDWR);
  printf("fd: %d\n", fd);

  char* buf = strerror(errno);
  printf("Error: %s\n", buf);

  return 0;
}
```

→ Test with varying permissions of `myfile`

Real-World Attacks on DAC

Attack #1: Symbolic link attacks

- Scenario
  - Utilize a world-writable directory, e.g., **`/tmp`**
  - Exploit privileged programs that write to predictable filename
    - e.g., a program writes to **`/tmp/output.log`**
- Attack `$ ln -s /etc/passwd /tmp/output.log`
  - When the program writes to **`/tmp/output.log`**,
    it actually overwrites **`/etc/passwd`** (which is only writeable by root)

```
➜ ls -alh /etc/passwd
-rw-r--r-- 1 root root 2.3K Apr 24 18:54 /etc/passwd
```

Attack #2: Race condition

- Time-of-check to time-of-use (TOCTOU)
  - A race condition where the resource checked is not the same as the resource used, due to changes in between
  - A program could check a condition (e.g., permission, file path) then later use the resource
  - If an attacker can modify the resource between check and use:
- Vulnerable pattern:

```
if (access("myfile", W_OK) == 0) {
  fd = open("myfile", O_WRONLY);
  write(fd, buf, 32);
}
```

Attack:

Race to replace `myfile` between check and use

```
$ ln -s /etc/passwd myfile
```

Summary

- Access control determines whether a subject's request to access an object should be granted
- ACLs and C-lists are two ways to represent the authorization state in discretionary access control (DAC)
- Unix-like systems use ACL-style, object-centric permissions
  - owner / group / others and RWX

Coming up next

- Problem with DAC: Information flow control problem
  - Once Alice shares data with Bob, she loses control over further propagation
  - Bob can copy, redistribute, or leak data
  - DAC only controls access, but not how information flows after access is granted

  → Motivation for Mandatory Access Control (MAC)

Questions?