Lec 17: Access Control

CSED415: Computer Security

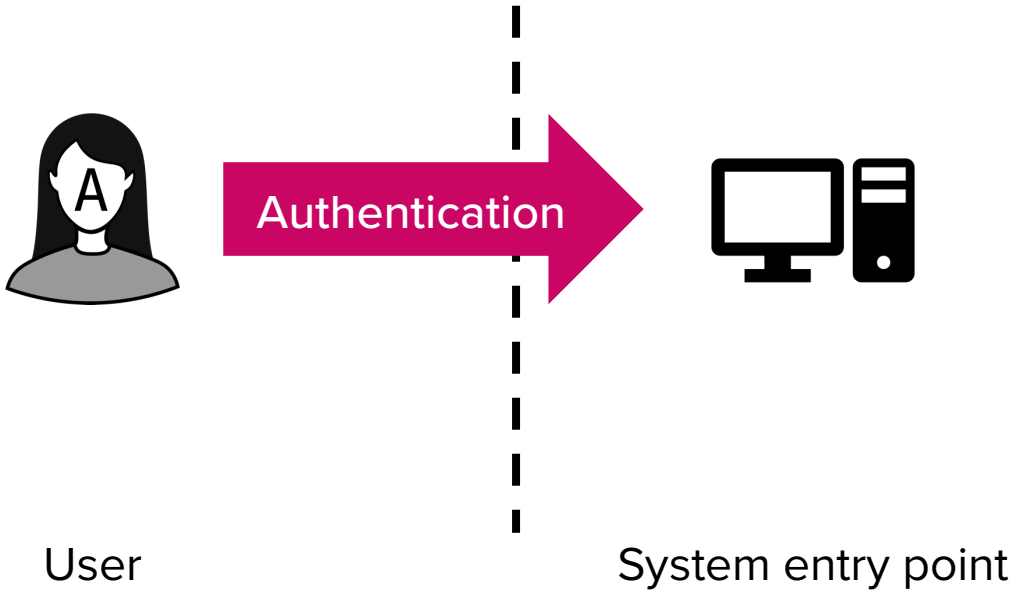
Spring 2025

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

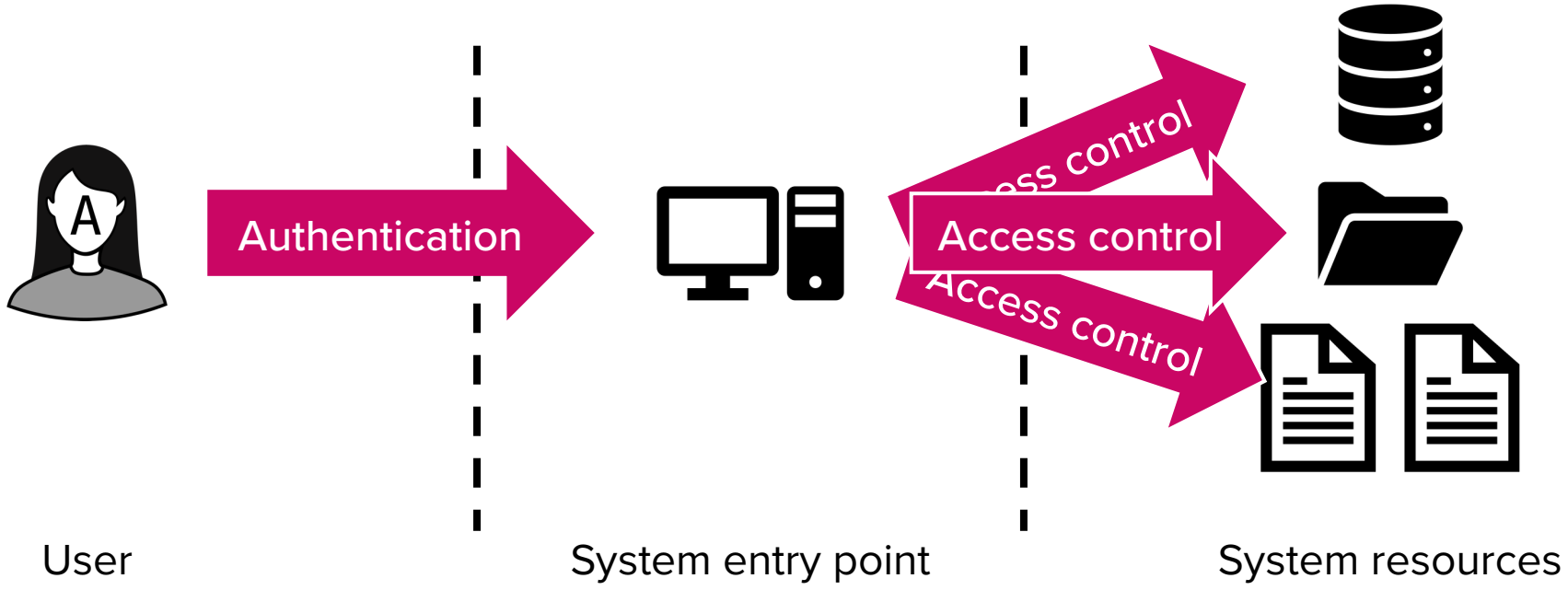
Recap

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



Today's topic: Access control

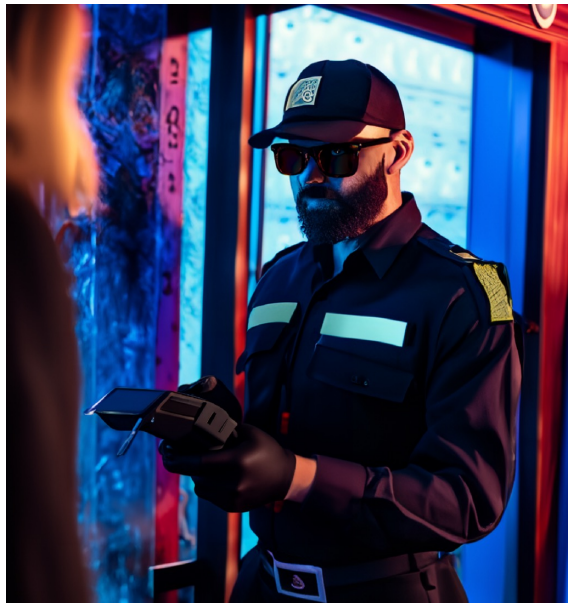
- Definition: Process of deciding whether to grant or deny a specific request to use an information or resource



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: Describe how a subject may access an object
  - File access rights:
    - Read: View data
    - Write: Add, modify, or delete data
    - Execute: Run as a program
  - Directory access rights:
    - Read: List directory entries
    - Write: Create or delete files
    - Execute: Enter inside

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 policy (hence mandatory)

Discretionary Access Control (DAC)

Elementary approaches

- 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)

Elementary approaches

- 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 elementary 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

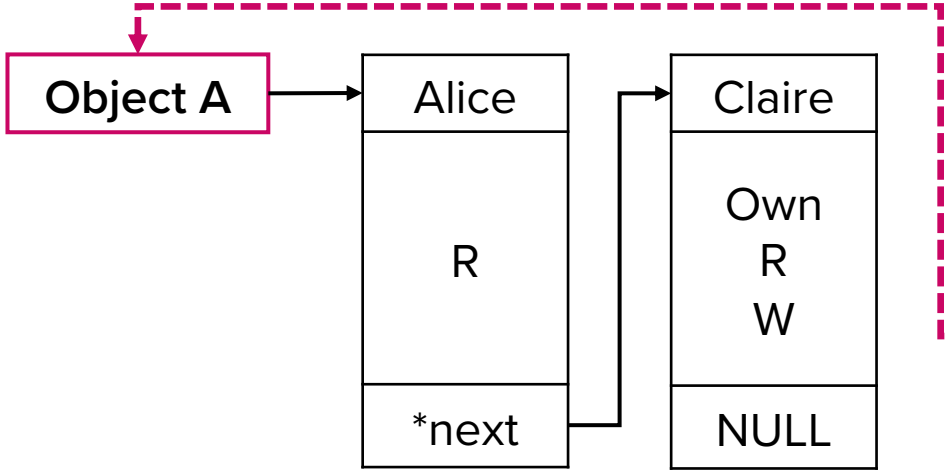
Access control matrix (ACM)

- 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

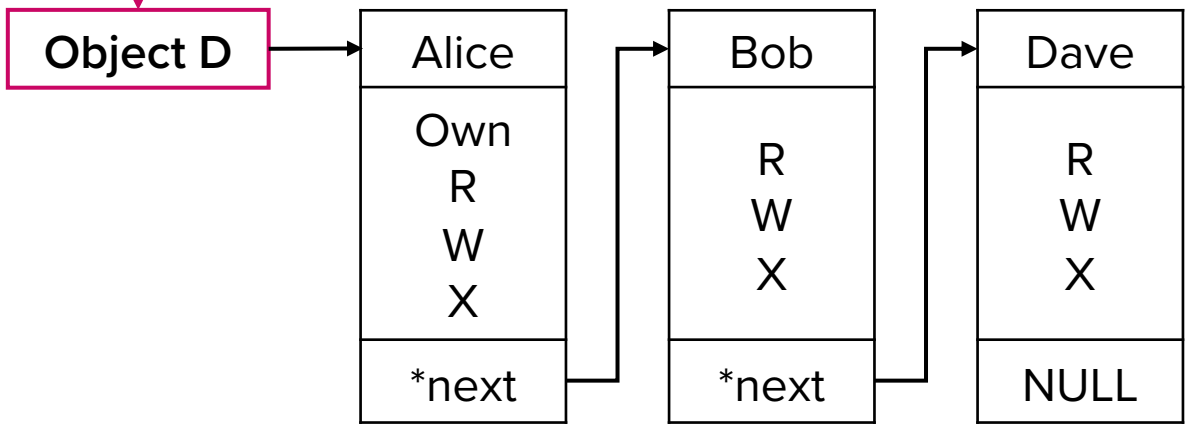
Subjects \ Objects	A	B	C	D
Alice	R	-	-	ORWX
Bob	-	RW	ORW	RWX
Claire	ORW	-	-	-
Dave	-	ORW	R	RWX

Access control lists (ACL)

- Slice ACM by columns (objects)



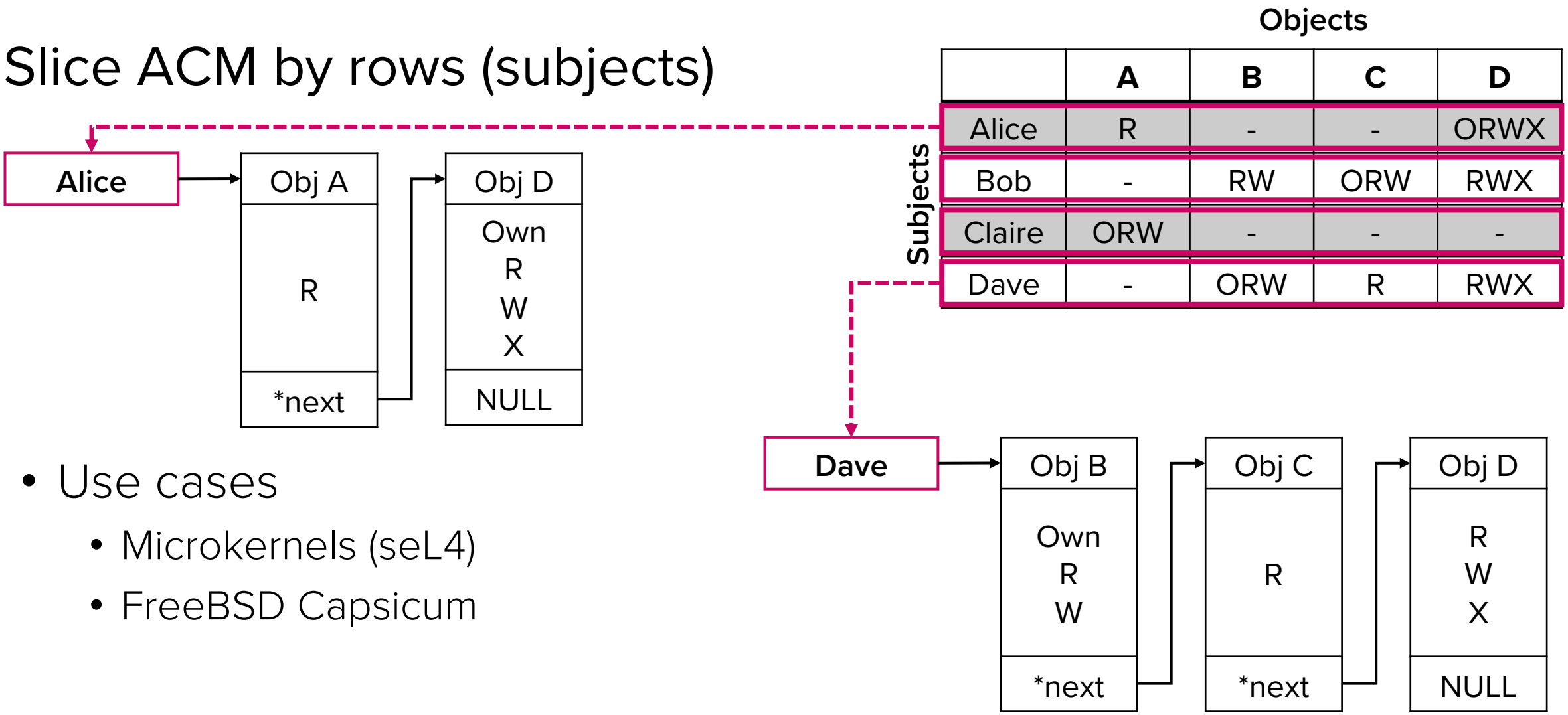
	Objects			
Subjects	A	B	C	D
Alice	R	-	-	ORWX
Bob	-	RW	ORW	RWX
Claire	ORW	-	-	-
Dave	-	ORW	R	RWX



  - Use cases
    - Traditional OS filesystems (Linux, Windows)

Capability lists (C-list)

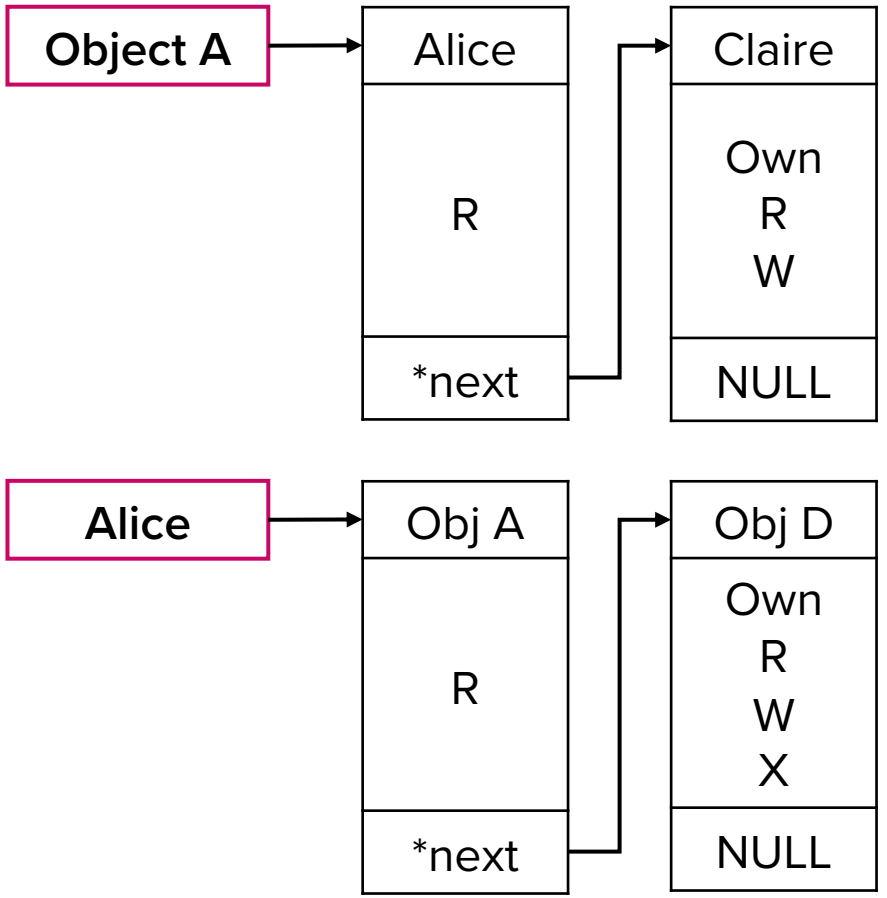
- Slice ACM by rows (subjects)



  - Use cases
    - Microkernels (seL4)
    - FreeBSD Capsicum

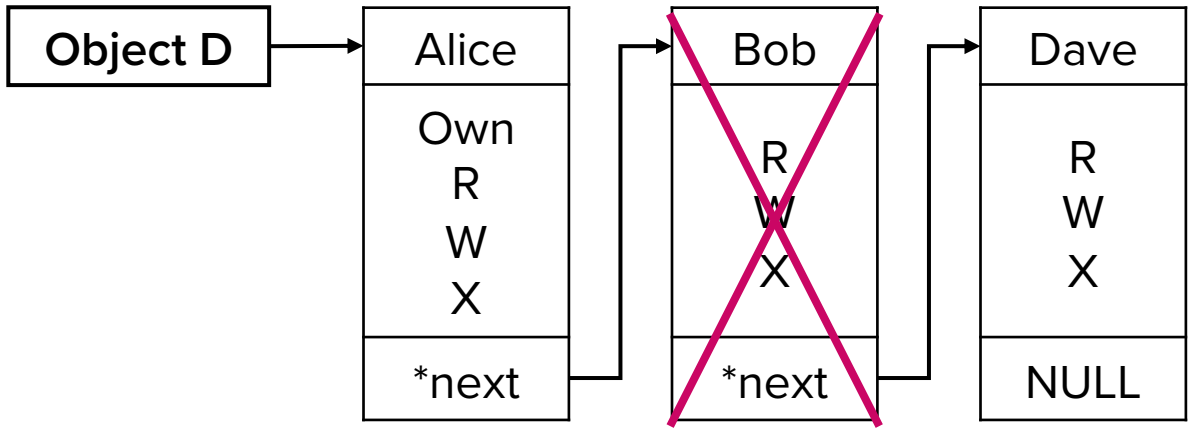
ACL vs C-list

- 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. 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
       - Alice (owner) cannot control Bob's C-list. System needs to intervene

ACL vs C-list

- Which one is better?
  3. 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

ACL vs C-list

- Which one is better?
  4. Delegation (Alice wants Bob to take her permissions)
     - ACL: Slow
       - Because ACL is slow at finding “Which objects can Alice access?”
     - C-list: Easy
       - Alice’s C-list can be passed onto Bob

Overall, ACL offers “natural” solutions to daily access control situations (accessing objects, revoking access, forensics, …)

But are there any problems that ACL cannot handle?

Confused deputy problem

- Setting
  - Imagine a pay-per-use compiler
    - Command: `$ compiler input_filename output_filename`
  - System wants to charge users when they use the compiler
  - The compiler updates a billing file after it is executed
    - The compiler has RW permissions for the billing file, but Alice doesn't
  - 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

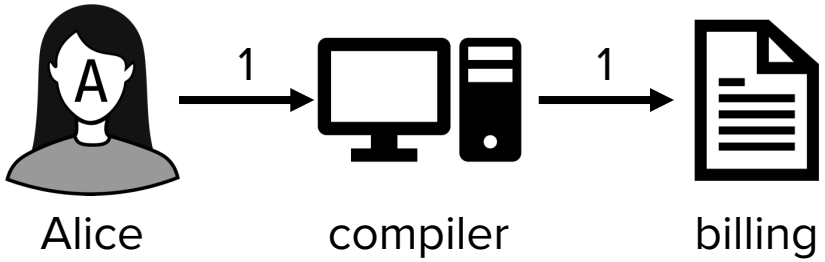
- Malicious behavior of Alice
  - Alice executes the compiler several times to compile programs
    - The compiler updates the billing file with Alice's records
  - Then, Alice executes:
    - `$ compile input_filename billing`
  - The billing file gets corrupted
    - The compiler, a deputy acting on behalf of Alice, is confused!
    - Alice can walk away without paying anything

Subjects \ Objects	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



is considered okay by ACL!

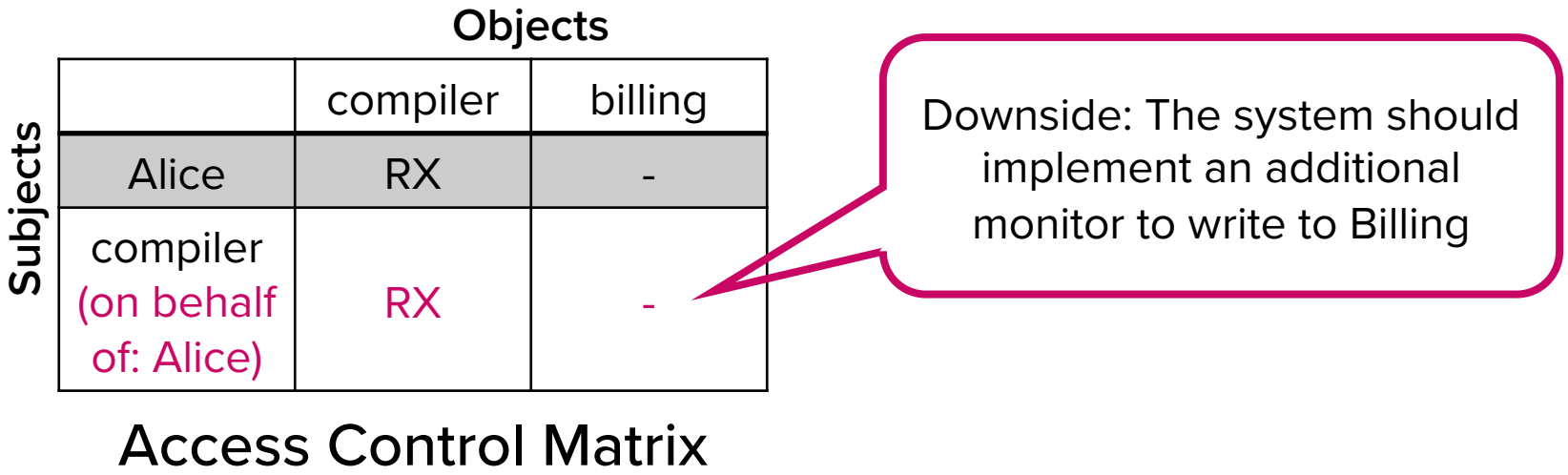
Subjects \ Objects	compiler	billing
Alice	RX	-
compiler	RX	RW

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, if running on behalf of Alice, cannot edit the billing file

  → Free from the confused deputy problem



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

        group name | gid | 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 used to lock file in memory (sticky!)
    - Now used on directories to limit delete operation
      - If sticky bit is set, requester must own file or directory to delete
      - Other users cannot delete even with write permission
    - 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
    - User, group, others permissions (Last three digits):

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

    - Special permissions (First digit):

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 involve 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`

Attacking Access Control

TOCTOU vulnerability

- Time-of-check to time-of-use (TOCTOU)
  - Access permission checking is performed when a file is opened
  - Once checked, the permission remains available until the file is explicitly closed (or the process terminates and implicitly closed)
    - `read()` or `write()` use the granted descriptor without re-checking
  - What if an attacker alter the permissions between check and use?
    → Time-of-check to time-of-use vulnerability

TOCTOU vulnerability

- Time-of-check to time-of-use (TOCTOU)
  - Example: `vi` (predecessor of `vim`)
    - `vi` keeps a backup of the original file upon save
      - e.g., Original filename: `file`
      - Save renames `file` to `file.bak`, creates a new file named `file`, and writes contents
    - Tricky situation: Handling ownership
      - e.g., If we run `vi` as root, modify `alice_file` that Alice owns, and save,
        - `rename("alice_file", "alice_file.bak");` retains the permissions
        - `open("alice_file", O_CREAT);` → owned by root, since `vi` is running as root
        - `vi` needs run `chown("alice_file", alice_uid, alice_gid);` to change the owner of `alice_file` to Alice

TOCTOU vulnerability

- Time-of-check to time-of-use (TOCTOU)

vi (Run as root)

```
rename("alice_file", "alice_file.bak");
```

```
open("alice_file", O_CREAT);   // TOC: root has permissions to create alice_file
```

Attacker runs:

```
$ ln -s  /etc/shadow  alice_file
```

```
chown("alice_file", alice_uid, alice_gid); // TOU: alice_file is no longer the created file
```

Result: /etc/shadow is owned by Alice

Summary

- Access control allows us to determine if a request from a subject to access an object can be granted
- ACLs and C-lists are two ways to represent states used for discretionary access control decisions
- Unix-like systems use ACL for access controls

Coming up next

- Information flow control problem
  - With DAC, Alice can never be sure that sensitive data she shares with Bob will not be further shared with others

  → Motivation for Mandatory Access Control (MAC)

Questions?
(Including Midterm exam and Lab 04)