Lec 19: Access Control

CSED415: Computer Security

Spring 2024

Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Lab 04 is due this weekend!
  - Questions?

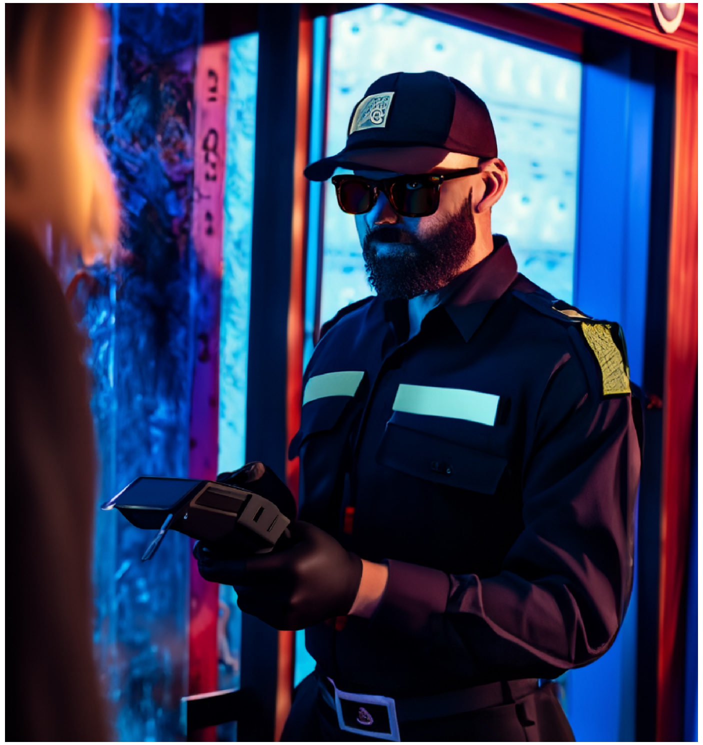
Recap

- User authentication
  - Enforces coarse-grained control for the entire system
  - Makes a binary decision: Grant or deny access

A nightclub example

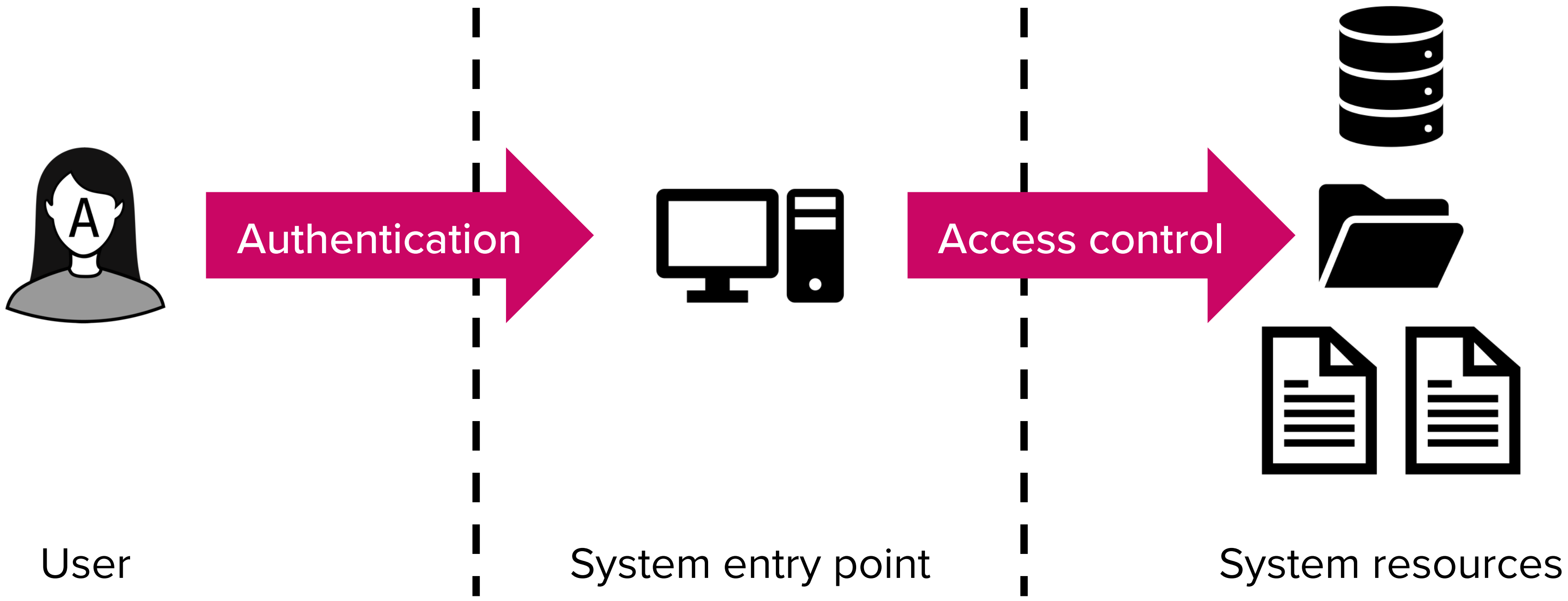
- Authentication
  - ID check at the gate
- Access control
  - Over 18 – only allowed in
  - Over 21 – allowed to buy and drink alcohol
  - On artist list – allowed to enter backstage and perform on stage
  - On VIP list – allowed to access VIP area

  → Focus: What you are **allowed** to do



Access control

- Definition: Process of granting or denying specific requests to obtain and use information or resources



Models for access control

- Entities
  - **Subject**: An entity capable of accessing objects
    - Owner: Creator of a resource
      - Ownership is exclusive; a resource cannot be co-owned
    - Group: Named group of users who can exercise access rights
    - World (or others): Users who are not the owner nor group for a resource
  - **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 information in a file
    - Write: Add, modify, or delete data in a file (includes read access)
    - Execute: Execute specified file
  - Directory access rights:
    - Delete: Delete files in a directory
    - Create: Create new file in a directory
    - Search: List files in a directory

Models for access control

- Two types of policy: DAC and MAC
  - DAC: Discretionary Access Control
    - Controls access based on the identity of the requestor and associated rules
    - Owner of the file determines access rights (hence “discretionary”)
  - MAC: Mandatory Access Control
    - Controls access based on security labels and clearances
    - System determines access rights (hence “mandatory”)

Discretionary Access Control (DAC)

Elementary forms of access control

- Authentication == Access control
  - Only available for single-subject, single-object environment
    - e.g., a safe
  - Allow access to authenticated subjects

Elementary forms of access control

- Blacklists and whitelists
  - Only available for multiple-subject, single-object environments
    - e.g., Email spam filter uses a blacklist
  - Blacklist: Allow by default, deny blacklisted subjects
    - Hard to reason about who can access resource
  - Whitelist: Deny by default, allow whitelisted subjects
    - Hard to deal with adding whitelist entries
  - Both lists can grow quite large

→ How to extend for modern systems with multiple subjects and objects?

Access control matrix (ACM)

- Allow multi-subject multi-object access 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)

- Finer-grained control is available with access rights
  - None, Own, Read, Write, Execute
- Problems
  - ACM is a “sparse matrix” by nature
    - High storage overhead
  - Size of ACM grows significantly as the number of subjects and objects increases

Objects

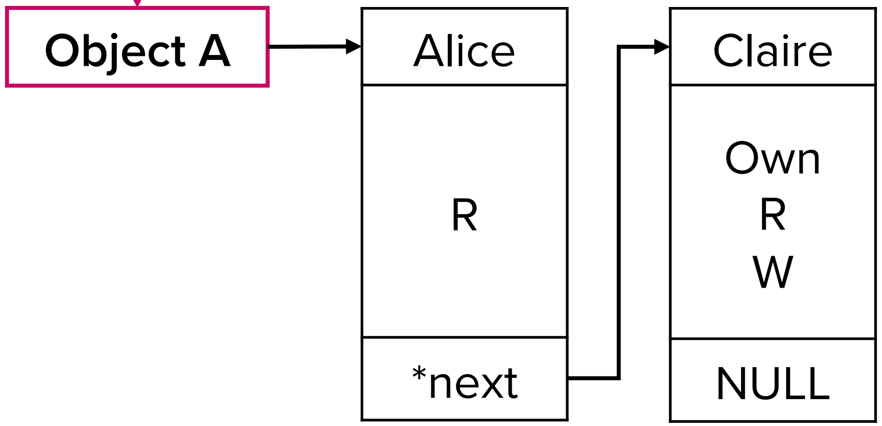
Subjects	A	B	C	D
Alice	R	-	-	ORWX
Bob	-	RW	ORW	RWX
Claire	ORW	-	-	-
Dave	-	ORW	R	RWX

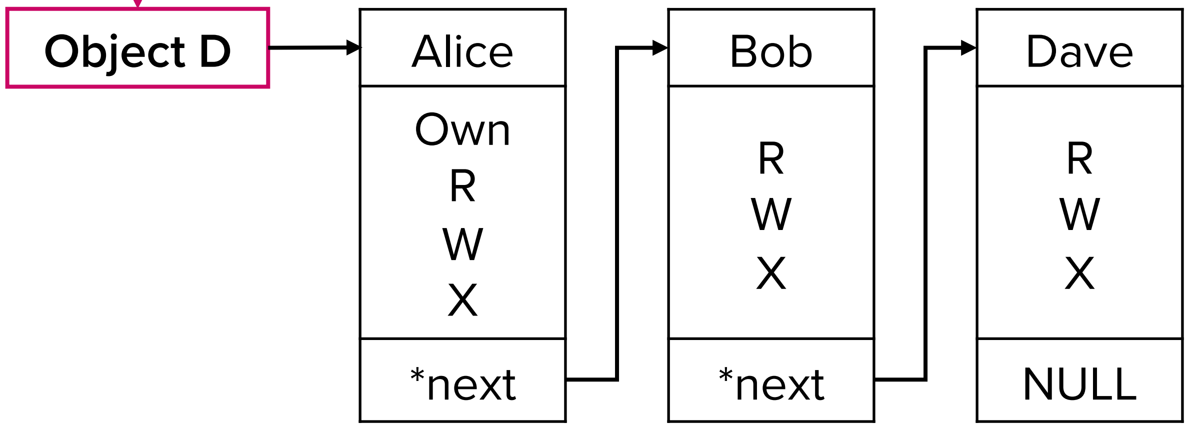
Access control lists (ACL)

- Slice ACM by columns (objects)

Objects

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





\+ Convenient to determine “who can access this resource?”

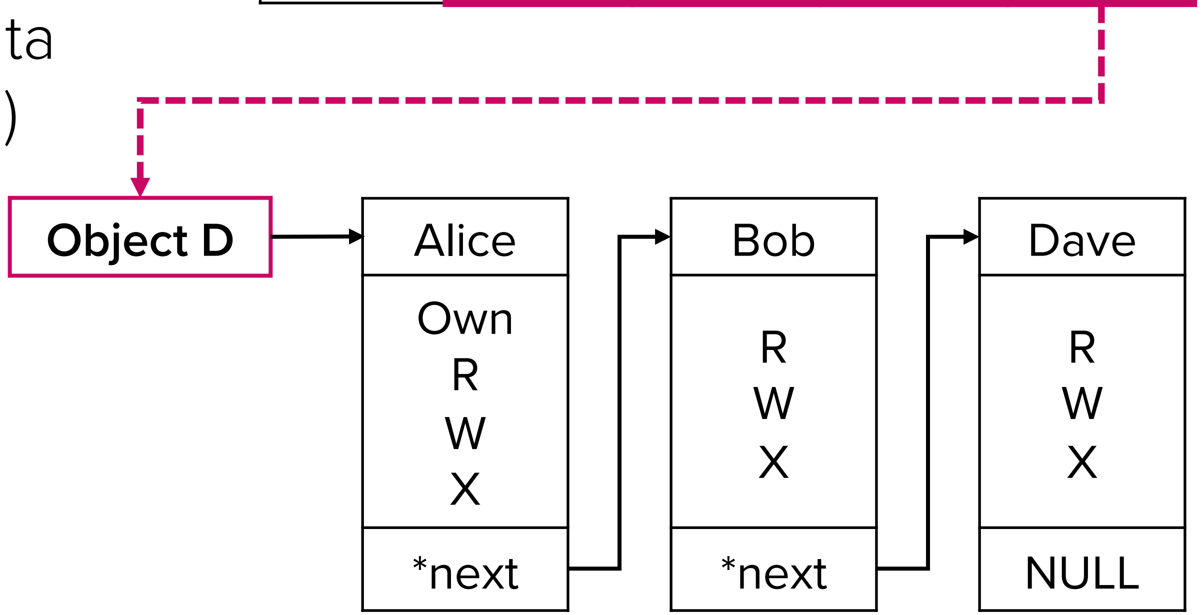
\- Very inefficient to determine the objects that a specific subject can access

Access control lists (ACL)

- Where should an ACL be stored?
  - In trusted part of the system (e.g., OS)
  - Storing the ACL with object's other metadata would be natural
    - A file object has associated metadata (size, created time, modified time, …)
    - ACL can also be stored as a metadata

Objects

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

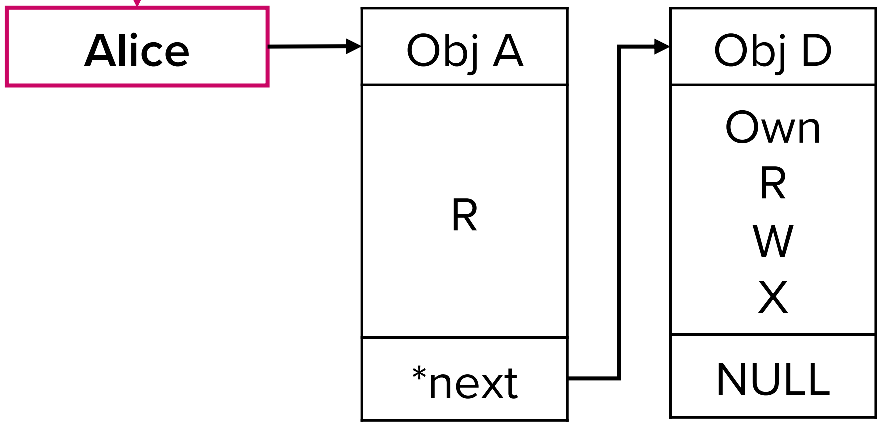


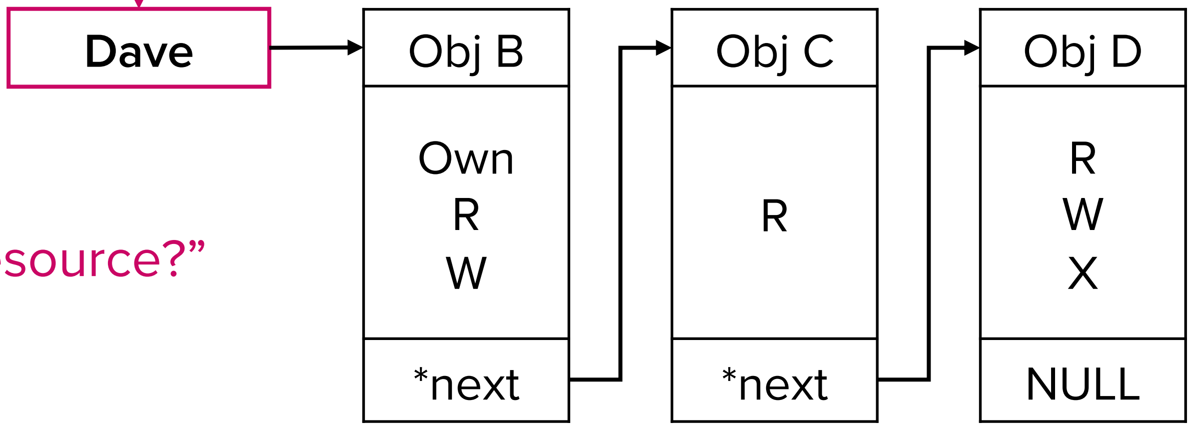
Capability lists (C-list)

- Slice ACM by rows (subjects)

Objects

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





+ Good for checking what a user can do

+ Provides flexibility for delegation

- Challenging to determine “who can access this resource?”

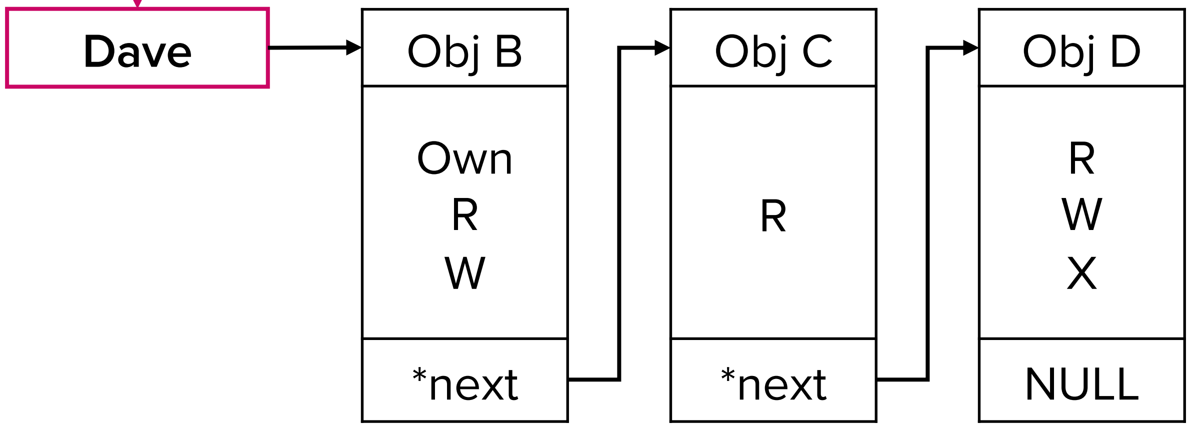
- Revocation is tricky

Capability lists (C-list)

- Where should a C-list be stored?
  - A “capability” is an unforgeable reference/handle for a resource
  - OS should maintain C-lists of all subjects (users)
  - Object sharing requires propagation of capabilities

Objects

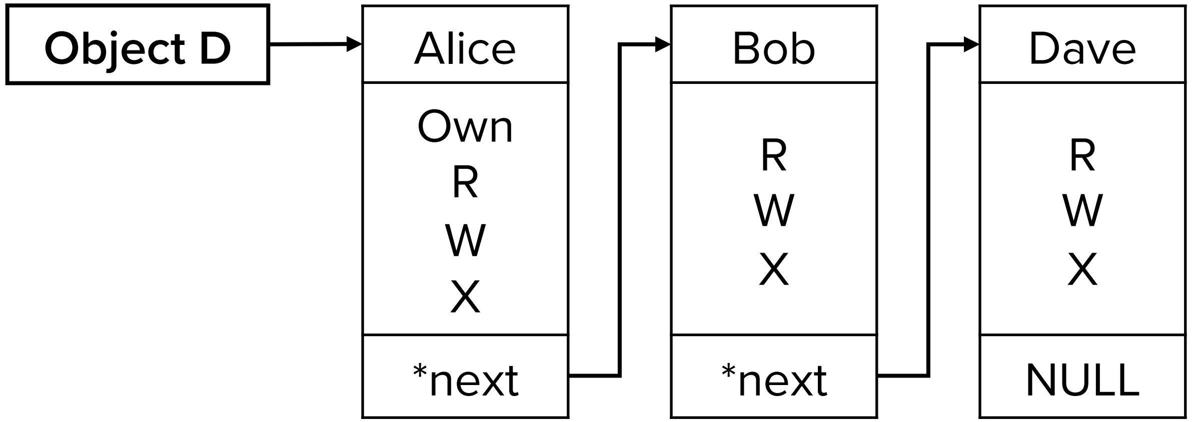
Subjects	A	B	C	D
Alice	R	-	-	ORWX
Bob	-	RW	ORW	RWX
Claire	ORW	-	-	-
Dave	-	ORW	R	RWX



ACL vs C-list

POSTECH

- Which one is better?
  1. Checking efficiency (e.g., subject tries to access an object)
     - ACL – System needs to traverse object's ACL to find the user ($O(n)$)



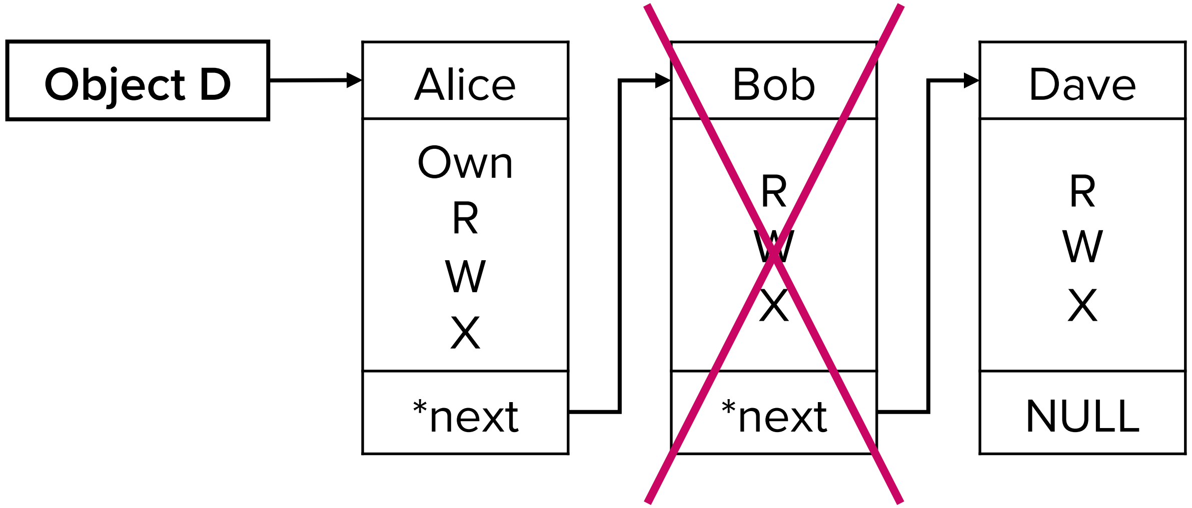
If Eve requests to read object D, time complexity is O(n) where n is the length of object D's ACL

     - C-list – Subject can present its capability ticket for the object ($O(1)$)

→ C-list wins

ACL vs C-list

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



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

→ ACL wins

ACL vs C-list

POSTECH

- Which one is better?
  3. Accountability (e.g., a sensitive file has been accessed and you want to find potential subject)
     - ACL – All information is available in one place, i.e., the ACL of the file
     - C-list – All subjects need to be investigated for their capabilities

     → ACL wins

Overall, ACL seems to outperform C-list

But are there any problems that ACL cannot handle?

Confused deputy problem

- Setting
  - Compiler is a pay-per-use service
    `$ 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
  - Alice wants to use the Compiler

Objects

Subjects	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
    - Compiler updates Billing file with Alice's records
  - Then, Alice executes:
    `$ compile input_filename Billing`
  - Billing file gets corrupted
    - Compiler, a deputy acting on behalf of Alice, is confused!

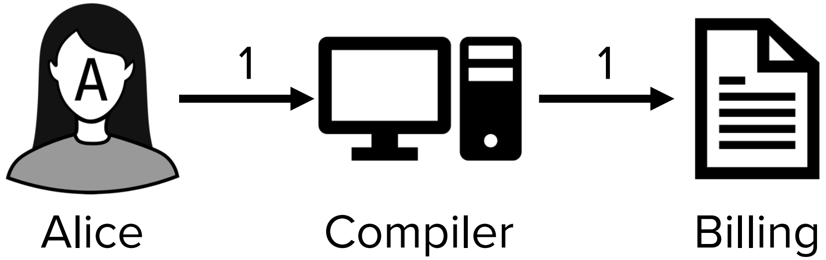
Objects

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



is considered okay by ACL!

Objects

	Compiler	Billing
Alice	RX	-
Compiler	RX	RW

Subjects

Access Control Matrix

Confused deputy problem

- C-list can solve this problem through delegation
  - Alice does not have a capability to write to Billing
  - Alice must delegate her C-list to the Compiler when executing it
  - The Compiler cannot write to Billing
  - → Free from the confused deputy problem

Objects

Subjects	Compiler	Billing
Alice	RX	-
Compiler (on behalf of: Alice)	RX	-

Access Control Matrix

Downside: The system should implement an additional monitor to write to Billing

DAC in Practice

Unix-like systems use ACL

- Background
  - In Unix, every access-controlled resource is represented as a file
    - Memory
    - Device drivers
    - Named pipes
    - …

Unix-like systems use ACL

POSTECH

- Background
  - Each file has an owner (UID), group (GID), and everyone (world)
    - User is someone capable of using files
    - Group is a list of user accounts
    - One user may belong to many groups
    - User's details are in `/etc/passwd`
      - `lab02:x:1012:1012::/home/lab02:/bin/bash`

        username uid gid home directory login command
    - Group details are in `/etc/group`
      - `lab02:x:1012:target,lab02,target02`

        group name gid member list

Unix-like systems use ACL

- File permissions
  - Available file permissions are read (r), write (w), and execute (x)
  - Original ACL implementation had a 9-bit representation
    - 3 bits for the owner, 3 bits for the group, 3 bits for everyone else
    - e.g., `rwxrw-r--` : Owner can rwx, group members can rw, and others can r

Unix-like systems use ACL

POSTECH

- Directory permissions work differently
  - Read: List contents of directory
  - Write: Create or delete files in directory
  - Execute: Use anything in or change working directory to directory

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

→ temporary directory for testing

→ shows (0775/drwxrwxr-x)

→ can cd (change directory) to kkk

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

→ access denied

Unix-like systems use ACL

- ACL implementation
  - Original ACL implementation had a 9-bit representation
    - 3 bits for the owner, 3 bits for the group, 3 bits for everyone else
    - e.g., `rwxrw-r--` : Owner can rwx, group members can rw, and others can r
  - Modern OSes support full ACL (Linux, BSD, MacOS, …)
    - 3 additional bits: setuid, setgid, and sticky bit

Unix-like systems use ACL

- ACL implementation
  - Numeric representation of permission bits consists of four digits
    - User, group, others permissions:

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:

Bit position	2	1	0
Permission	setuid	setgid	sticky bit

→ setuid: $2^2 = 4$

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

POSTECH

- Extended permissions
  - Sticky bit
    - Originally used to lock file in memory (sticky!)
    - Now used on directories to limit delete operation
      - If sticky bit is set, must own file or directory to delete
      - Other users cannot delete even with write permission
    - Example

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

→ temporary directory for testing

→ shows (`1775/drwxrwxr-t`)

Q) Where can you find this permission?

Unix-like systems use ACL

- Extended permissions
  - SetUID: If set, program runs as the owner no matter who executes it
  - SetGID: If set, program runs as the member of the group
  - “Runs as” == Runs with the same privileges as
  - Examples:
    - Lab target binaries

```
$ stat /home/lab02/target | grep Access
Access: (4750/-rwsr-x---)  Uid: ( 1999/  target)   Gid: ( 1012/   lab02)
```

    - sudo

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

Unix-like systems use ACL

- Extended permissions
  - SetUID/SetGID binaries are great targets for attackers seeking privilege escalation
    - e.g., CVE-2012-0809 in sudo binary
      - sudo had a debug mode (-D), which invokes sudo_debug function
      - The function printfs the program name (i.e., sudo)
      - The program name can be controlled by an attacker
        - e.g., by creating a symbolic link: `ln -s /usr/bin/sudo arbitrary_name`
        - "arbitrary_name" can be a format string payload, which leads to arbitrary code execution (format string vulnerabilities are not covered in this course)
      - Huge impact - The attacker can become root and execute any command as root

Unix-like systems use ACL

POSTECH

- Interacting with files in Unix-like systems through syscalls
  - 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

- Interacting with files in Unix-like systems through syscalls
  - open’s error check example:

```
#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`

Unix-like systems use ACL

- Interacting with files in Unix-like systems through syscalls
  - 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!

Attacking Access Control

TOCTOU vulnerability

- Time-of-check-time-of-use (TOCTOU)
  - Access right 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)
  - What if file permission is changed during this time?

TOCTOU vulnerability

- Time-of-check-time-of-use (TOCTOU)
  - Example: `vi` (text editor) - predecessor of `vim` (VI iMproved)
    - `vi` keeps a backup of the original file upon save
      - Save: **rename** the original file as a backup and **create** a new file with the original name
    - If we run `vi` as root, modify Alice's file and save,
      - `rename("alice_file", "alice_file.bak");` retains the permissions
      - `open("alice_file", O_CREAT);` → this is owned by root (since `vi` is running as root)
      - `vi` needs to change the owner to Alice by invoking `chown` syscall

TOCTOU vulnerability

- Time-of-check-time-of-use (TOCTOU)
  - Example: `vi` (text editor) - predecessor of `vim` (VI iMproved)

vi

```
rename("alice_file", "alice_file.bak");
    ↓
open("alice_file", O_CREAT);              TOC
    ↓
chown("alice_file", alice_uid, alice_gid); TOU
```

Attacker

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

Q) How can we prevent such attacks?

Result: `/etc/passwd` is owned by Alice

TOCTOU vulnerability

- Time-of-check-time-of-use (TOCTOU)
  - Example: `vi` (text editor) - predecessor of `vim` (VI iMproved)

vi

```
rename("alice_file", "alice_file.bak");
        ↓
open("alice_file", O_CREAT);            TOC
        ↓
chown("alice_file", alice_uid, alice_gid);   TOU
```

Attacker

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

Result: `/etc/passwd` is owned by Alice

Prevention:

1. Make window as small as possible (although not failproof)
2. Use unpredictable path for alice_file
3. Create alice_file in a secure directory

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?