

CY2550 Foundations of Cybersecurity

Access Control

Alina Oprea

Associate Professor, Khoury College

Northeastern University

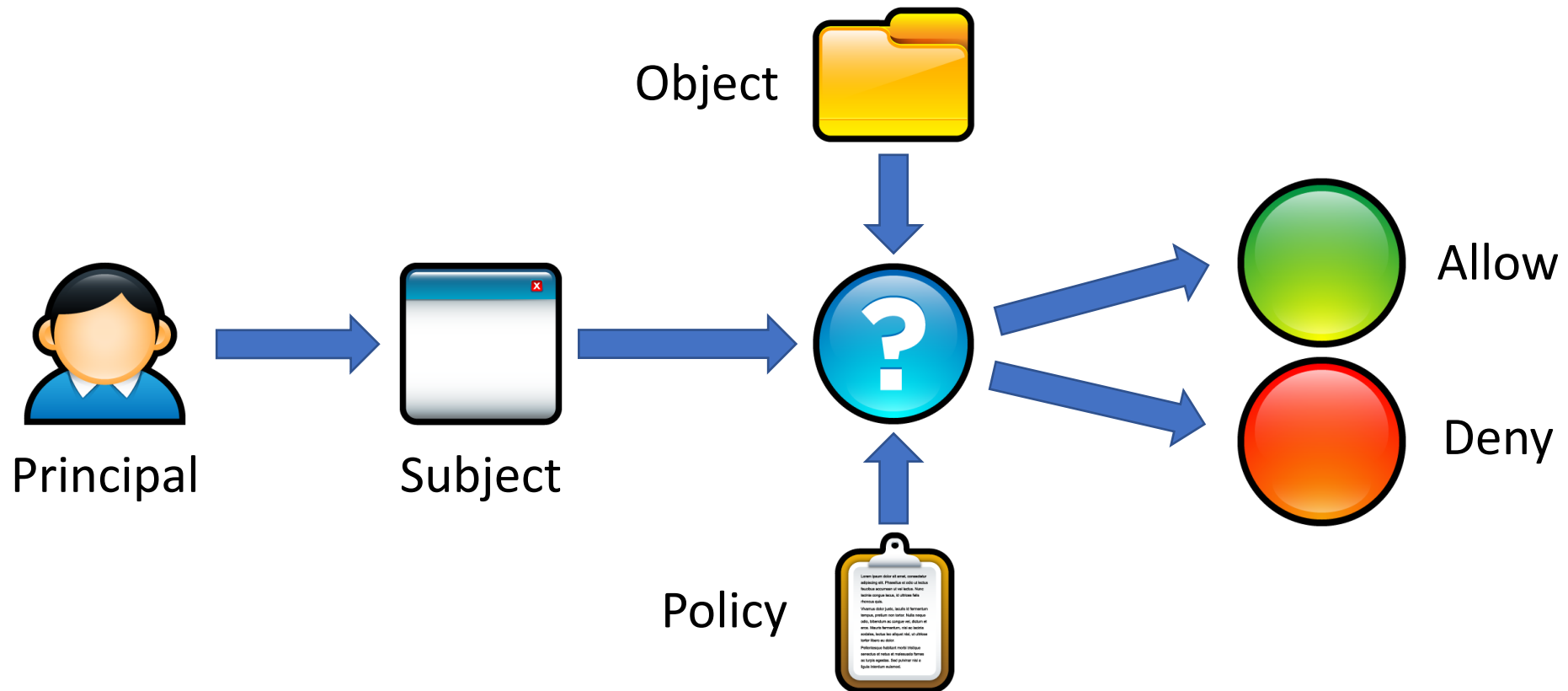
March 19 2020

Announcements

- Hope everyone is safe!
- Lectures are recorded and link posted on Piazza
- Everyone can use video in Zoom
- Can raise hands for questions
- Can use chat window if you have questions
 - TAs monitor the chat window
- Assignment on social engineering and ethics moved to next Thursday, March 26

Access Control Check

- Given an access request from a **subject**, on behalf of a **principal**, for an **object**, return an access control decision based on the **policy**

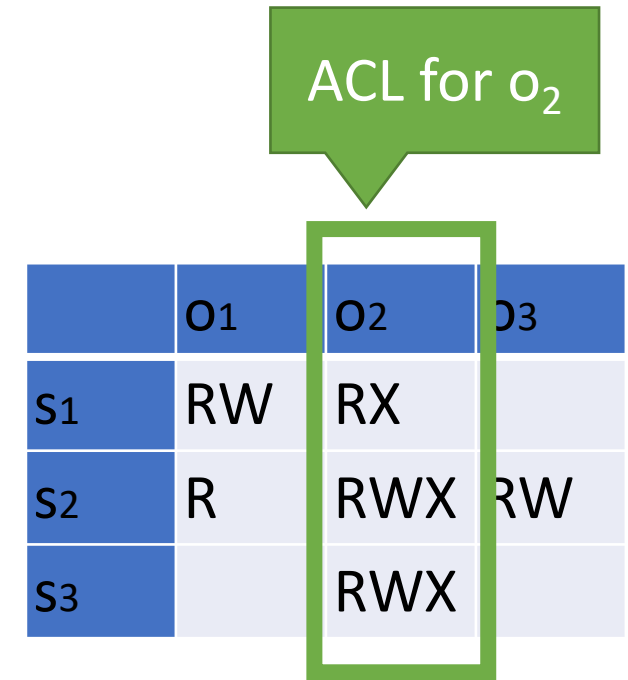


Access Control Models

- **Discretionary Access Control (DAC)**
 - The kind of access control you are familiar with
 - Most widely deployed (Windows, Unix)
 - Access rights propagate and may be changed at subject's discretion
 - Owner of resource controls the access rights for the resource
- **Mandatory Access Control (MAC)**
 - Access of subjects to objects is based on a system-wide policy
 - Global policy controlled by system administrator
 - Might deny users full control over resources they create

DAC: Access Control List (ACL)

- ACL per object
 - A column in access control matrix
- Each object has an associated list of permissions for each subject
- Authorization verified for each request by checking list of tuples
- Implemented by Windows
- **Very flexible, but complicated to manage**



The diagram shows an Access Control Matrix (ACM) with subjects (S1, S2, S3) as rows and objects (O1, O2, O3) as columns. The permissions are as follows:

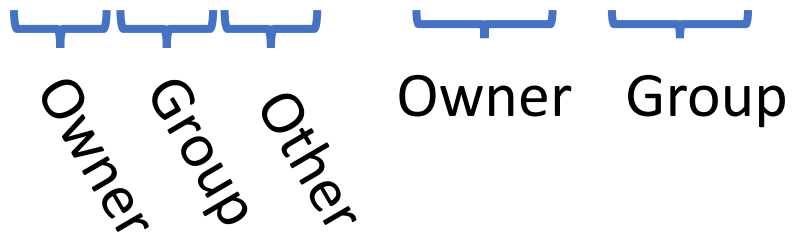
	O1	O2	O3
S1	RW	RX	
S2	R	RWX	RW
S3		RWX	

A green box highlights the column for object O2, and a green callout bubble points to it with the text "ACL for o₂".

Access control
matrix

DAC: Unix Permissions

```
alice@DESKTOP:~$ ls -l
drwxrwxrwx 0 alice  alice   512 Jan 29 22:46 my_dir
-rw-rw-rw- 1 alice  alice    17 Jan 29 22:46 my_file
-rwxrwxrwx 1 alice  faculty 313 Jan 29 22:47 my_program.py
-rw----- 1 root   root   896 Jan 29 22:47 sensitive_data.csv
```

The diagram illustrates how the first three characters of a Unix permission string are grouped. For the first line 'drwxrwxrwx', the first three characters 'd', 'r', and 'w' are grouped under the label 'Owner'. The next three characters 'w', 'x', and 'r' are grouped under the label 'Group'. The final three characters 'r', 'w', and 'x' are grouped under the label 'Other'. This same structure is shown for the other lines, with the first three characters of each line's permission string being grouped under 'Owner', 'Group', and 'Other' respectively.

Very simple, easy to manage, but not all policies can be supported

Problems with Principals in DAC

setuid

The Confused Deputy Problem

Capability-based Access Control

From Principals to Subjects

- Thus far, we have focused on **principals**
 - What user created/owns an object?
 - What groups does a user belong to?
- What about **subjects**?
 - When you run a program, what permissions does it have?
 - Who is the “owner” of a running program?

Process Owners

```
alice@DESKTOP:~$ ls -l
-rwxr-xr-x 1 alice alice 313 Jan 29 22:47 my_program.py
alice@DESKTOP:~$ ./my_program.py
...
```

alice is the
owner. Why?

Who is the
owner of this
process?

```
alice@DESKTOP:~$ ps aux | grep my_program.py
alice      tty1      S        01:06   0:00 python ./my_program.py
```

Process Owners

```
alice@DESKTOP:~$ ls -l /bin/ls*  
-rwxr-xr-x 1 root root 110080 Mar 10 2016 /bin/ls  
-rwxr-xr-x 1 root root 44688 Nov 23 2016 /bin/lsblk  
alice@DESKTOP:~$ ls
```

Who is the
owner of this
process?

alice is the
owner. Why?

```
alice@DESKTOP:~$ ps aux | grep ls  
alice      tty1      S        01:06   0:00 /bin/ls
```

Subject Ownership

- Under normal circumstances, subjects are owned by the principal that executes them
 - **File ownership is irrelevant**
- Why is this important for security?
 - A principal that is able to execute a file owned by root should not be granted root privileges
 - In previous example, file /bin/lis is owned by root, alice can execute it but does not get root privileges

Corner Cases

```
alice@DESKTOP:~$ passwd  
Changing password for alice.  
(current) UNIX password:
```

- Consider the *passwd* program
 - All users must be able to execute it (to set and change their passwords)
 - Must have write access to */etc/shadow* (file where password hashes are stored)
- Problem: */etc/shadow* is only writable by root user

```
alice@DESKTOP:~$ ls -l /etc/shadow  
-rw-r----- 1 root shadow 922 Jan  8 14:56 /etc/shadow
```

setuid

- Objects may have the *setuid* permission
 - Program may execute as the **file owner**, rather than **executing principal**

```
alice@DESKTOP:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 47032 May 16 2017 /usr/bin/passwd
alice@DESKTOP:~$ passwd
Changing password for alice.
(current) UNIX password:
```

```
alice@DESKTOP:~$ ps aux | grep passwd
root      tty1      S        01:06   0:00 passwd
```

chmod Revisited

- How to add *setuid* to an object?

```
chmod u+s <file1> [file2] ...
```

```
chmod 2### <file1> [file2] ...
```

- **WARNING: SETUID COULD HAVE SERIOUS SECURITY IMPLICATIONS**
 - Only set *setuid* on compiled binary programs
 - Enable additional checks for security (e.g., `passwd` checks the user ID of the executing principal and can only change the password entry in the shadow file for that user)

Another setuid Example

- Consider an example *turnin* program

/cy2550/turnin <project#> <in_file>

1. Copies *<in_file>* to *<project#>* directory
 2. Grades the assignment
 3. Writes the grade to */cy2550/<project#>/grades*
- Challenge: students cannot have write access to project directories or grade files
 - *turnin* program must be *setuid*

Executed by
student

```
alice@login:~$ /cy2550/turnin project1 pwcrack.py  
/cy2550/project1/pwcrack.py
```

Thank you for turning in project 1.

```
alice@login:~$ ls -l /cy2550/
```

```
drwx--x--x 0 alice  faculty      512 Jan 29 22:46 project1  
-rwsr-xr-x 1 alice  faculty       17 Jan 29 22:46 turnin
```

```
alice@login:~$ ls -l /cy2550/project1/
```

```
-r-x----- 0 alice  faculty      512 Jan 29 22:46 pwcrack.py  
-rw----- 1 alice  faculty       17 Jan 29 22:46 grades
```

- pwcrack.py is created in project1 (owned by alice)
- grades modified by turnin script (executed as alice)

Ambient Authority

- Ambient authority
 - A subject's permissions are automatically exercised
 - No need to select specific permissions
- Systems that use ACLs or Unix-style permissions grant ambient authority
 - A subject automatically gains all permissions of the principal
 - A setuid subject also gains permissions of the file owner
- Ambient authority is a security vulnerability



The Confused Deputy Problem

```
mallory@login:~$ /cy2550/turnin project1 pwcrack.py  
/cy2550/project1/grades  
Thank you for turning in project 1.
```

- The *turnin* program is a **confused deputy**
 - It is the deputy of two principals: *mallory* and *alice*
 - *mallory* cannot directly access */cy2550/project1/grades*
 - However, *alice* can access */cy2550/project1/grades*
- Key problem: the subject cannot tell which principal it is serving when it performs a write

Preventing Confused Deputies

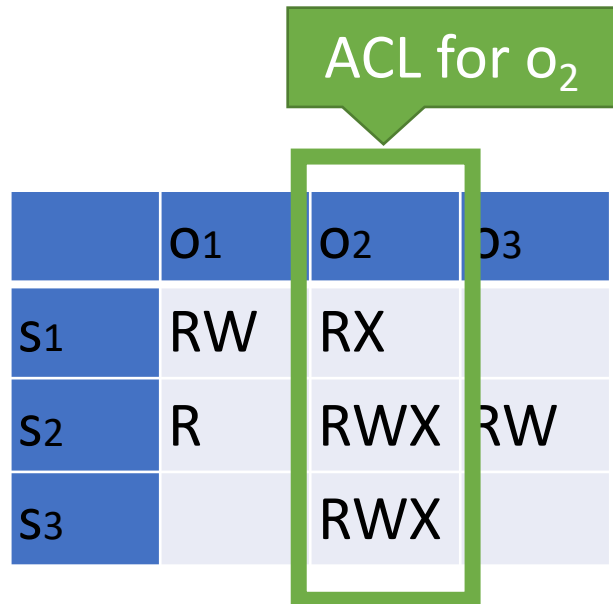
- ACL and Unix-style systems are fundamentally vulnerable to confused deputies
 - Ambient authority provides all permissions of the principal to programs
- Solution 1:
 - Pass the identity of the executing principal to every program with setuid permission
 - Passwd does this and checks the specific entry in shadow file matches the executing principal
- Solution 2: move to **capability**-based access control system
 - More general solution



Capabilities

ACLs

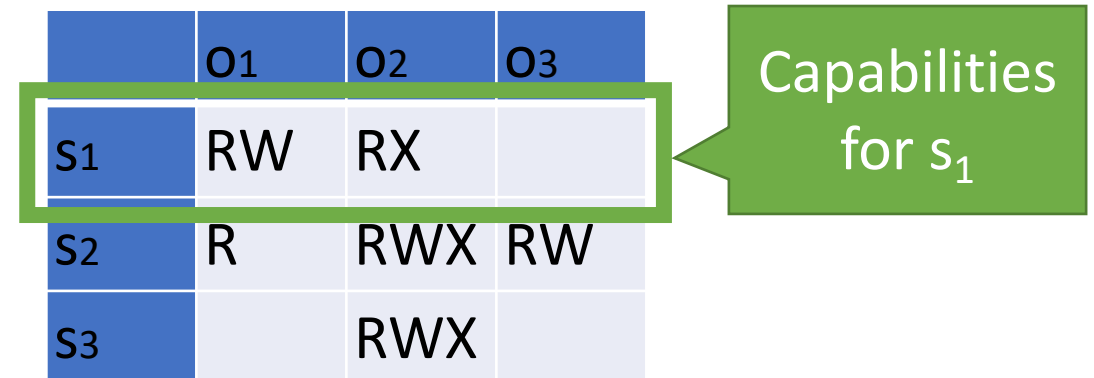
- Encode columns of an access control matrix



	O1	O2	O3
S1	RW	RX	
S2	R	RWX	RW
S3		RWX	

Capabilities

- Encode rows of an access control matrix



	O1	O2	O3
S1	RW	RX	
S2	R	RWX	RW
S3		RWX	

Capabilities vs. ACLs

- Consider two security mechanisms for bank accounts

1. Identity-based

- Each account has multiple authorized owners
- To authenticate, show a valid ID at the bank
- Once authenticated, you may access all authorized accounts

- ACL system
- Ambient authority to access all authorized accounts

2. Token-based

- When opening an account, you are given a unique hardware key
- To access an account, you must possess the corresponding key
- Keys may be passed from person to person

- Capability system
- No ambient authority

How Can Capabilities Be Implemented?

- A capability can be thought of as a pair (x, r) where x is the name of an object and r is a set of privileges or rights
 - For each subject, we can store its capabilities
 - The subject presents to the OS a capability to get access to an objects
 - Capability is disconnected from user's identity
- Store process capabilities in special capability segments, only writable by kernel, but read by other processes
 - Need hardware support to manage capabilities
 - The Plessey System 250 and CAP systems
- Amoeba use encrypted capabilities, where each capability is encrypted using a key only available to the Amoeba kernel
 - Software implementation
 - $Enc_k(x, r)$ is the capability; with key k known only to the kernel
 - Can be transferable to other processes

Capability-based Access Control

- A capability is a **token, ticket, or key that gives the possessor permission to access an object** in a computer system
- Subjects need to present capabilities which:
 - Give them access to objects (e.g., files in file system)
 - Are **transferable** (can be passed from subject to subject)
 - Are **unforgeable** (can not be created by unauthorized users)
- Can be implemented in hardware or software
- Why do capabilities solve the confused deputy problem?
 - When attempting to access an object, a capability must be selected
 - The program can check that the subject has the permissions

Confused Deputy Revisited

Principal	...	/home/mallory/*	/cy2550/project1/grades	...
mallory	...	RWX	---	...

Allow (owned by
mallory)

```
mallory@login:~$ /cy2550/turnin project1 /home/Mallory/pwcrack.py  
ls /cy2550/project1/grades
```

Deny

- Principal running turnin program must pass capabilities to objects (files) at invocation time
 - *mallory* has permission to access pwcrack.py
 - *mallory* does not have permission to access */cy2550/project1/grades*
- No ambient authority in a capability-based access control system
 - Principal cannot pass a capability it doesn't have

Capabilities In Real Life

- From a security perspective, capability systems are more secure than ACL and Unix-style systems
- ... and yet, most major operating systems use the latter
- Why?
 - Easier for users
 - ACLs are good for user-level sharing, intuitive
 - Capabilities are good for process-level sharing, not intuitive
 - Easier for developers
 - Processes are tightly coupled in capability systems
 - Must carefully manage passing capabilities around
 - In contrast, ambient authority makes programming easy, but insecure

Small Steps Towards Capabilities

- Some limited examples of capability systems exist
 - **Android/iOS app permissions**
 - User must grant permissions to apps at install time
 - May only access sensitive APIs with user consent
 - **POSIX capabilities**
 - Specified processes may be granted a subset of root privileges
 - CAP_CHOWN: make arbitrary changes to file owners and groups
 - CAP_KILL: kill arbitrary processes
 - CAP_SYS_TIME: change the system clock

Mandatory Access Control

Multi-level Security

Bell-LaPadula Model

Biba Model

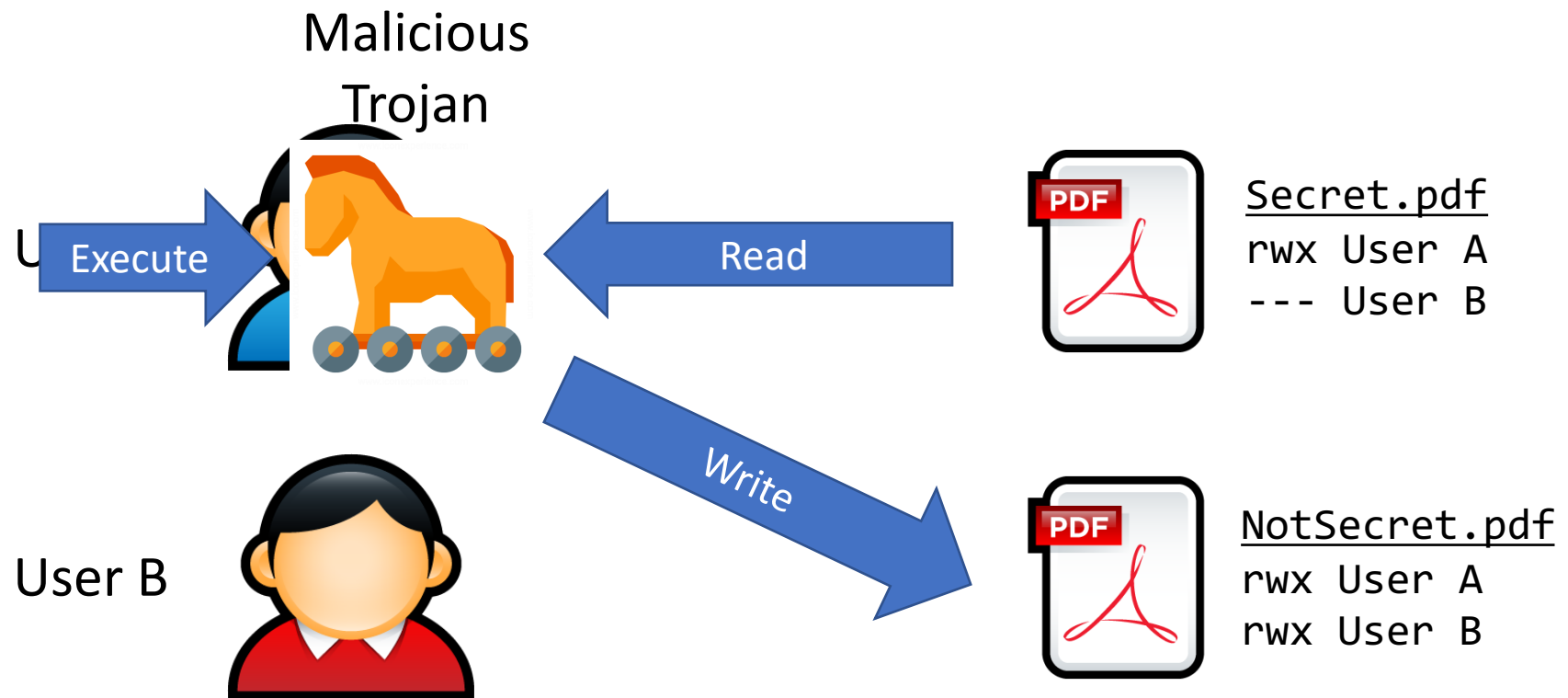
Keeping Secrets?

- Suppose we have secret data that only certain users should access
- Is DAC enough to prevent leaks?

```
charlie@DESKTOP:~$ groups
charlie topsecret
charlie@DESKTOP:~$ ls -la /top-secret-intel/
drwxr-xr-x 0 root root      512 Jan  8 14:55 .
drwxr-xr-x 0 root root      512 Oct 11 19:58 ..
-rw-r----- 1 root topsecret 896 Jan 29 22:47 northkorea.pdf
charlie@DESKTOP:~$ groups mallory
mallory secret
charlie@DESKTOP:~$ ls -la /home/mallory
drwxrwxrwx 0 mallory mallory  512 Jan  8 14:55 .
drwxr-xr-x 0 root      root    512 Oct 11 19:58 ..
charlie@DESKTOP:~$ cp /top-secret-intel/northkorea.pdf /home/mallory
charlie@DESKTOP:~$ ls -l /home/mallory
-rw-r----- 1 charlie charlie 896 Jan 29 22:47 northkorea.pdf
charlie@DESKTOP:~$ chmod ugo+rw /home/mallory/northkorea.pdf
```

Failure of DAC

- DAC cannot prevent the leaking of secrets



Why is DAC Vulnerable?

- Implicit assumptions
 - Software is benign
 - Software is bug free
 - Users are well behaved
- Reality
 - Software is full of bugs (e.g., confused deputies)
 - Malware is widely available
 - Users may be malicious (insider threats)



Towards Mandatory Access Control (MAC)

- Mandatory access controls (MAC) restrict the access of subjects to objects based on a system-wide policy
 - System security policy (as set by the administrator) entirely determines access rights
 - Denying users full control over to resources that they create
- Often used in systems that must support **Multi-level Security (MLS)**
- Implemented in SELinux and AppArmor for Linux

Review Access Control

- Two main methods
 - DAC: ACL (Windows-style) or Linux style (3 levels of permissions per object)
 - MAC
- Main issues with DAC
 - Ambient authority (subjects inherit all permissions of principals)
 - Confused deputies (subject doesn't know which principal it serves)
 - Fixes: capability-based access control
 - Hardware and software implementations exist
 - Solves confused deputy problem
 - Challenging to adopt in practice