ILLINOIS TECH

College of Computing

CS 450 Operating Systems
Introduction to OS Security

Yue Duan

Secure Systems

- A secure system will
  - do what is expected
  - not do the unexpected
- Example:
  - do more: reveal secrets
  - do less: fail to store or retrieve information

Security Properties: CIA

- **Confidentiality**: keeping secrets
  - who is allowed to learn what information
- **Integrity**: permitting changes
  - what changes to the system and its environment are allowed
- **Availability**: guarantee of service
  - service should be “timely”

Security in Computer Systems

- Gold Standard for Security [Lampson]
  - Authorization:
    - mechanisms that govern whether actions are permitted
  - Authentication:
    - mechanisms that bind principals to actions
  - Audit:
    - mechanisms that record and review actions

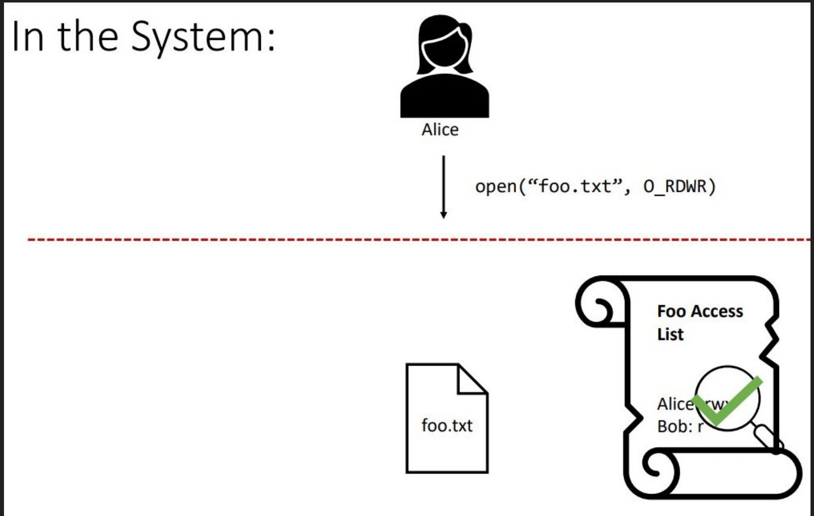
Authorization

- Access control
  - now that we’ve authenticated someone on the system
  - how can we determine whether or not they have access to resources?

	ObjA	ObjB	Objc …
Principal 1	Rw-	rwx	Rw-
Principal 2	---	---	---
Principal 3	R-x	R--	R--

Access Control List

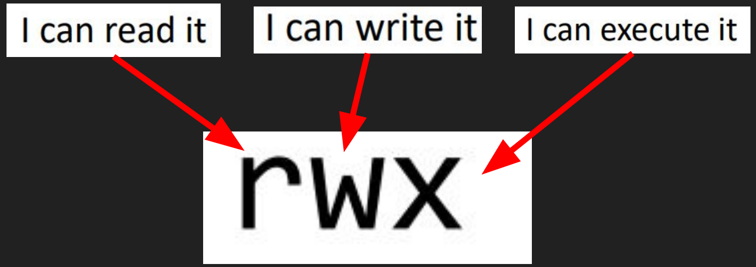
- For each resource (e.g., a file)
  - os manages a list
  - contains allowed principals
  - if requesting agent is not on the list…no beans



Access Control List

- Where to store ACLs?

```
-rw-r--r--  1 yueduan yueduan   77 Apr 22 01:11 hello.c
```



- Which user and group?
  - store UID, GID of the user who created file in inode
  - compare against requester (who called open())

Access Control List

101	101	101
r-x	r-x	r-x
this user	users in this group	anyone else

Access Control List

- Advantages:
  - efficient to review of permissions for an object
  - revocation is straightforward
- Disadvantages:
  - inefficient review of permissions for a principal
  - large ACLs take up space in object
  - vulnerable to **confused deputy attack**

Capabilities

- The *capability list* for a principal P is a list
  - e.g., ⟨dac.tex, {r,w}⟩ ⟨dac.pptx, {r,w}⟩
  - performing operation Op on object Obj requires a principal to hold a capability for Obj and Op
  - *capabilities* must be unforgeable, so they cannot be counterfeited or corrupted
- Note: Capabilities are (typically) transferable.

Capabilities

- Advantages:
  - eliminates confused deputy problems
  - natural approach for user-defined objects
- Disadvantages:
  - review of permissions
  - delegation
  - revocation

ACLs vs. Capabilities

	ACLs: For each Object: <P_1,$privs_1$> <P_2,$privs_2$>…	**Capabilities:** For each Principal: <O_1,$privs_1$> <O_2,$privs_2$>…
Review rights for object O	Easy! Print the list.	Hard. Need to scan all principals' lists.
Review rights for principal P across all objects	Hard. Need to scan all objects' lists.	Easy! Print the c-list.
Revocation	Easy! Delete P from O's list.	If kernel tracks capabilities, invalidates on revocation. Harder if object tracks revocation list.

Authentication

- Something you know
  - Password, PIN, shared secret, etc
- Something you have
  - Keycard, USB key, credit card, key, etc
- Something you are
  - Fingerprint, Iris, facial structure, voice, etc
- In the case of an OS
  - done during login
  - OS wants to know who the user is

Multiple Factors

- Two-factor Authentication:
  - authenticate based on two independent methods
  - for instance:
    - password + secret question
    - password + registered cell phone
- Multi-factor Authentication:
  - two or more independent methods

Passwords

- Secret known only to the subject
- Top 20 passwords suffice to compromise 10% of accounts [Skyhigh Networks]

Top 10 passwords in 2017: [SplashData]

1. 123456
2. password
3. 12345678
4. qwerty
5. 12345
6. 123456789
7. letmein
8. 1234567
9. football
10. iloveyou

16: starwars, 18: dragon, 27: jordan23

Verifying Passwords

- How does OS know that the password is correct?
- Simplest implementation:
  - OS keeps a file with ⟨login, password⟩ pairs
  - user types password
  - OS looks for a login -> password match
- Goal: make this scheme as secure as possible
  - display the password when being typed?

Storing Passwords

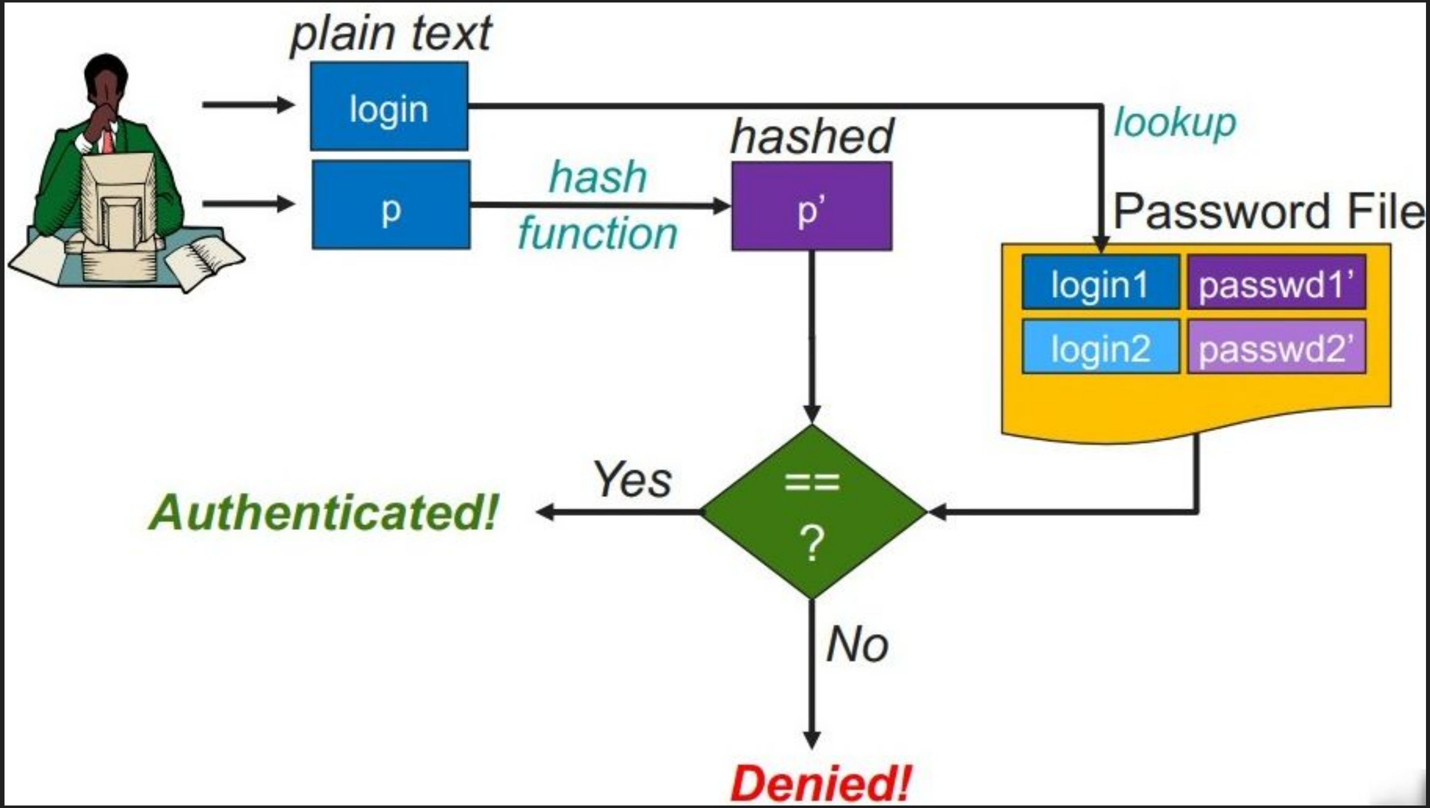
- 1. Store username/password in a file
    - attacker only needs to read the password file
    - security of system now depends on protection of this file!
    - need: perfect authorization & trusted system administrators
- 2. Store login/encrypted password in file
    - access to password file ≠ access to passwords
    - store the hash value!

Hashing

- a function f such that:
  - 1. easy to compute and store h(p)
  - 2. hard to compute p given h(p)
  - 3. hard to find q such that q ≠ p, h(q)==h(p)
- Store h(password)

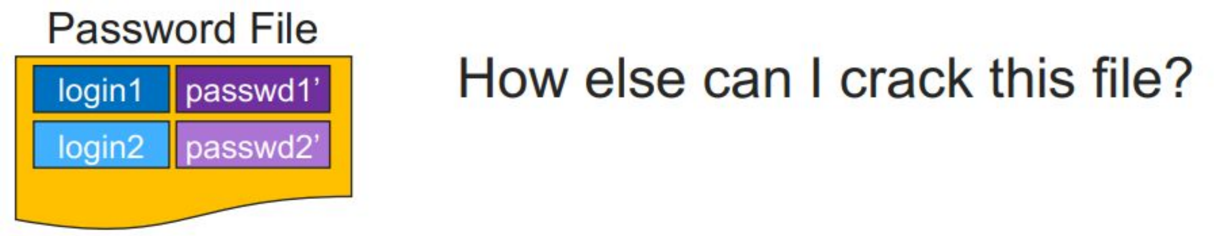
Remember: *h(encrypted-password) ≠ password*
h^{-1}(encrypted-password) = password
h^{-1} hard to compute (hard ≈ impossible)

Storing and Checking Passwords



Hash may not be Enough!

- Suppose attacker obtains password file:
  - /etc/passwd
  - public hash fn + hard to invert ⇒ hard to obtain all the passwords



- Brute Force Attack:
  - enumerate all possible passwords p
  - calculate h(p) and see if it matches
- Dictionary Attack:
  - list all the likely passwords p and check

Salting

- Salt:
  - a unique system-chosen nonce
  - include it as part of each user's password
  - store {username, salt, E(password+salt)}
  - simple dictionary attack will not work

login	salt	h(p||s)
abc123	4238	h(423812345)
abc124	2918	h(2918password)
abc125	6902	h(6902LordByron)
abc126	1694	h(1694qwerty)
abc127	1092	h(109212345)
abc128	9763	h(97636%%TaeFF)
abc129	2020	h(2020letmein)

- If the hacker guesses 12345, has to try:
  - h(000112345), h(000212345), etc.
  - UNIX adds 12-bit salt

THANK YOU!