Announcements
Recap of Crypto

- Symmetric vs. asymmetric

↑
AES
DES, 3DES

↑
RSA
EC
DH

Trade-offs:
RSA is cheaper than Symmetric harder to crack + implement (to design & setup) Public-key is slower
Digital Signatures

In some cases, we’re not worried about secrecy, but about authenticity.

→ Want a guarantee that data hasn’t been changed in transit.

Goal:
- Unforgeable & verifiable
- Cheap to compute
- Signed document is unchanged
How? - Encryption!

Being able to encrypt is itself a signature!

If only I knew $K$, then $C = E(M, K)$ is a signature by me.

But how to set up checks with symmetric encryption and/or asymmetric encryption?
Symmetric: - Agree on secret key
    (Infeasible for entire Internet!)
    - Trusted 3rd party
      Certificate authorities

Asymmetric:
    - Signer encrypts w/ private key
    - Receiver can check w/public key
      (no trusted 3rd party)

Problems:  - computationally expensive
            - can hit with man-in-the-middle
Access Control

The prevention of unauthorized use of a resource, including the prevention of use in an unauthorized manner.

Probably the central element of computer security.
Access Control incorporates:

1. Authentication
2. Authorization
3. Audit (later)
1. **Authentication**

4 basic strategies:

1) Something you know - **passwords**
2) Something you possess
3) Something you are
4) Something you do

Which is most common?
Passwords: Common Attacks

- Brute force / dictionary attacks
- Key loggers
- Shoulder surfing
- Phishing/social engineering
- Protocol specific attacks
Defenses against password attacks

- cap logins
- force no dictionary words
- change passwords often
- enforce guidelines
- incorporate questions
- picture recognition
- education
Hashed Passwords

In general, only hashed versions of passwords are saved.

Why? Target minimize risk if broken
Is this enough?

Suppose I get encrypted list. How could I attack, assuming I know the hash function?

(Reasonable - Linux systems all use the same one!)

Take good guesses of passwords, hash them. Look for matches.
Solution: Salts

- Choose a random # for each user id

  Compute \( h(p, s) \) & store with \( s \)

Note - usually stored in plaintext!

Any issue here?

Still vulnerable
Unix Implementation

- User password of 8 digits
  → 56-bit value

- 12-bit salt value, usually based on account creation time

- Hash function (based on DES) is run ~25 times.

- Resulting 64-bit value is converted to a 11-character sequence

Sounds impressive...
In 2003, a super computer managed over 500 million password guesses in 80 minutes.

(Back then, a regular machine could have done the same in a month or so.)

Stronger variants of password verification essentially use stronger but slower hashing algorithms. (One even just runs a dummy for loop!)
More recent

In 2012, Ars Technica challenged 3 hackers to crack 16,000 hashed & salted passwords.

They got over 90% using dictionary attacks in 20 hours.

User education
Single Most Important Defense:

User education!

- choose secure passwords, since dictionary attacks are first effort.
Password checkers

Algorithms that allow or reject passwords based on how likely they are to be cracked.

1. Rule enforcement:
   - at least 8 digits
   - one number, one letter etc.
(2) Markov model: Simple version with 3-letter alphabet
2 (cont)

For English, they start with a dictionary of passwords.

Transitions are based on how common small letter sequences are.

Prev ex: \[
\frac{\text{# strings with 'a'}}{\text{# strings with 'ab'}} = 0.5
\]

(first order model)

Model catches most dictionary passwords, but still user-friendly.
Bloom Filters

Start with dictionary of passwords to avoid.
Take $k$ independent hash functions.
Hash all dictionary passwords:

$$0 \quad 1 \quad 2 \quad 3 \quad \ldots \quad N-2 \quad N-1$$

$$\begin{align*}
H_1(\text{"secret"}) &= 3 \\
H_2(\text{"secret"}) &= N-2 \\
H_3(\text{"secret"}) &=
\end{align*}$$
3 (cont)

When a new password is given, its k hash values are all computed. If all = 1 in hash table, it is rejected.

Note: Could reject good password. Know bad ones get rejected.
Math is beyond this class, but with "good" hash functions,

\[ P[\text{false positive}] \approx (1 - e^{\frac{kD}{N}})^k \]

\[ k = \# \text{ hash functions} \]
\[ N = \# \text{ bits in hash table} \]
\[ D = \# \text{ words in dictionary} \]
Why use Bloom filters?

Simple example: dictionary of 1 million words, so takes ~8 MB.

Suppose we want a .01 probability of rejecting a password not on the dictionary. If we want 6 hash functions, then we need \( \frac{N}{D} = 9.6 \)

\[ \Rightarrow \text{Hash table of } 9.6 \times 10^6 \text{ bits, or 1.2 MB.} \]

Saves space and time.
Token-Based Authentication
(something you possess)

Examples:
- RSA fobs
- cell/text authentication
- ID cards

Attacks:
- Theft

Problem: Loss
Biometric Authentication
(Something you are or do)

- Hard to steal
- Expensive
- People change → hard to make effective
- Possible (if not easy) to fool
A Note About Remote Authentication

Goal: Give eavesdroppers as little info as possible.

Sample (a simple) protocol:

1) user transmits identity
2) host sends a nonce (random #, r) and specifies 2 functions f and h
3) user sends: f(r, h(password))
2. **Authorization: Access Control Policies**

A) Discretionary Access Control
B) Mandatory Access Control
C) Role-Based Access Control

(These aren't necessarily mutually exclusive, either.)
Terminology

- **Subject**: a process or user
  - 3 classes:
    - owner
    - group
    - world

- **Object**: a resource

**Def.** Access rights describe ways which subjects may interact with objects.
Discretionary Access Control (DAC)

- Most common in modern OS

- Based on subject's identity combined with access rights stating what each subject is allowed to do.

Note: An entity may be given access rights which allow it to give another subject access rights.
Access Control Matrix:
DAC model developed by Lampson in '71.

<table>
<thead>
<tr>
<th>SUBJECTS</th>
<th>File 1</th>
<th>File 2</th>
<th>File 3</th>
<th>File 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>User A</td>
<td>Own Read Write</td>
<td>Own Read Write</td>
<td>Own Read Write</td>
<td></td>
</tr>
<tr>
<td>User B</td>
<td>Read</td>
<td>Own Read Write</td>
<td>Write</td>
<td>Read</td>
</tr>
<tr>
<td>User C</td>
<td>Read</td>
<td>Read</td>
<td>Own Read Write</td>
<td></td>
</tr>
</tbody>
</table>

Image taken from course text, with permission.
How to implement?

In practice, this matrix tends to be very sparse.

(Think of the number of files and users on our Linux systems, much less in larger labs.)

So saving it as a matrix is a waste of memory.
ACL

Windows: Access Control Lists

**Good:**
- less space
- fast to check if user gets access when they ask for it

**Bad:**
- difficult to look this up by user
Capability lists: reverse the previous implementation

Good:

Bad:
Mandatory Access Control (MAC)

Based on comparing security labels with security clearances.

Mandatory: a subject with access to some resource may not share access with another subject.

General use: government.
Since the 1960s, DoD (and other agencies) have been employing people to develop MAC policies.

Ex: Biba 2

(We'll see more of these later)
Role-Based Access Control (RBAC)

Access rights are based on what roles the user assumes in the system, rather than the user's identity.

Roles may own or control other roles, as well as files or directories.

RBAC is the "hot new thing":

RBAC is the newest category of access control; it enjoys "widespread commercial use and remains an area of active research"

-- Stallings & Brown
Example of RBAC: Medical practitioners