CS443 – Authentication & Access Control

Note Title 1/16/2013

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 authentication –

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

Goal:

- Unforgeable & verifiable
- Cheap to compute
- Signed document is unchanged

<u>How?</u> - Encryption!

Being able to encrypt is itself a signature!

If only I know 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
- reciever can check w/ public
(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 userid
  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 11-character sequence

Sounds impressive...

In 2003, a supercomputer 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 & slower hashing algorithm.

(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

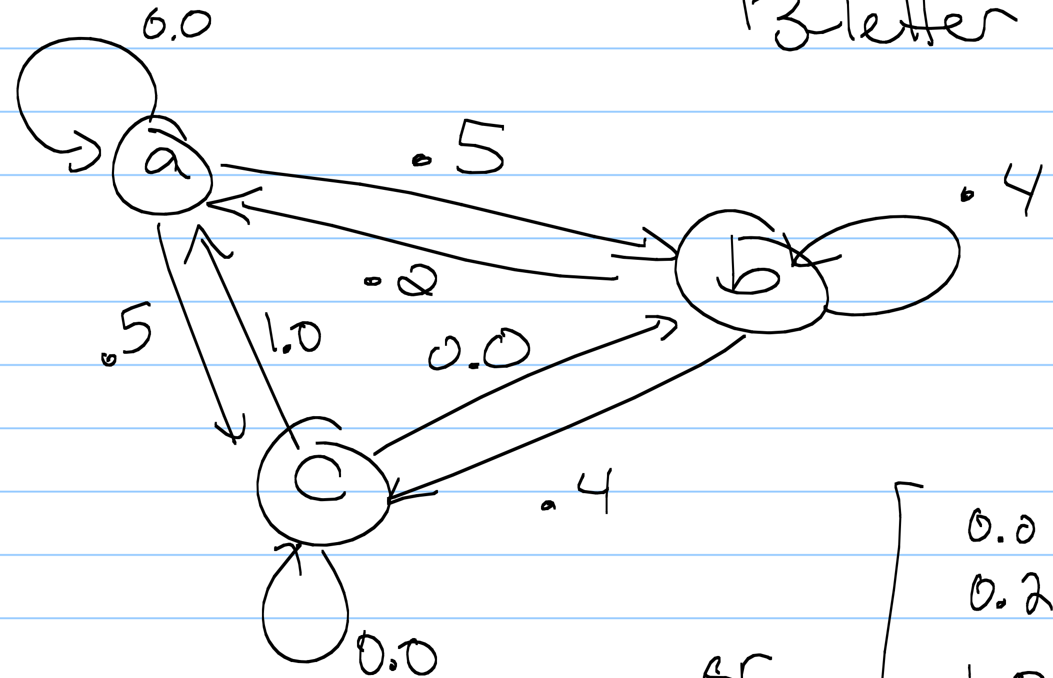
Algorithms that allow or reject passwords based on how likely they are to be cracked.

1. Rule enforcement:

   - at least 8 digits
   - one number, one letter etc.

   . :

② Markov model: simple version with 3-letter alphabet



or

$$\begin{bmatrix} 0.0 & 0.5 & 0.5 \\ 0.2 & 0.4 & 0.4 \\ 1.0 & 0.0 & 0.0 \end{bmatrix}$$

② (cont)

For English, they start with a dictionary of passwords

Transitions are based on how common small letter sequences are.

Prev ex: $\frac{\#\text{strings with 'a'}}{\#\text{strings with "ab"}} = .5$

first order model

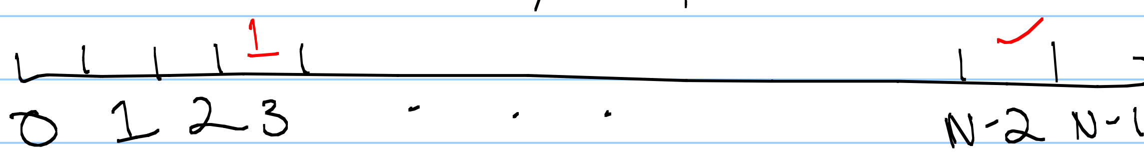
Model catches most dictionary passwords, but still user friendly.

3 Bloom filters

Start with dictionary of passwords to avoid.

Take k independent hash functions.

Hash all dictionary passwords:



H_1("secret") $= 3$

H_2("secret") $= N-2$

H_3("secret") $=$

$\vdots$

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.

③ (cont)

Math is beyond this class, but with "good" hash functions,

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

k = # hash functions
N = # bits in hash table
D = # 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 in the dictionary.

If we want 6 hash functions, then need $\frac{N}{D} = 9.6$

$\Rightarrow$ Hash table of 9.6×10^6 bits, or 1.2 MB.

Saves space <u>and</u> <u>time</u>.

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 (& simple) protocol:

1) user transmits identity
2) host sends a nonce (random #, r) and specifies 2 functions f and h
3) user sends: $f(r, h(\text{password}))$

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

<u>Dfn</u>: 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:

		OBJECTS			
		File 1	File 2	File 3	File 4
SUBJECTS	User A	Own Read Write		Own Read Write	
	User B	Read	Own Read Write	Write	Read
	User C	Read Write	Read		Own Read Write

Image taken from course text, with permission

How to implement?

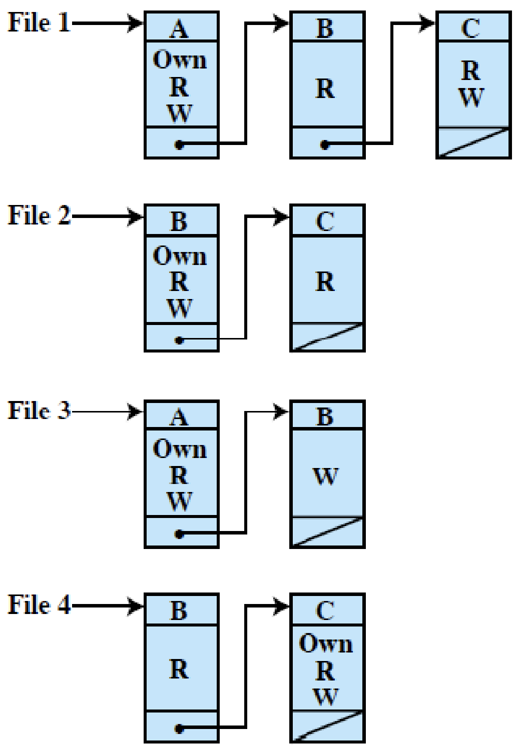
In practice, this matrix tends to be *very* sparse.

(Think of the number of files & 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



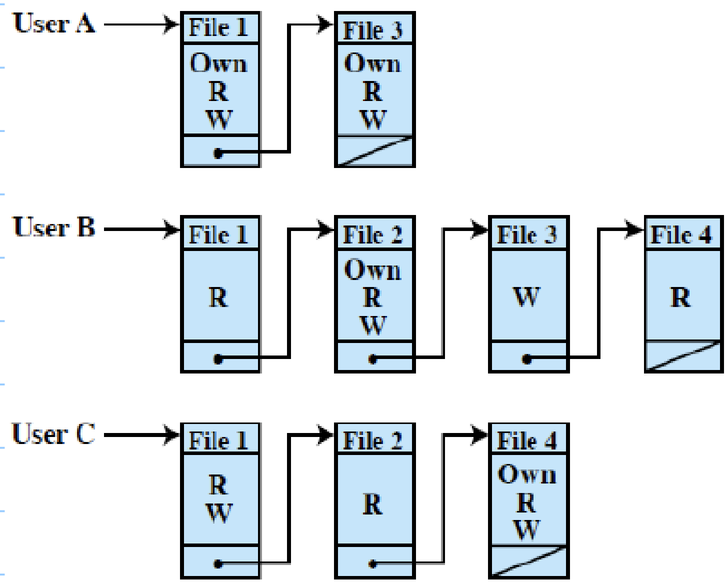
Good: - less space

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)

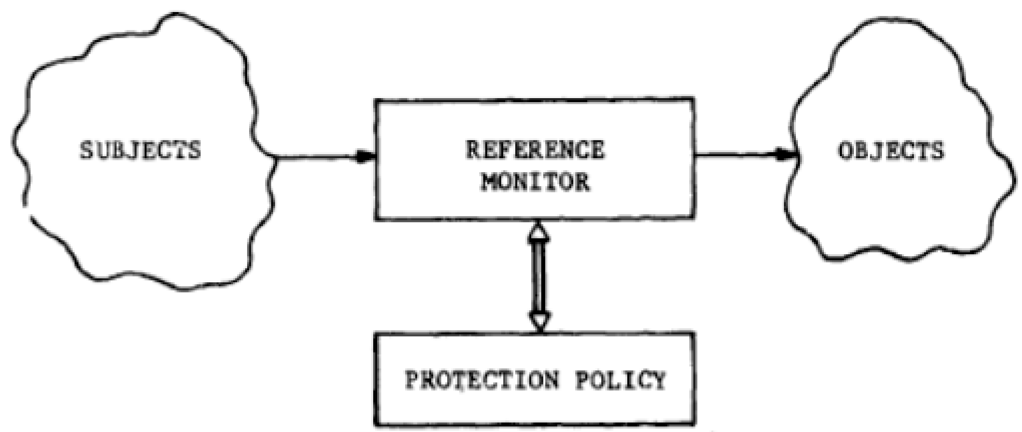
Based on comparing security labels with security clearances.

Mandatory: a subject with access to some resource may <u>not</u> share access with another subject

General use: government

Since the 1960's, DoD (+ other agencies) have been employing people to develop MAC policies

Ex: Biba



(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