CS342 Computer Security **Handout # 5**
Profs. Daniel Bilar and Lyn Turbak **Tue, Sep. 18, 2006**
Wellesley College

Authentication

Sources for this Lecture:

Bishop, *Computer Security: Art and Science*, Chapter 12 (*Authentication*)

Smith, *Authentication: from Passwords to Public Keys*

What is a Authentication?

Authentication binds a subject/principal outside the computer to an identity inside the the computer.

All subsequent stages assume the mapping is correct, so this is really important!

Information for Authentication

- *What you know*: password, cryptographic key, algorithm
- *What you have*: smart card, USB token, RSA SecurID tokens
- *What you are (biometrics)*: fingerprint, voice, etc.
- *Where you are*: e.g., sitting at microfocus Linux workstation; person can't be in two places at once!

Can use multiple forms of information

- two-factor authentication: e.g., smart card/SecurID token (what you have) with PIN/password (what you know)
- GPS device (what you have, where you are)

Authentication Model (Bishop)

$f : A \rightarrow C$ and $I : A \times C \rightarrow Bool$, where

- A is authentication information provided to system
- C is "complementary" information stored in system
- f is "complementation function" producing complementary information (should be a "one-way" functions that's easy to verify but hard to break).
- I is authentication function

What's the purpose of f? Don't want to store "raw" authentication info on computer. E.g., bad to store usernames and passwords in plaintext password file because Mallory knows passwords if (1) he accesses password file or (2) sees passwords communicated with system in plaintext.

Idea: system stores result of one-way function f on password (e.g., a one-way hash or encryption with a secret key).

Linux Passwords

- On older systems, all username/password information stored in publicly readable `/etc/passwd` file:

  username`:`*hashed password*`:`*userid*`:`*groupid*`:`*user info*`:`*home directory*`:`*shell program*
  `foobar:FYKPRGRuRigmq:713:876:Foo Bar:/home/fbar:/bin/bash`
  `bazquux:$1$HEAnjWig$Ni36AoF6ntscTMOKKZ6JK0:714:243:Baz Quux:/home/bquux:/bin/csh`

- Each hashed password contains both "salt" and "hash":

Type	**Salt**	**Hash**
crypt-style password	`FY`	`KPRGRuRigmq`
MD5-style password	`HEAnjWig`	`Ni36AoF6ntscTMOKKZ6JK0`

  We'll see later that salt helps to foil dicitionary attacks.

- Pasword checking algorithm:

  1. User enters username u and password p.
  2. System finds line in `/etc/passwd` for u.
     (a) If line has form u`:`s`h:...`, verify $H(s,p) = h$, where H is DES-based hash function *crypt*
     (b) If line has form u`:$1$`s`$`h`:...`, verify $H(s,p) = h$, where H is MD5-based hash function.
     (c) If no line matches, indicate an error.

- Putting hashed passwords in publicly readable file encourages attacks, so many systems put public info in publicly readable `/etc/passwd` and hashed passwords in private `/etc/shadow`:

  `/etc/passwd` (world readable)

  ```
  foobar:x:713:876:Foo Bar:/home/fbar:/bin/bash
  bazquux:x:714:243:Baz Quux:/home/bquux:/bin/csh
  ```

  `/etc/shadow` (readable by password-checking system)

  ```
  foobar:FYKPRGRuRigmq:11845:0:99999:7:::143255312
  bazquux:$1$HEAnjWig$Ni36AoF6ntscTMOKKZ6JK0:12936:0:99999:7:::
  ```

Problems with Encrypted/Hashed Passwords

- Encrypted passwords can be recorded and replayed (e.g., using packet sniffer).
- Social engineering can be used to find/guess password.
  - *shoulder surfing*: watch user type in password.
  - pretend to be system administrator (e.g. call on phone, phish sites).
  - look in office/home for clues (e.g. post-its on monitor, dumpster diving).
  - ask user for passwords as part of "survey" or "contest".
- If one-way function and password file are public, attacker can launch *glass-box* dictionary attacks.
  - Although brute force usually untractable for *all possible* passwords, it is often tractable for *typical* passwords, which are based on dictionary words.
  - Many password-cracking applications available: John the Ripper, l0phtcrack, etc.
  - "good" passwords help to prevent these attacks, but its hard for humans to remember good passwords (esp. lots of different ones).
  - "algorithms" for generating passwords helps. E.g. "All work and no play makes Jack a dull boy" ⇒ `Aw&0pmJadb`
  - *salting* helps to prevent these attacks: encryption function depends on salt, which increases work of the password cracker.
- If function or password file aren't public, glass-box dictionary attacks aren't possible.
  - Careful, assumptions can be violated!
    - one-way function may become known
    - password file may be obtained by another attack or accidentally released (real ssue in spy world with old keys becoming known, revealing sensitive information.)
  - *black-box* dictionary attacks still possible: try authenticating with actual system using common passwords. Impossible to prevent, but in practice the following are helpful:
    - backoff (longer waits between successive tries)
    - disconnecting after some number of tries.
    - disabling account (but this allows denial of service attacks!)
    - jailing: allow access to slow or impoverished system; related to honeypot notion.

Foiling Replay Attacks

Problem: passwords are reusable and can be replayed. What to do?

- *Password aging*: force users to change passwords on a regular basis.
  - bad social properties: humans will try to circumvent
  - aging period must be chosen relative to time of expected attacks.
- *One-time Passwords (OTP)*: user has a sequence of passwords and uses each only once (they're *non*reusable!).
  - General problems:
    - How to generate password sequence and communicate them to user?
    - How to synchronize user and system? (sequence numbers, clock)
  - Example: RSA SecurID token displays number determined from seed and time. User enters displayed number and PIN/password, which are verified by central server (which also knows seed and PIN/password).
  - Example: S/Key
    - one-way hash function H is to generate sequence of $n+1$ passwords from "seed" p_0: $p_i = H(p_{i-1})$.
    - User is n pairs of the form $\langle i, p_i \rangle$, where i ranges from 1 to n.
    - Invariant: system stores one pair $\langle i, p_{i+1} \rangle$, where i is initially n.
    - System challenges user with i; user supplies p_i; system verifies $H(p_i) = p_{i+1}$; and changes state to $\langle i-1, p_i \rangle$.
    - System is secure from break-in, but where should user passwords be kept?
- *Challenge-Response*: System generates challenge that user responds to. This is covered in notes for simple authentication protocols. E.g., if key K a secret shared between user and system:
  - Challenge = nonce R; response = $E_K(R)$ or $H(K, R)$.
  - Challenge = $E_K(R)$; response = R.
  - Challenge = $E_K(R)$; response = $E_{K+1}(R)$.

Zero-Knowledge Authentication

- Example: Ali Baba's cave
- Example: 3-colorable graphs

Biometrics

- Authentication based on what you are: fingerprint, iris/retina scans (e.g., National Geographic Afghan girl), face, voice, gait, hand geometry, DNA, keystroke/mouse dynamics
- Not a panacea — lots of problems:
  - Accuracy
  - Faking
  - Replay
  - Privacy issues – why should others get your fingerprints, DNA, etc.?