CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 21st, 2011

Authentication

Overview

Authentication is the problem of determining who someone is so that you can make a decision about whether they're allowed to access a resource.

Goal: Verify identity of user

Forms (factors) of authentication:

- Something you know
  - Password
- Something you have
  - A key, a SecurID device, credit card
- Something you are
  - Thumbprint, eye-print, face-print

There is a way to combine all of these, creating what's called **multi-factor authentication,** where you require multiple forms of authentication.

Example:

- typing in a password and also using a SecurID device (2FA)
- typing in a password, using a SecurID device and scanning your thumbprint (3FA)

Note: These factors have to come from **different categories**. For instance, there's really no point in having a user type two passwords.

Authenticating passwords

When using password authentication the problem arises of **how should the server store the password?**

- The server could choose to store the password itself directly
  - The password can be stolen easily if someone breaks in
- The server can store a **hash of your password** (without a salt).
  - A hash function H that is hard to invert is applied on the password
  - This can be brute forced by insiders who obtain the hash of a password
  - Password broken on one site can be used on a different site that hashes the password in the same way
  - An attacker can build a table that maps $H(password) \rightarrow password$, for common passwords, since there's really about a million passwords that people use in practice
- The server could choose to hash each password with a different **salt**, such that every password is stored as:

$$salt \mid H(salt \mid password)$$

  - Every time the user changes his password, the system can **refresh the salt**
  - The attacker now has to **take all the possible salts into considerations when computing his table** for all users, so now he **can only attack one user with one table.**

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 21st, 2011

Recent clever trick: In order to slow down an attacker even more, the hash function can be applied k times repeatedly on passwords.

$$salt \mid H^k(salt \mid password)$$

- k is never stored, and will be different for every user (in fact, it is user controlled)
- to check someone's password, the server will hash the password until it matches the hash in the database (or some threshold value for k is reached)
    - this way, on the server side, the server will never store k for any user

The attacker has a **bigger problem** now building his table: because he will never know how many times to apply H.

Biometrics

One of the main issues with biometrics:

"How many passwords can you have in your life-time?"

"You can have as many as you want."

"Well, with biometrics, you can only have one face-print, 10 fingerprints and two eyes. Tough luck."

Characteristics of biometrics:

- They are **unchangeable**: you cannot change your thumbprint, if someone stole it from you (assuming you ran out of fingers)
- They can be **public**
    - Fingerprints, iris prints, faces, etc.
- They can also be **private**
    - Blood vessel patterns
- They are shared across different accounts
- They are vulnerable to insider threats
    - Most biometrics have to be stored as they were captured (can't really hash them) because when you compare them at authentication you will not deal with the same exact image of your iris/thumbprint so you cannot do a bit-by-bit comparison
    - **Cool trick:** You can use **error correcting codes** along with hash functions to fix this for some biometrics

Iris biometrics

Iris biometrics work by taking an image of your iris and converting it to a 2048-bit string.

$$iris\ image \rightarrow 2048 \text{ bit digest string}$$

Furthermore, two images x and x' of the same iris, have the property that the **Hamming weight** $h(x \oplus x') \simeq 200$. This means that the 2048-bit string for the two iris images will agree in about 1800 positions and differ in about 200 positions.

To authenticate a user:

- Obtain an image of their iris and convert it to a 2048-bit string x
- Compare the obtained string x with the string y stored in the database for that user

- If $h(x \oplus y) \simeq 200$ then successfully authenticate that person, otherwise deny him access.

Note: Evidence shows human irises are very different, so it is highly unlikely that one person's iris will be too similar to another person's iris.

Problem: Server has to store the 2048-bit strings for each user as they are. The server cannot store the hash of the strings because the Hamming weight comparison would not work anymore since different scans of the same iris will produce different bit-strings which will hash to different values.

Solution: There is a cool trick that you can do to prevent people from stealing stored iris image or the 2048-bit digest string using **error correcting codes** and hash functions.

Error correcting codes

Motivation: Correcting transmission errors over unreliable communication channels.

Let $ECC(x) = c$ be the error correcting code for the message x.

Example: You transmit $x|c$ in a packet of data, but the recipient might receive $x'|c'$, which will differ slightly from what was sent (3 bit errors). The recipient has a way of correcting the transmission error by applying a function ECC^{-1} to $x'|c'$:

$$ECC^{-1}(x'|c') \rightarrow x$$

Note: Assuming not too many bits differ in the received message $x'|c'$ (the bit-error threshold is not exceeded) then the inverse function ECC^{-1} will work successfully outputting the original message x. Otherwise, a different message y will be outputted.

Using ECCs with iris biometrics

At registration:

- At user registration, the sever computes the 2048-bit string x from the image of the user's iris and computes $c = ECC(x)$
- The server only stores $hash(x), c$ for a user.
    - c only reveals around 400 bits of information about the iris string

At login:

- a 2048 bit string x' is computed from the user's iris image
- the server applies the inverse ECC^{-1} function to $x'|c$, ideally getting back x
- the server hashes x and compares it to the stored hash to authenticate the user

This allows you to do some **other cool tricks.**

If you have a **hard drive with sensitive info**, you want to encrypt it with as much randomness as possible. One solution is to just use a password, but they are not ideal as we've seen (easy to break, used for other accounts, etc).

It would be **nice to allow users to carry around some extra randomness** to encrypt the laptop HDD, like their fingerprints. The major block is the same as with the iris biometrics: you will not get the same image when you scan your finger twice.

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, November 21st, 2011

Why not **apply the same technique** using ECCs to compute $hash\left(ecc\left(digest(some\ biometric)\right)\right)$ and use that as additional randomness? These are called **fuzzy extractors.**

Leaked biometrics

Problem: If your fingerprint gets posted online, then you have to re-encrypt your entire hard drive.

Solution: Instead of using the fingerprint and password to encrypt the entire hard drive, use them to encrypt the encryption key for the hard drive, such that if the fingerprint gets compromised all you have to do is re-encrypt the encryption key under a new fingerprint and password.

Description of HDD encryption system:

- User inputs password pwd and $x' = \text{finger print scan}$
- System computes $k' = hash(pwd \mid ECC^{-1}(x' \mid c))$, maybe with some salts in there
- System will store:
  - c, the error correcting code for the original finger print scan of the user
  - $Enc_{k'}(k)$, the hard drive encryption key, encrypted under the key k' which is computed as explained above

The system uses this computed k' to decrypt $Enc_{k'}(k)$.

If someone manages to steal your fingerprint, you can just re-encrypt k with a new k'', computed out of another fingerprint.

Laptop authentication

In a hospital, the doctors carry tablets with patient information, but the doctors don't want to authenticate every 30 seconds. It would be nice to authenticate the doctors without having them type a password every time.

Solution: The doctors wear Bluetooth wrist bands which connect to the laptop when the doctor is in range.

When the doctor moves away from the laptop:

- The laptop would encrypt RAM with an encryption key k stored on the wrist band
- Delete its copy of the encryption key k

When the doctor approaches the laptop and is in range:

- The laptop retrieves the key k over a secure Bluetooth channel and decrypts the RAM

It turns out that the time to do this is perfect: as soon as the doctor gets to the tablet, the laptop is ready to go