This is the point in the semester when we start talking about security issues that involve actual people and human behavior, rather than precise cryptographic attacks involving Alice and Bob.

Authenticating People

There are three approaches. We can authenticate:

- Something you *know* (e.g., a password).
- Something you *have* (e.g., a physical key).
- Something you *are* (e.g., your fingerprint).

The primary approach for something you *know* is, of course, the **password**.

Passwords

For passwords, the threat model is as follows:

1. The user picks a password and remembers it.
2. To log in, you present a $(name, password)$ pair.
3. The adversary wants to log in as some user.
4. The adversary *might* be able to compromise the server.

Password database

The first approach: server keeps a "password database" containing $(name, password)$ pairs. The big drawback is that if the adversary can see the password database, then they can impersonate any user because the passwords are stored as plaintext.

This is a **significant** problem both because you might worry that someone can hack in from the outside *and* your own system administrators then have access to sensistive user information.

Salted hash

A better approach is to store $(name, PRF(password, salt||name), salt)$. The salt is a random bit string that doesn't need to be kept hidden.

There are several advantages to this approach:

- We don't store the password as plaintext, so an adversary will be reduced to figuring out the password based on the PRF output.
- The salt guarantees that an attacker can't precompute a hash function because they need to incorporate the salt after they've seen it. In addition, the attacker can't hash a common password and check which users match that hash ("rainbow table" attack) because every user will have a different $salt||name$.

However, the server doesn't have access to your password, which can be a minor inconvenience for users (i.e., server can reset your password but not remind you what it is).

Attacks on passwords

There are several possible approaches to acquiring a user's password:

- **Online**: try to log in as user.
- **Offline**: attacker acquires the password database and performs a computational search over passwords (i.e., in the privacy of his own datacenter).
- **Tricking** the user into disclosing their own password, e.g., through phishing or social engineering. An example of the latter: call up the CEO's secretary, tell her you really need your password to do something urgent for the CEO, etc.
- **Spoofing**: impersonate the server, user logs in, you acquire their password. Similar attacks can be launched on ATM machines to read users' credit cards.
- If user wrote down password, *read it*.
- **"Shoulder surfing"**: Look over the user's shoulder while they type their password.
- Compromise a user's device and record their actions, e.g., with keyloggers.
- Get a user's password from one site and try it on another. (Users have on average three or four passwords, so the odds are good that you can reuse them as an attacker.)
- **Change the password database**: if the attacker can go and change the user's password, then they can get into the site through impersonation.

Most of these attacks involve a combination of trickery and technical means, often exploiting the ways that users tend to behave.

Countermeasures

How can we counter these attacks?

- Teach users *not* to divulge, i.e., educate them on best practices around password management.
- Make guessing *harder*.
  - **vs. online**: Slow down the process. Add a time delay after failures. Lock down the account after some number of failures.
  - **vs. offline**: Slow down the verification process, e.g., by hashing the password n times for some large value of n such that the hashing process takes half or a quarter of a second–not awful for login purposes, but very expensive for hackers launching brute force attacks. Even with a scheme like this, roughly half of passwords are guessable given an hour of CPU time (mostly because users tend to pick guessable or common passwords).
- Reduce guessability, i.e., make users choose uncommon passwords.
  - This is hard to quantify; the only sure way is to make the password truly random, generated from some large space. (Problem, of course, is that users are bad at remembering long, random strings, so they'll write it down, leading to another vulnerability.)
  - **Format rules** are also popular (e.g., "Must include a capital letter"). However, studies have shown that they generally don't improve security because users incorporate these rules in very predictable ways (e.g., by putting a single punctuation mark at the end). They'll useful, but not *that* useful.

In general, there's a tension between *unguessability* and *memorization*. And the problem is only getting worse with Moore's Law: adversaries can guess twice as many passwords per unit time every 18 months, so your password must be drawn from an even larger space.

Password hygiene

Similar to key hygiene. Passwords should be changed periodically and requested frequently (i.e., expire idle sessions). You should also require old passwords in order to change passwords (this is related to the idle session problem: even if a user is logged in, they may have just walked away from their computer, so you should *always* request when changing).

The hard problems remain:

- Initial account setup: you don't have an existing relationship with that user and you need to get their password to them securely.

- Password recovery: users can no longer authenticate themselves at all, but you need to get them into a good state. As the server, you *also* don't know their password. (This is typically done via email, but this reduces the security of your system to that of the email service (and, of course, you can't do this if *you're* the email service).)

Multifactor authentication

This usually involves a password along with something else, which could be:

- A small physical token: usually a small device that displays a six-digit number which is constantly changing.
- A smartphone app: along the lines of Google Authenticator.
- One-time passwords, written on paper.
- Text message to phone.
- Biometrics, e.g., fingerprints.

These are certainly useful and make a big difference because the password becomes insufficient for gaining access.

Evidence-based (or Bayesian) authentication

Treat password entry as *evidence* of identity, but with less than 100% certainty. You then collect all other relevant evidence:

- Geolocation.
- Device identity: has Alice logged in from this device before?
- Software version: is this the version of Chrome that Alice typically uses?
- Behavior patterns: has Alice performed this action before? Does she always upload files this large?

Treat these as clues and reason in a probabilistic way. If you run a substantially large service, you can probably apply reasonably accurate estimates to each of these factors and come up with a good estimate.

If your confidence is too low, then get more evidence. For example, send a text to Alice and ask her to type it in, or ask her to enter her password again after some period of time.

Challenge-response protocols

We might want to avoid sending our password to server.

1. User has a password p.
2. Server comes up with c, a random challenge, and sends it across to the user.
3. The user sends back $PRF(p, c)$.

The advantage here is that an eavesdropper can't replay a log-in session, so we're more robust to man-in-the-middle (MITM) attacks. In addition, if our server is being spoofed and we're actually communicating with an adversary, then they won't be able to recover our password based on what we send them.

Biometrics

Based on measuring a physical aspect of the body. This could be:

- Fingerprint.
- Facial image.
- Iris scan.
- Retina scan.
- Voice.
- Gait.
- Typing patterns.
- Hand geometry.

This has some advantages, namely that these factors are not transferable (or, at the very least, more difficult to replicate). On the downside:

- These measurements must be approximate due to small physiological changes that take place over time.
- These features are publicly observable. For example, you leave fingerprints in public constantly.
- These features can't be changed. For example, you can change password, but you can't get a new fingerprint.

- Still susceptible to spoofing. For example, you can hold up a picture of someone's face to spoof a facial image.

This doesn't mean that we shouldn't *use* these biometric factors. But these technologies tend to work best when, in addition to the biometric, you have an observer checking to see that there's no spoofing going on.