We have looked at how to use public key crypto (mixed with just the right amount of trust) for a website to authenticate itself to a user's browser. What about when Alice needs to authenticate herself to an online service?

Conceptually, Alice could follow the same procedure as an online service: she could generate a public/private key pair, and either directly share the public key with the online service, or she could obtain a certificate from a trusted third party. This is not uncommon in practice: most Unix distributions have a file in each user's home directory, $HOME/.ssh/authorized_keys2 in which a user can list a set of public keys. If Alice can prove that she has one of the equivalent secret keys, then the system allows her to log in as the user.

Unfortunately, this process is too cumbersome for many users, and can decrease a system's usability. To log into, say, Gmail, you would always have to have your private keys on you. This may be easy from your personal computer, but what if you want to log in from a friend's machine?

Moreover, what happens if you lose your private key? Expecting users to remember a 2048 bit number is too cumbersome for most (and as we have seen, it would violate the principle that "Human factors matter").

This motivates the need for a different way to authenticate users.

==

USER AUTHENTICATION

Ultimately, what we need is a way to uniquely identify a user. There are three broad things that differentiate us from one another:

(1) WHAT WE KNOW

The canonical approach to solving user authentication is PASSWORDS: a password is useful only if the individual user is the only one who knows it.

There are other examples, as well: friends can authenticate one another by sharing private details that only the two of them could know (see every body-swap movie ever). When accessing a credit report, you are often asked questions that few people besides yourself would be able to answer (all of your prior addresses, business affiliations, etc.). Of course, anyone with access to your credit report gains access to these things.

Because we so often prove our identities with what we know, our private information often becomes somehow intermingled with our very identity. As a result, when someone obtains information that is typically used to authenticate a victim, we say that that victim has had his or her identity stolen.

The main challenge with authenticating based on what we know is that .. um .. wait, I forgot.

(2) WHAT WE HAVE

Another common form of authentication is to have a physical object that would be very difficult for us to obtain if we were not who we claimed to be. Classic examples include badges and keys.

A more recent example is a "token": a small device that is easy to carry around (people often attach them to their physical key ring), and that has a small LED screen and a button. Upon pressing the button, a small piece of code generates a seemingly random number and displays it on the screen. When Alice wishes to authenticate herself using this token, she enters the number (almost always along with another mode of authentication, like user name and password). The server authenticating Alice runs a corresponding piece of code that generates the same seemingly random number as Alice's token. If they match, then the user who entered it is very likely to have the physical token.

In practice, security based on things that we have boils down to two very difficult tasks: (1) developing a physical object that is difficult to forge, and (2) developing a physical object that is difficult to tamper with or reverse engineer.

In the case of the token, it is imperative that an attacker who gains temporary physical access not be able to extract the seed used in computing new numbers.

(3) WHAT WE ARE

Often viewed as the holy grail of authentication, BIOMETRICS are physical properties of our bodies that make us unique and that are easy to measure. The canonical example is fingerprints: surprisingly, it was not until the late 19th century until anyone floated the idea of using fingerprints to identify people! And it was not until the early 20th century when fingerprint analysis was applied in a criminal investigation in the US.

There are other examples, each with a different trade-off between accuracy and cost:

Body part	Accuracy	Cost
Face	Low	Low
Fingerprint	Medium	Medium
Iris	High	High

What makes biometrics so appealing is that, if it works, it requires virtually no effort on behalf of a user to authenticate himself or herself. Simply by existing and being our unique special selves, we are doing our job. Go us!

The problem is that, at least with many existing techniques, it has been quite easy to forge another's biometrics, from taking a picture of someone's eye to making a mould of someone's fingerprints.

(4) OTHERS

There have also been other schemes, such as WHERE we are, how we react to emotional stimuli (such as Rorschach tests), and so on. These are more difficult to classify into the above three, but they all serve a very similar purpose: the goal is to try to set an individual apart from others in a way that "comes naturally", that is, without requiring users to have to worry about forgetting or creating a good way to authenticate themselves.

Often, two of the above approaches are used in conjunction, a process called TWO-FACTOR AUTHENTICATION. The most common form of two-factor authentication combines something we know (our password) and something we have (a secure token). Either one of these alone can be relatively easy to guess (after all, we often use pretty poor passwords, and the token only generates a relatively small string of numbers), but when taken together, they are often an effective means of secure authentication.

There are many interesting results in designing and building trusted, tamper-resistant hardware, as well as in designing and building devices to accurately measure biometrics. But these are beyond the scope of this discussion. We will focus on what is by far the most common forms of user authentication: passwords.

==

PASSWORDS

Abstractly, a password scheme is one that consists of three algorithms:

(1) SETTING the password: here, Alice establishes an initial password with Bob, and Bob updates some of his local state. (Imagine for the time being that he stores her password, but we will see in a moment that this does not achieve the security properties we want).

(2) LOGGING IN: after setup, Alice provides her password to Bob. Bob checks this against his local state, and if there is a match, then he grants Alice access to his system. Otherwise, he rejects access, possibly even locking anyone from logging in as Alice for some period of time.

(3) RECOVERING a forgotten password: We've all been there. If Alice forgets her password, then ideally there is a protocol by which she can recover it. This often requires falling back on a different form of authentication---going to the IT help desk would be a form of what we are plus what we have (student ID); having a new password sent via text to our cellphones is a form of what we have (a phone tied to that phone number).

These should all seem very familiar to you. But take a moment and consider: what kind of information does Bob store to be able to verify Alice's password?

==

STORING PASSWORD DATA AT A SERVER

Just like with our cryptographic primitives, we can also define the correctness and security properties that we want from a user authentication scheme:

CORRECTNESS:

If Alice has set up a password P, and if she tries to log in with P, then she gains access.

SECURITY: While we normally thing of security from the perspective of how difficult Alice's password is to predict, we are going to focus on the security at the *server*:

Suppose an attacker has gained access to Bob's server, and all of the local state that he has stored for all of his users. A user authenticating service is secure if gaining access to this server does not allow an attacker to recover users' passwords.

This security property has two important ramifications that will help guide our design. In particular, it makes the following two approaches fail:

FAILED ATTEMPT 1: Store the passwords in plaintext

In this strawman protocol, Bob simply store the user's password in plaintext in some server-side file. Setting, verifying, and recovering a password becomes trivial---for both the server and the attacker!

FAILED ATTEMPT 2: Store the passwords encrypted with Bob's secret key

Next up, suppose that Bob generates a symmetric key K and, instead of storing the password in plaintext, stores Enc(K,password), e.g., with AES-128 in CBC mode and a properly generated IV. Verifying a user's login would thus involve decrypting the stored password and checking it against what the user provided.

If the attacker is unable to gain access to K, then this protocol has promise for a few reasons: if Bob uses a properly generated, random IV for each user's password, then recovering a single user's password does not reveal another user's password. This is because each invocation of Enc(K,password) is likely to result in a seemingly random value.

However, recall our assumption: that the attacker gains access to *all* of Bob's local state. This includes not just the file in which he stores the encrypted passwords, but also his secret key K used for encrypting and decrypting the passwords. As a result, the attacker can simply decrypt all users' passwords---no better than storing all of the user passwords in plaintext!

The problem here is that, although we obfuscated the passwords (with encryption), this obfuscation was invertible using state local to Bob.

The problem with the first failed approach was that we added no obfuscation of the passwords whatsoever. The problem with the second failed approach was that, although we obfuscated the passwords (with encryption), this obfuscation was invertible using state local to Bob (the symmetric key).

The idea is to use a *non-invertible* means of obfuscation: our trusty one-way friend, the hash function. But, don't we need a way to recover the password from our saved local state? No, we don't! All we need to do is to be able to determine whether the password a user provides in a login attempt matches the latest password they asked Bob to save. So, if we were to store H(p), then when a user tries to log in, she provides p', then all we need to do is to hash their login input, H(p'), and compare it to what we have stored, H(p). Because a good hash function is collision resistant, it is extremely unlikely that H(p) = H(p') unless p = p'.

PARTIAL SOLUTION 1: Hash functions

So let's analyze a scheme where Bob locally stores H(password) for each user: is this going to be enough? Recall that, so long as H is a good hash function (e.g., SHA-256), then it is pre-image resistant: given H(x) where x is chosen at random from a VERY LARGE SET, it is very difficult to recover x. However, passwords are not chosen at random from a very large set! Rather, users' passwords typically come from a very small set (as far as "large" goes in cryptography: case in point, AES keys are chosen at random from a set of size at least 2^128, while users' passwords typically come from a set of size about 2^22).

What this means is that, if we only store H(password) for each user, then we are susceptible to an OFFLINE DICTIONARY ATTACK:

```
1. For each word w in a "dictionary" of common user passwords
   - Store w and H(w)
2. When the attacker gains access to the database of passwords
   - For each 'H(password)' stored in the database
   - For each w in the dictionary:
      - Compare H(w) to the stored H(password)
      - If they match, then the user's password is w.
```

Note that step 1 can be precomputed, greatly reducing the amount of time the attacker needs access to Bob's server before being able to compromise a user's account.

But that's not all: since two users may end up choosing the same password as one another ("123456" was recently named the most common password), an attacker could target his efforts by first seeing which H(password) appears most commonly in Bob's database: recovering that one password provides the attacker with access to multiple user accounts.

PARTIAL SOLUTION 2: Salted hash functions

The problem with the above partial solution is that there simply is not enough randomness in the inputs to the hash function to let us realize the hash function's properties of being pre-image resistant (i.e., difficult to "invert").

Rather than ask users to "make more random passwords!", we can simply add in a source of randomness ourselves, by concatenating a large, random nonce to the user's password before hashing it.

Nonces seem to take on different names depending on the context: in encryption, nonces were called "initialization vectors." When it comes to hash functions, particularly when storing passwords, we refer to a nonce as a "salt". But make no mistake: what matters is that it's random, and thus chosen from a good source of randomness.

So here's our scheme thus far: when a user saves a password P, Bob's server generates a random salt S, and stores in his database:

```
S and H(S || P)
```

where "||" denotes concatenation.

First the good news: the attacker cannot precompute the password-to-hash mapping like in the offline dictionary attack from above. Rather, the attacker must launch an ONLINE DICTIONARY ATTACK:

```
1. When the attacker gains access to the database of passwords
   - For each 'salt : H(salt || password)'
   - For each w in the dictionary
      - Compare H(salt || w) to the stored H(salt || password)
      - If they match, then the user's password is w.
```

Because the attacker cannot precompute all of the hashes, this takes the attacker more time than in the offline attack: precisely, if the attacker has to try N different passwords before finding a match, and if it takes time T to compute one hash function, then it takes the attacker time N*T to recover a password. This is an improvement, since, without salts, the attacker was able to save this time via precomputation.

Now the bad news: by design, hash functions are FAST! So even though it takes N*T time, T is very small, and thus the overall amount of time is still not a large speedbump to attackers.

This brings us, at last, to our FULL SOLUTION. The high-level idea is to make it take longer to test any one password by taking, not one, but MANY HASH FUNCTIONS of the user's password:

```
(STORING)  When the user stores a password P
         - Compute a random salt S
         - Compute H(H(H(.....(H(S || P)))))....), i.e., the hash
```

of the hash of the hash of.... (k times)
  * Refer to this is H^k(S || P)

(VERIFYING) When the user attempts to log in with password P':
- Look up the user's salt S and stored value H^k(S || P)
- Compute H^k(S || P')
- If H^k(S || P') == H^k(S || P), then it is very likely that P == P', so allow the user access.

The idea behind choosing k (the number of times we chain the hash function together) is to balance between:

- Choose a k LARGE enough that it takes the attacker too long to find a password to be useful. (i.e., N*T*k should be large).

- Choose a k SMALL enough that users do not notice (or at least are not annoyed by) the extra time it takes to verify (i.e., T*k should not be extremely large: ideally less than 100 milliseconds, roughly the amount of time when users start to notice delays).

H^k() (and other similar variants) are sometimes referred to as "slow hash functions", in that they serve the same purpose of H(), but take longer to do so.

===

WHAT DO STRONG PASSWORDS GET US?

Let us now consider what at first blush might seem like a silly question: do strong passwords really get us anything? Well, of course they must, right? They sure sound like the lynch pin of all security, based on how often we're expected to change them, how many special characters we're supposed to include, and so on.

To understand what kinds of attacks strong passwords get us, let us first decompose the attacks into a taxonomy along two dimensions:

- Is the attack:
  * ONLINE: the only way the attacker can try to guess the password is by issuing login attempts),

  * OFFLINE: the attacker can gain access to the database storing passwords (or coerce, bribe, or torture users).

- Does the attack target:
  * a SINGLE TARGETED USER: that is, the attacker has one user in mind and wants to recover his or her password, alone.

  * ANY user: here, the attacker does not care which user's password he obtains, so long as it is at least one.

Note that all OFFLINE attacks can be handled with our salted, repeated hash function strategy above. We actually assumed poor passwords (which is why we needed unique, random salts for each user), so using a strong password would not help us there.

Now let us consider an ONLINE attack targeting a single user. One common mechanism is to use a "THREE-STRIKES YOU'RE OUT" rule: if, within some given amount of time, a user fails to provide the correct password, then lock the user out from attempting other passwords for some amount of time. Do we need good passwords if we have such a rule in place? Let's consider an example:

- Suppose passwords are 6 digit numbers (poor passwords!)
- If a user issues 3 wrong login attempts within a 24 hour period, then the user is locked out of her account for 24 hours.

Suppose an attacker was given 10 YEARS of attempts at such a password. This leaves the attacker 3 * 365 * 10 attempts, or roughly 10^4 attempts. The chances that the attacker attempted the right password after 10 years is therefore

```
      10^4 / 10^6 = 1%
```

So, after ten YEARS of attempts to guess a 6-digit number, the attacker only had a 1% chance of getting it right. Strong passwords did not help here, either. Of course, there is a tradeoff here: it is also trivial to lock a user out of his or her account.

This leaves us with one remaining class of attack: ONLINE ATTACKS AGAINST ANY TARGET. If we use salted, repeated hash functions and the three-strikes you're out rule, then the attacker can still issue a small number of login attempts for EACH user in the system. If there are many users, and if users choose their passwords from a small set, then the attacker will be likely to recover at least one user's password.

So yes, strong passwords do matter, but they are not the sole solution to remaining resilient to attackers trying to recover victims' passwords.