Passwords[1]

Passwords are widely used for authentication, especially on the web. What practices should be used to make passwords as secure as possible?

1 Risks and weaknesses of passwords

Passwords have some well-known usability shortcomings. Security experts recommend that people pick long, strong passwords, but long random passwords are harder to remember. In practice, users are more likely to choose memorable passwords, which may be easier to guess. Also, rather than using a different, independently chosen password for each site, users often reuse passwords across multiple sites, for ease of memorization. This has security consequences as well.

From a security perspective, we can identify a number of security risks associated with password authentication:

- *Online guessing attacks.* An attacker could repeatedly try logging in with many different guesses at the user's password. If the user's password is easy to guess, such an attack might succeed.
- *Social engineering and phishing.* An attacker might be able to fool the user into revealing his/her password, e.g., on a phishing site. We've examined this topic previously, so we won't consider it further in these notes.
- *Eavesdropping.* Passwords are often sent in cleartext from the user to the website. If the attacker can eavesdrop (e.g., if the user is connecting to the Internet over an open Wifi network), and if the web connection is not encrypted, the attacker can learn the user's password.
- *Client-side malware.* If the user has a keylogger or other client-side malware on his/her machine, the keylogger/malware can capture the user's password and exfiltrate it to the attacker.
- *Server compromise.* If the server is compromised, an attacker may be able to learn the passwords of people who have accounts on that site. This may help the attacker break into their accounts on other sites.

We'll look at defenses and mitigations for each of these risks, below.

[1] Notes thanks to David Wagner

2 Mitigations for eavesdropping

There is a straightforward defense against eavesdropping: we can use SSL (also known as TLS). In other words, instead of connecting to the web site via http, the connection can be made over https. This will ensure that the username and password are sent over an encrypted channel, so an eavesdropper cannot learn the user's password.

Today, many sites do use SSL, but many do not.

Another possible defense would be to use more advanced cryptographic protocols. For instance, one could imagine a challenge-response protocol where the server sends your browser a random challenge r; then the browser takes the user's password w, computes $H(w, r)$ where H is a cryptographic hash (e.g., SHA256), and sends the result to the server. In this scheme, the user's password never leaves the browser and is never sent over the network, which defends against eavesdroppers. Such a scheme could be implemented today with Javascript on the login page, but it has little or no advantage over SSL (and it has some shortcomings compared to using SSL), so the standard defense is to simply use SSL.

3 Mitigations for client-side malware

It is very difficult to protect against client-side malware.

To defend against keyloggers, some people have proposed using randomized virtual keyboards: a keyboard is displayed on the screen, with the order of letters and numbers randomly permuted, and the user is asked to click on the characters of their password. This way, a keylogger (which only logs the key strokes you enter) would not learn your password. However, it is easy for malware to defeat this scheme: for instance, the malware could simply record the location of each mouse click and take a screen shot each time you click the mouse.

In practice, if you type your password into your computer and your computer has malware on it, then the attacker learns your password. It is hard to defend against this; passwords are fundamentally insecure in this threat model. The main defense is two-factor authentication, where we combine the password with some other form of authentication (e.g., a SMS sent to your phone).

4 Online guessing attacks

How easy are online guessing attacks? Researchers have studied the statistics of passwords as used in the field, and the results suggest that online guessing attacks are a realistic threat. According to one source, the five most commonly used passwords are `123456`, `password`, `12345678`, `qwerty`, `abc123`. Of course, a smart attacker will start by guessing the most likely possibilities for the password first before moving on to less likely possibilities. A careful measurement study found that with a dictionary of the 10 most common passwords, you can expect to find about 1% of users' passwords. In other words, about 1% of users choose a password from among the top 10 most commonly used passwords. It also found that, with a dictionary of the 2^{20} most commonly used passwords, you can expect to guess about 50% of users' passwords: about half of all users will have a password that is in that dictionary.

One implication is that, if there are no limits on how many guesses an attacker is allowed to make, an attacker can have a good chance of guessing a user's password correctly. We can distinguish targeted from untargeted attacks. A *targeted attack* is where the attacker has a particular target user in mind and wants to learn their password; an *untargeted attack* is where the attacker just wants to guess some user's password, but doesn't care which user gets hacked. An untargeted attack, for instance, might be relevant if the attacker wants to take over some existing Gmail account and send lots of spam from it.

The statistics above let us estimate the work an attacker would have to do in each of these attack settings. For an untargeted attack, the attacker might try 10 guesses at the password against each of a large list of accounts. The attacker can expect to have to try about 100 accounts, and thus make a total of about 1000 login attempts, to guess one user's password correctly. Since the process of guessing a password and seeing if it is correct can be automated, resistance against untargeted attacks is very low, given how users tend to choose their passwords in practice.

For a targeted attack, the attacker's workload has more variance. If the attacker is extremely lucky, he might succeed within the first 10 guesses (happens 1% of the time). If the attacker is mildly lucky, he might succeed after about one million guesses (happens half of the time). If the attacker is unlucky, it might take a lot more than one million guesses. If each attempt takes 1 second (to send the request to the server and wait for the response), making 2^{20} guesses will take about 11 days, and the attack is very noticeable (easily detectable by the server). So, targeted attacks are possible, but the attacker is not guaranteed a success, and it might take quite a few attempts.

5 Mitigations for online guessing attacks

Let's explore some possible mitigations for online guessing:

- *Rate-limiting.* We could impose a limit on the number of consecutive incorrect guesses

that can be made; if that limit is exceeded, the account is locked and the user must do something extra to log in (e.g., call up customer service). Or, we can impose a limit on the maximum guessing rate; if the number of incorrect guesses exceeds, say, 5 per hour, then we temporarily lock the account or impose a delay before the next attempt can be made.

Rate-limiting is a plausible defense against targeted attacks. It does have one potential disadvantage: it introduces the opportunity for denial-of-service attacks. If Mallory wants to cause Bob some grief, Mallory can make enough incorrect login attempts to cause Bob's account to be locked. In many settings, though, this denial-of-service risk is acceptable. For instance, if we can limit each account to 5 incorrect guesses per hour, making 2^{20} guesses would take at least 24 years—so at least half of our user population will become essentially immune to targeted attacks.

Unfortunately, rate-limiting is not an effective defense against untargeted attacks. An attacker who can make 5 guesses against each of 200 accounts (or 1 guess against each of 1000 accounts) can expect to break into at least one of them. Rate-limiting probably won't prevent the attacker from making 5 guesses (let alone 1 guess).

Even with all of these caveats, rate-limiting is probably a good idea. Unfortunately, one research study found that only about 20% of major web sites currently use rate-limiting.

- *CAPTCHAs.* Another approach could be to try to make it harder to perform *automated* online guessing attacks. For instance, if a login attempt for some user fails, the system could require that the next time you try to log into that same account, you have to solve a CAPTCHA. Thus, making n guesses at the password for a particular user would require solving $n-1$ CAPTCHAs. CAPTCHAs are designed to be solvable for humans but (we hope) not for computers, so we might hope that this would eliminate automated/scripted attacks.

  Unfortunately, this defense is not as strong as we might hope. There are black-market services which will solve CAPTCHAs for you. They even provide easy-to-use APIs and libraries so you can automate the process of getting the solution to the CAPTCHA. These services employ human workers in countries with low wages to solve the CAPTCHAs. The market rate is about \$1–2 per thousand CAPTCHAs solved, or about 0.1–0.2 cents per CAPTCHA solved. This does increase the cost of a targeted attack, but not beyond the realm of possibility.

  CAPTCHAs do not stop an untargeted attack. For instance, an attacker who makes one guess at each of 1000 accounts won't have to solve any CAPTCHAs. Or, if for some reason the attacker wants to make 10 guesses at each of 100 accounts, the attacker will only have to solve 900 CAPTCHAs, which will cost the attacker maybe a dollar or two: not very much.

- *Password requirements or nudges.* A site could also impose password requirements (e.g., your password must be 10 characters long and contain at least 1 number and 1 punctuation symbol). However, these requirements offer poor usability, are frustrating

for users, and may just tempt some users to evade or circumvent the restriction, thus not helping security. Therefore, I would be reluctant to recommend stringent password requirements, except possibly in special cases.

Another approach is to apply a gentle "nudge" rather than impose a hard requirement. For instance, studies have found that merely showing a password meter during account creation can help encourage people to choose longer and stronger passwords.

6 Mitigations for server compromise

The natural way to implement password authentication is for the website to store the passwords of all of its passwords in the clear, in its database. Unfortunately, this practice is bad for security. If the site gets hacked and the attacker downloads a copy of the database, then now all of the passwords are breached; recovery may be painful. Even worse, because users often reuse their passwords on multiple sites, such a security breach may now make it easier for the attacker to break into the user's accounts on other websites.

For these reasons, security experts recommend that sites avoid storing passwords in the clear. Unfortunately, sites don't always follow this advice. For example, in 2009, the Rockyou social network got hacked, and the hackers stole the passwords of all 32 million of their users and posted them on the Internet; not good. One study estimates that about 30–40% of sites still store passwords in the clear.

6.1 Password hashing

If storing passwords in the clear is not a good idea, what can we do that is better? One simple approach is to hash each password with a cryptographic hash function (say, SHA256), and store the hash value (not the password) in the database.

In more detail, when Alice creates her account and enters her password w, the system can hash w to get $H(w)$ and store $H(w)$ in the user database. When Alice returns and attempts to log in, she provides a password, say w'; the system can check whether this is correct by computing the hash $H(w')$ of w' and checking whether $H(w')$ matches what is in the user database.

Notice that the properties of cryptographic hash functions are very convenient for this application. Because cryptographic hash functions are one-way, it should be hard to recover the password w from the hash $H(w)$; so if there is a security breach and the attacker steals a copy of the database, no cleartext passwords are revealed, and it should be hard for the attacker to invert the hash and find the user's hashes. That's the idea, anyway.

Unfortunately, this simple idea has some shortcomings:

- *Offline password guessing.* Suppose that Mallory breaks into the website and steals a copy of the password database, so she now has the SHA256 hash of Bob's password.

This enables her to test guesses at Bob's password very quickly, on her own computer, without needing any further interaction with the website. In particular, given a guess g at the password, she can simply hash g to get $H(g)$ and then test whether $H(g)$ matches the password hash in the database. By using lists of common passwords, English words, passwords revealed in security breaches of sites who didn't use password hashing, and other techniques, one can generate many guesses. This is known as an *offline guessing attack*: offline, because Mallory doesn't need to interact with the website to test a guess at the password, but can check her guess entirely locally.

Unfortunately for us, a cryptographic hash function like SHA256 is very fast. This lets Mallory test many guesses rapidly. For instance, on modern hardware, it is possible to test something in the vicinity of 1 billion passwords per second (i.e., to compute about 1 billion SHA256 hashes per second). So, imagine that Mallory breaks into a site with 100 million users. Then, by testing 2^{20} guesses at each user's password, she can learn about half of those users' passwords. How long will this take? Well, Mallory will need to make 100 million $\times 2^{20}$ guesses, or a total of about 100 trillion guesses. At 1 billion guesses per second, that's about a day of computation. Ouch. In short, the hashing of the passwords helps some, but it didn't help nearly as much as we might have hoped.

- *Amortized guessing attacks.* Even worse, the attack above can be sped up dramatically by a more clever algorithm that avoids unnecessarily repeating work. Notice that we're going to try guessing the same 2^{20} plausible passwords against each of the users. And, notice that the password hash $H(w)$ doesn't depend upon the user: if Alice and Bob both have the same password, they'll end up with the same password hash.

  So, consider the following optimized algorithm for offline password guessing. We compute a list of 2^{20} pairs $(H(g), g)$, one for each of the 2^{20} most common passwords g, and sort this list by the hash value. Now, for each user in the user database, we check to see whether their password hash $H(w)$ is in the sorted list. If it is in the list, then we've immediately learned that user's password. Checking whether their password hash is in the sorted list can be done using binary search, so it can be done extremely efficiently (with about $\lg 2^{20} = 20$ random accesses into the sorted list). The attack requires computing 2^{20} hashes (which takes about one millisecond), sorting the list (which takes fractions of a second), and doing 100 million binary searches (which can probably be done in seconds or minutes, in total). This is *much* faster than the previous offline guessing attack, because we avoid repeated work: we only need to compute the hash of each candidate password once.

6.2 Password hashing, done right

With these shortcomings in mind, we can now identify a better way to store passwords on the server.

First, we can eliminate the amortized guessing attack by *incorporating randomness into the hashing process.* When we create a new account for some user, we pick a random *salt* s. The salt is a value whose only purpose is to be different for each user; it doesn't need to be secret.

The password hash for password w is $H(w, s)$. Notice that the password hash depends on the salt, so even if Alice and Bob share the same password w, they will likely end up with different hashes (Alice will have $H(w, s_A)$ and Bob $H(w, s_B)$, where most likely $s_A \neq s_B$). Also, to enable the server to authenticate each user in the future, the salt for each user is stored in the user database.

Instead of storing $H(w)$ in the database, we store $s, H(w, s)$ in the database, where s is a random salt. Notice that s is stored in cleartext, so if the attacker gets a copy of this database, the attacker will see the value of s. That's OK; the main point is that each user will have a different salt, so the attacker can no longer use the amortized guessing attack above. For instance, if the salt for Alice is s_A, the attacker can try guesses $g_1, g_2, \ldots, g_n$ at her password by computing $H(g_1, s_A), \ldots, H(g_n, s_A)$ and comparing each one against her password hash $H(w_A, s_A)$. But now when the attacker wants to guess Bob's password, he can't reuse any of that computation; he'll need to compute a new, different set of hashes, i.e., $H(g_1, s_B), \ldots, H(g_n, s_B)$, where s_B is the salt for Bob.

Salting is good, because it increases the attacker's workload to invert many password hashes. However, it is not enough. As the back-of-the-envelope calculation above illustrated, an attacker might still be able to try 2^{20} guesses at the password against each of 100 million users' password hashes in about a day. That's not enough to prevent attacks. For instance, when LinkedIn had a security breach that exposed the password hashes of all of their users, it was discovered that they were using SHA256, and consequently one researcher was able to recover 90% of their users' passwords in just 6 days. Not good.

So, the second improvement is to *use a slow hash*. The reason that offline password guessing is so efficient is because SHA256 is so fast. If we had a cryptographic hash that was very slow—say, it took 1 millisecond to compute—then offline password guessing would be much slower; an attacker could only try 1000 guesses at the password per second.

One way to take a fast hash function and make it slower is by iterating it. In other words, if H is a cryptographic hash function like SHA256, define the function F by

$$F(x) = H(H(H(\cdots(H(x))\cdots))),$$

where we have iteratively applied H n times. Now F is a good cryptographic hash function, and evaluating F will be n times slower than evaluating H. This gives us a tunable parameter that lets us choose just how slow we want the hash function to be.

Therefore, our final construction is to store $s, F(w, s)$ in the database, where s is a randomly chosen salt, and F is a slow hash constructed as above. In other words, we store

$$s, H(H(H(\cdots(H(w, s))\cdots)))$$

in the database.

How slow should the hash function F be? In other words, how should we choose n? On the one hand, for security, we'd like n to be as large as possible: the larger it is, the slower offline password guessing will be. On the other hand, we can't make it too large, because that will slow down the legitimate server: each time a user tries to log in, the server needs to evaluate

F on the password that was provided. With these two considerations, we can now choose the parameter n to provide as much security as possible while keeping the performance overhead of slow hashing down to something unnoticeable.

For instance, suppose we have a site that expects to see at most 10 logins per second (that would be a pretty high-traffic site). Then we could choose n so that evaluating F takes about one millisecond. Now the legitimate server can expect to spend 1% of its CPU power on performing password hashes—a small performance hit. The benefit is that, if the server should be compromised, offline password guessing attacks will take the attacker a lot longer. With the example parameters above, instead of taking 1 day to try 2^{20} candidate passwords against all 100 million users, it might take the attacker about 3000 machine-years. That's a real improvement.

In practice, there are several existing schemes for slow hashing that you can use: Scrypt, Bcrypt, or PBKDF2. They all use some variant of the "iterated hashing" trick mentioned above.

7 Implications for cryptography

The analysis above has implications for the use of human-memorable passwords or passphrases for cryptography.

Suppose we're building a file encryption tool. It is tempting to prompt the user to enter in a password w, hash it using a cryptographic hash function (e.g., SHA256), use $k = H(w)$ as a symmetric key, and encrypt the file under k. Unfortunately, this has poor security. An attacker could try the 2^{20} most common passwords, hash each one, try decrypting under that key, and see if the decryption looks plausibly like ciphertext. Since SHA256 is fast, this attack will be very fast, say one millisecond; and based upon the statistics mentioned above, this attack might succeed half of the time or so.

You can do a little bit better if you use a slow hash to generate the key instead of SHA256. Unfortunately, this isn't enough to get strong security. For example, suppose we use a slow hash tuned to take 1 millisecond to compute the hash function. Then the attacker can make 1000 guesses per second, and it'll take only about 15 minutes to try all 2^{20} most likely passwords; 15 minutes to have a 50% chance of breaking the crypto doesn't sound so hot.

The unavoidable conclusion is that deriving cryptographic keys from passwords, passphrases, or human-memorable secrets is usually not such a great idea. Password-based keys tend to have weak security, so they should be avoided whenever possible. Instead, it is better to use a truly random cryptographic key, e.g., a truly random 128-bit AES key, and find some way for the user to store it securely.

8 Alternatives to passwords

Finally, it is worth noting that there are many alternatives to passwords, for authenticating to a server. Some examples include:

- Two-factor authentication.
- One-time PINs (e.g., a single-use code sent via SMS to your phone, or a hardware device such as RSA SecurID).
- Public-key cryptography (e.g., SSH).
- Secure persistent cookies.

We most likely won't have time to discuss any of these further in this class, but they are worth knowing about, for situations where you need more security than passwords can provide.

9 Summary

The bottom line is: don't store passwords in the clear. Instead, sites should store passwords in hashed form, using a slow cryptographic hash function and a random salt. If the user's password is w, one can store

$$s, H(H(H(\cdots(H(w, s))\cdots)))$$

in the database, where s is a random salt chosen randomly for that user and H is a standard cryptographic hash function.