Passwords

Storing passwords

When you log onto a site, you send your password over to the server. How does the server verify your password is correct? One approach would be a simple string comparison with the password you sent and the one stored on the server. If the stored password is in plain text, that's a really bad idea. Any attacker that hacks into the system would be able to read your password. Any unscrupulous administrators with access to the system could do the same. An improvement would be to store an encrypted version of the password on the server. But that's still not very good. If an attacker breaks into the system, they could very likely find the decryption key and then get access to your password and everyone else's. Rogue administrators could do the same.

Instead, the current best practice is to *hash* the password. When you log on to a site, you send them your password. They hash the password and compare it to the stored hash of the password to see if it matches. Since hashes act like mostly unique fingerprints, this will work to make sure you can log in as you and no one else. Since only the hash is stored on the server, if an attacker gets access to the server, they will get the hashes only, not the plain text passwords. Because hashes are hard to reverse, they will have to do some work to recover the passwords from the hash, if they even can.

Cracking passwords

Here is the MD5 hash of the word "security": `e91e6348157868de9dd8b25c81aebfb9`. If you put that hash into a search engine, it will tell you right away that it's the MD5 hash of "security". How does that work? Even MD5 is secure in the sense that you couldn't look at that string of hex digits and figure out that it came from the word "security". Instead, we do a brute-force trial and error process. We could take all the words in the dictionary, hash them, and compare the hash to `e91e6348157868de9dd8b25c81aebfb9`. Once we get a word that hashes to that value, then we have "cracked" the hash. Here is some Python code to do just that. Assume that `wordlist.txt` contains a list of English words.

```
h = 'e91e6348157868de9dd8b25c81aebfb9'
for w in open('wordlist.txt'):
    if md5(w.strip().encode()).hexdigest() == h:
        print(w)
```

Here's another example of a basic brute-force attack that attempts to crack a password that consists of 3 lowercase letters. The nested loops generate all possible 3-letter strings, and the code in the loops compares the hashes of those strings to the given hash.

```
h = 'd16fb36f0911f878998c136191af705e'
alpha = 'abcdefghijklmnopqrstuvwxyz'
for c in alpha:
    for d in alpha:
        for e in alpha:
            if md5((c+d+e).encode()).hexdigest() == h:
                print(c+d+e)
```

It's possible to write similar code to attempt to crack passwords that fit all sorts of other types of patterns, like all passwords of 6 random keyboard characters, all passwords that consist of a name followed by two digits and a symbol, etc. And there are tools out there that will do this for you. Some popular ones are the website crackstation.net and the command-line tool hashcat. A highly optimized tool, such as hashcat, running on a fast GPU can easily search through tens of billions of MD5 OR SHA-256 hashes a second. A network of GPUs or even specialized hardware built for hashing things, can increase this into trillions per second or more.

This is why it is recommended not to use hash functions like MD5, SHA1, SHA2, or SHA3 for hashing passwords. They are just too fast. We want to slow down brute-force attempts. Some good hash functions for passwords include bcrypt, scrypt, and argon2. These are tunable so that you specify how long it would take to hash something. For instance, you can set them so that it takes 0.1 seconds to do a hash on typical hardware. So

instead of an attacker being able to try 10 billion hashes a second, they can only test 10. Of course, this also means more work for the actual server you're logging into, so you don't want to set the time too high.

Rainbow tables and salting

One way an attacker can speed up the brute-force search is to do it ahead of time. For instance, they can generate all strings of 1 to 6 lowercase letters, hash each, and store them in a big lookup table. Then, when they have a hash to crack, they just have to look it up in the table, which is really fast. This type of table is called a *rainbow table*. This is much faster than running a separate brute-force search for each password. Rainbow tables do take up a lot of space, so there are limitations to how big the table can be.

To stop rainbow tables, a simple technique called *salting* is used. The idea is not to store the hash of the password itself, but rather to generate a unique random number, combine that with the password, and then store the hash of the combination. That random number is called a *salt*, and it's stored together with the password hash. This makes a rainbow table useless because there is no way an attacker could take the salt into account when generating their rainbow table, and the salts are different for each password.

Calculations

If you're in charge of a site, you should make sure to use a proper password hashing function and not something like SHA-256 or MD5. But you can't depend on other sites to do that. So it's important to make sure passwords you use for other sites are resistant to brute-forcing. Let's see how resistant various types of passwords are to brute-forcing. We'll assume that an attacker can test 10 billion passwords (10^{10}) a second.

1. A password that is a single dictionary word — There are about 300,000 words in the dictionary. At 10 billion hashes per second, it would take at most $300000/10^{10} = .00003$ seconds. So this is not secure at all.
2. A 6-character password of random lowercase letters — There are 26^6 such passwords, and $26^6/10^{10} = .03$ seconds, so this is still not secure.
3. An 8-character password of random keyboard characters — There are about 96 characters we can easily type on a standard keyboard, and $96^8/10^{10} = 721390$ seconds, which is about 8.3 days.
4. A password consisting of four common English words followed by two digits and a special character from a list of 32 possible characters. Let's suppose there are about 30,000 common English words. Then the calculation would be $30000^4 \cdot 10^2 \cdot 32/10^{10} = 259200000000$ seconds, which is over 8200 years.

When deciding on a password, you would have to take into account the threat model. There are two types of attacks. The first is where an attacker gets the password hash file with many users' passwords from a server and tries to crack as many passwords as they can. For each individual password, they are probably not going to spend more than a short amount of time trying to crack it. Passwords like #1 and #2 above would be cracked in one of their first passes over the file. But something like the password in #3 above would probably be okay. If there are even 1000 strong passwords in that file, and each requires 8 days to crack, that would be 8000 days of work, which is more than most attackers would be willing to do.

However, if your threat model is an attacker particularly trying to hack you, and not just whatever easy-pickings they can get from a password file, then the password in #3 would not be okay anymore. Eight days is not that much time for someone who is really after you, and they could cut that time down further with good hardware. Something like the password in #4 is better, unless you have some really powerful actors after you, at which point you have bigger problems than just your password.

For extreme safety, you could go with a password of 20 random keyboard characters. That would be 96^{20} possibilities to check, which even at the crazy rate of 10 quadrillion hashes per second would take longer to crack than the estimated age of the universe. The downside is that 20 random characters are not that easy to remember.

Personally, I like to go with something like #4 above. Four words, a couple of digits, and a special character are pretty easy to remember and provide a decent amount of security against brute-forcing attacks. The major downside is it takes a while to type, especially on a device without a keyboard. The idea was inspired by the famous XKCD "correct horse battery staple" comic (https://xkcd.com/936/).

A 15-character password of random lowercase letters is easier to type and provides a similar level of security to #4 above. If the site requires special characters and digits, you could tack them onto the end. It might seem like 15 letters is hard to remember, but you would be surprised.

The NIST password recommendations

In 2017 the National Institute of Standards and Technology (NIST) published some updated recommendations dealing with passwords. Some are pretty obvious, but some are a bit controversial. Here are a few of the highlights:

1. Limit the number of login attempts people get — This is to stop people from continually trying passwords at a login to try to gain access as someone else.
2. Allow all Unicode characters — Some sites limit you in what types of characters you can use in a password. I guess those sites have technical reasons for the limitation, but for password security, the wider the range of characters allowed, the harder it will be for people to brute-force passwords.
3. No unreasonable maximum password sizes — Again, some sites have their own technical reasons for not wanting users to enter very long passwords, and of course allowing passwords that are millions of characters long would be a burden for servers, but reasonable maxes, like 64 characters, should be the norm.
4. Allow copy-paste — Some sites don't let you copy and paste a password into the login form. There are a few reasons for this, but the problem is that it makes it harder to use password managers. Password managers generate long, random passwords, and it's much easier to paste them into a form than to try to type them correctly.
5. Check passwords against a known list of insecure passwords — When users change their password, NIST recommends you make sure it's not on a list of common passwordss. There are many such lists out there, some with millions of passwords. Not a lot of sites had been doing this, but it's a good idea.
6. No password hints — Some sites allow users to enter a hint to help if they forget their passwords. Unfortunately people often put the password itself into the hint, or they do other things that make attackers' lives easier, like "hint: pet name + birthday".
7. No knowledge-based authentication — This is referring to security questions like "What is your mother's maiden name?" or "What was the name of your first pet?" An attacker can often figure out the answers to these with a little internet research or guesswork. Personally, if a site has these questions, I answer them with something bizarre. If the question is about where I went to high school, instead of answering "Chicago", I'll answer something like "sulfur dioxide". Some people answer these by mashing the keyboard, but you have to be careful about. An attacker trying to social-engineer a help-associate will tell the associate that they just mashed the keyboard, and the associate, trying to be helpful, will accept the answer.
8. No rules about passwords having to contain specific types of characters — Many sites require passwords to contain a lowercase letter, an uppercase letter, a digit, and a special character. These restrictions end up making passwords hard to remember, and people end up doing very predictable things like replacing the letter S with a dollar sign or always putting the digits at the end.

   These rules end up disallowing a strong password like `ouatuqzuuwznizx`, while allowing weak passwords like `Pa$$w0rd` or `Password123!`. NIST's solution is to get rid of rules like this, while checking potential passwords against a list of commonly-used passwords.

9. No unnecessary forced password resets — Many organizations require users to change their passwords every couple of months. In fact, it's one of the rules for PCI compliance. The problem is that it takes some effort to create a remember a strong password. If every site you use requires you to keep creating new passwords, eventually fatigue will set in, and you will fall into bad habits. A really common thing people do is to increment the number at the end of their password. If their initial password is "password', then after the next reset, they will use "password2" and then "password3" after that, etc.

   Of course, if a site is breached, then a password reset is a good idea. But requiring resets every few months is different. There's usually a tradeoff between usability and security. If you try to make things too secure, a side-effect will often be that things become harder to use. If they are too hard to use, people will look for ways around them, making the system less secure overall.

Password security tips

Reusing passwords across multiple sites If you use the same password on multiple sites, and one of those sites has a breach exposing your password, then attackers will often try the password they got at other common logins. For instance, if you use the same password for Gmail and some random online store, and that online store gets hacked, then attackers could get access to your Gmail. Once they have access to your Gmail, they could do password resets at other sites, like your bank or Amazon, and get the temporary passwords sent to your Gmail account (which they now control), and take over your account on those sites.

Even if you use a really strong password, if it turns out that you use that password on a site that stores passwords in plain text (and many sites do), then you're still vulnerable if you use that password across several sites.

Multifactor authentication The three common types or *factors* of authentication are described as below.

1. Something you know — examples include a password or an answer to a security question
2. Something you have — examples include a cell phone or hardware security module (HSM), like a YubiKey
3. Something you are — examples include biometrics like a fingerprint or face scan

Multifactor authentication is when you use more than one of these. The most common two-factor authentication is a password along with a code sent by text message to your phone. With the increase in SIM-swapping attacks, where attackers convince phone company reps to switch your SIM card over to the attacker, this is one of the weaker forms of multifactor authentication. But the general principle is that the more things an attacker needs in order to log in, the harder it will be.

Password managers These applications are used to generate and store secure passwords. Usually you need to create a single strong password to access the password manager, and it handles the rest of the details. It will generate long, random passwords and remember which sites they go with. A lot of people recommend password managers, though you do have to trust that the password manager itself and their security practices.

Writing passwords down on paper It's often recommended that you don't do this. In certain situations, it's a really bad idea, like taping your work password to your computer monitor at work. But if you're writing down passwords to keep somewhere safe where you live, and you trust the people you live with, then it's better than using weak passwords that are easy to remember.