Making Password Calculator Guessing Attacks Impractical

Alan H. Karp

Abstract

Password managers that calculate your passwords have some advantages over those that store them. but they are vulnerable to offline guessing attacks. One mitigation is to force the adversary to try so many guesses online that these attacks are impractical.

Introduction

There are two kinds of password managers. One kind stores your passwords in an encrypted vault; the other calculates each one when needed. Both protect your passwords with a secret, your *super password*. The former are more popular, perhaps because password calculators are vulnerable to an offline guessing attack.

A guessing attack to determine your super password starts with an adversary learning the generated password and other inputs to the calculation for a single site. Perhaps your site password and userid were exposed in a breach. Once the adversary has found a super password that results in the known site password, the adversary can then generate all your passwords. The question is how to make this attack impractical.

Guessing Attack Mitigations

The most common algorithm for computing passwords uses a hash function. In the simplest form, your super password is hashed with something specific to the site, such as the domain name, app name, user id, a user selected nickname, or a combination of them. The result of the hash is converted to a string that is used as the site password.

There are several design decisions. Which hash function? How many bits in the result? How to convert the bits of the hash to characters? What if the hash doesn't produce a string that matches the site's password rules? Most importantly, are there answers to these questions that make an offline guessing attack impractical?

The simplest way to make the adversary's job harder is to use a strong super password. A good, 20-character super password takes centuries to guess even at a billion guesses per second. Unfortunately, many people choose weaker ones. A determined adversary might be able to guess one of them in a a few minutes to hours. It is important to protect users who pick weaker super passwords.

Another mitigation is to use a different super password for different types of sites. For example, use one super password for banking and health accounts, another for subscriptions, and a third for all others. A bank is less likely to suffer a breach than the site for an online magazine. Guessing such a super password doesn't give the adversary access to your money. This approach is adds friction to logging in, so it might not be used.

A third mitigation is to use a slow hash function. If each guess takes longer, then the adversary is able to try fewer guesses in a given time, which can take the time to guess your super password from minutes to years. Unfortunately, users don't want to wait to get their passwords, limiting how slow the hash function can be.

A fourth mitigation was used by PassPet. When you first enter a super password on a given machine, PassPet recursively hashes it for several seconds and caches the result on the machine. When you ask for a password, PassPet hashes the cached value with the other inputs to the calculation, which is fast. An adversary trying to guess your super password would have to do the same amount of work needed to create the cached value for every guess. The problem is knowing which master password to compute with if you have more than one. It also gives an adversary who learns the cached value another vehicle for guessing your super password.

Yet another mitigation is to use a second factor in the computation. In this case, learning your super password does not give the adversary the ability to compute all your site passwords. One issue with this approach is incorporating it into different implementations of a password calculator. For example, those provided as a browser add-on often have a web page to use when the add-on isn't available. Another is finding a second factor that can be used with a variety of devices. Finally, the low uptake of 2-factor authentication shows that users don't like it.[1]

The mitigation described next is to use an algorithm that creates a lot of *pre-images*, guesses that produce the correct site password for one set of inputs but do so with low probability for any other inputs. The adversary will have to try each such pre-image online to see if the guess is your actual super password. These online tests are slow and can often be detected as a guessing attack by the site. Forcing a large enough number of online tests makes offline guessing attacks impractical.

Forcing Online Tests

The password generation algorithm takes a super password s that is as guessable as a random set of S bits and a site specific, public[2] salt n_i to produce a password p_i for site i having P_i bits, where $p_i=F(h(s,n_i))$. Here h is an H-bit hash function, and F is a function

[1] "Why is Daddy crying?" "Two-factor authentication."

[2] Not all the inputs may be public, *e.g.,* a user selected nickname for the site, but the salt is probably easy to guess.

that converts the bits output by the hash function into a password acceptable to the site.

Assume the adversary learns n_i and p_i for site *i,* perhaps from a breach. The adversary can then start guessing values s_j to see if $F(h(s_j,n_i))=p_i$. If there is only one s_j that produces p_i, the adversary can then generate passwords for all your sites. If more than one guess produces p_i, the adversary must test the guess online at least one of your other sites. This candidate is either *s* or a pre-image. The guessing attack becomes impractical if the probability the guess is *s* is sufficiently small even if it doesn't take long to find pre-images.

The adversary can weaken any guessing protection by filtering the pre-images. Most of the pre-images will be effectively random strings. Human memorable strings typically contain one or more dictionary words or variants, making them distinguishable from most pre-images. Between 1:1,000 to 1:10,000 guesses will contain a dictionary word for reasonable assumptions about the dictionary and super password size. Therefore, we need to choose parameters that generate orders of magnitude more pre-images.

Pre-images are generated when $S>P$, where *S* and *P* are the entropies in bits of *s* and p_i, respectively.[3] The number of guesses for *s* that generate p_i is 2^{S-P} for $S-P\geq 0$, and *1* otherwise. Unfortunately, making $S-P$ large is hard because *S* is limited by the user's ability to remember *s*, and *P* needs to be large enough to make p_i hard to guess. Nevertheless, there is a reasonable range of parameters for which a guessing attack is impractical.

An *L*-character p_i computed from a good hash function has entropy *P*. Since *s* must be memorable, its entropy will be less than that of a random string of the same length. If *K* is the length of a random string with entropy *S*, then the number of pre-images is $2^{a(K-L)}$, where $a = log_2 |A|$, and $|A|$ is the number of characters in the alphabet.

Consider some examples with a 64-character alphabet. With a good 20-character super password and a strong site password of 16 characters, there will be 2^{24} pre-images, which is enough pre-images for adequate guessing protection even after filtering. Shorter super passwords don't get this much protection unless the site password is weaker. A 16-character super password would be adequately safe from an offline guessing attack with a 12-character site password, which is moderately secure when stored using a slow hash function. A 12-character super password would only be safe from a guessing attack when the site password is very weak.

So far the salt has been assumed to be public, but that's not entirely the case. In some password managers, it consists of two terms, your userid for the site and your easy-to-remember nickname for the site with a separator, *e.g.,* the tab character, between them. The nickname is supposed to be easy to remember, *e.g.*, amazon, which makes

[3] Ignoring the fact that different sites may have different password strengths for simplicity.

it easy to guess, but the adversary doesn't know if it might be Amazon, doubling the number of pre-images. Other sites, *e.g.*, bestbuy, best buy, Best Buy, BestBuy, provide even more protection. Appending the same, easy to remember string, such as your birth year, to every nickname increases the number of pre-images enough to counter the adversary's ability to filter guesses. In other words, $K\text{-}L = 2$ gives adequate protection from offline guessing.

There is another source of pre-images. The computed site password may not meet the rules for the site. For example, it may start with a number, which the site won't accept. One strategy for finding a valid password is to use the computed candidate as input to a fast hash function, the output of which is the next candidate for p_i. This process is iterated until an acceptable password is found or some condition is met to give up trying.

If a guess generates a password that is not acceptable to the site, the adversary will have to iterate the fast hash until it returns one that does. If that result matches the known site password, the guess may be a pre-image. In other words, an initial hash result that looks nothing like the known site password was generated by a guess that is actually a pre-image.

This protection is limited because simple passwords that require only numbers and letters usually don't require any extra iterations. Those that require special characters typically need more iterations, offering stronger protection from offline guessing attacks.

An adversary who controls a site can reduce the number of required online tests by setting the password rules to require a very long site password, resulting in a small or even negative value for $S\text{-}P$. In this case, the adversary controls P while S is limited by the user's ability to remember s. The risk to the adversary is that users might become suspicious if a site with no obvious security requirements asks for a very long password.

Note that an adversary can filter almost all guesses by learning the inputs and site password for a second site. The very fact that makes guessing attacks infeasible if the adversary has information for one site allows an adversary who has that information for more than one to almost completely avoid online tests.

Users with weaker master passwords should change them as soon as they find out that their login information at a site was compromised. Unfortunately, they will then need to change passwords everywhere they log in, which is what users must do if the site that stores their passwords is compromised.

Summary

Password calculators have a long history, but the most successful password managers store users' passwords rather than calculating them. One reason is the risk of an

offline guessing attack against password calculators. Choosing an algorithm that generates a large number of pre-images makes this attack impractical.

Appendix

Other considerations that influence the algorithm are discussed here.

Selecting the Hash Function

An algorithmic mitigation to guessing attacks is to use a slow hash function that makes each guess take longer. One commonly used approach is to repeat a fast hash, such as SHA-256, many times. Unfortunately, that weakens the hash in the sense that there will be many guesses that produce the right passwords for multiple sites. "Nevertheless, standard cryptographic functions such as SHA256 are believed to be sufficiently secure for as many as 1M iterations."[4]

The Password-Based Key Derivation Function (PBKDF) doesn't suffer from this weakness. It is commonly used in authentication systems that compare the hash of a user's submitted password to what is stored in a database. It is also useful as a password calculation algorithm because is allows choosing both the number of bits in the output, called the *key*, and the number of iterations. These values are typically chosen to balance the amount of stored data with the application's latency requirements.

Another reason to pick PBKDF is that all major browsers support it with the built-in function PBKDF2. Using this function means that each iteration takes a time closer to the best the adversary can achieve than using a JavaScript implementation of another hash function.

One criticism of PBKDF is that it is not memory intensive, allowing an adversary to run many independent guessing attacks on a single machine. There are alternatives, scrypt which is more memory intensive, and Argon2 which is also GPU resistant. However, neither is built into browsers.

Converting the Key to Characters

There are a few ways to generate a password from the hash. The most commonly used is base64, but the converted result may not contain a special character from the site's required set. Another is to create an array of valid characters, convert the first bits of the hash into an index into that array, and use that character as the first

[4] https://toc.cryptobook.us/book.pdf, Section 14.3.

character of the password. Take the next few bits and repeat until the password is complete.

Including only the characters the site accepts in the array increases the probability that the password will be acceptable. For example, if the site requires a special character from the set %$@, then include only them in the array. Of course the probability of selecting one of them is only a tenth the probability of selecting a lower case letter. As a result, some of the algorithms described below will be less likely to find a password that requires a lot of special characters. A solution is to put several copies of these characters into the array. Any advantage an adversary gains in guessing the site password by knowing that was done can be mitigated by making the site password one character longer.

The simplest way to index into the character array is to use a byte of the key per character. Alternatively, the index could take 6 or 7 bits of the key, since the character array typically is of length 62-72. The latter approach gives more characters for the same number of bits, but the index computation is more complicated.

There are two ways to use a full byte of the hash to select a character. One alternative is to repeat the characters in the array until it has 256 characters so that all 8 bits of the byte are used as an index. Of course, some characters may appear more times than others, which makes site passwords slightly easier to guess. For example, the first 8 characters of a 62-bit alphabet will appear one more time than the others. This selection bias can be compensated for by making the site password one character longer.

An alternative is to use b mod $|A|$ as the index into the character array, where b is the byte's value, and $|A|$ is the alphabet size. With this approach, all characters in the array have the same probability of being selected, but not all bits in b are used. For example, only 6 of the 8 bits are used with a 64-character alphabet.

Computing an Acceptable Password

Some sites have complex password rules, specifying how many upper/lower case letters, numbers, and special characters your password must contain. Some also specify the set of allowed special characters. A password derived from a hash may not meet the site's requirements.

There are several ways to deal with a computed L-character password that doesn't meet the requirements. All are based on the premise that the bits produced by the hash function are indistinguishable from random.

1. Rotate the array of characters by one position and try again until an acceptable password is found or the array is back to where it started.

2. Convert more bits of the hash to characters than needed for the password, and select the first L as the candidate password. If it doesn't pass, drop the first of those characters and try again until an acceptable password is found or the characters have all been used. The number of characters produced from the hash is limited by the required latency.
3. Keep computing additional hashes until an acceptable password is generated. These should be fast hashes in order to increase the chance of finding a valid password within the required latency. Stop if one hasn't been found in time.
4. Insert characters of the the required types at the beginning of the output and fill the rest with characters selected from the alphabet. Then shuffle the position of the characters based on bytes from the hash to make the site password less guessable. The bytes used by the shuffle can be the same ones used to select the characters. Unlike the previous algorithms, this one is guaranteed to find an acceptable password if one exists, *e.g.*, the total number of required characters doesn't exceed the password length
5. Use the bytes of the hash to both select a random character from the allowed set and another byte to select a random place to put it in the site password. Like the previous one, this one is virtually guaranteed to find an acceptable password if one exists, *e.g.*, the total number of required characters doesn't exceed the password length. This method exposes more bits of the hash to the adversary, weakening the *S-P* factor.

These approaches have been tested and are able to generate an acceptable password with reasonable probability in even extreme cases, such as requiring one number, one upper case letter, one lower case letter, and 5 special characters for a 12-character site password.

All of these approaches affect the hash parameters, but only the last makes the adversary's job easier. The adversary sees only the bytes in the computed password for the first four. The last uses many more bytes since the random location might already be populated. Experiments show it uses 3-5 additional bytes per character in the calculated password, giving the adversary many more chances to reject a guess without testing online.

The possibility that the password derived from the initial hash doesn't meet the site's rules, which results in more pre-images. favors using one of the first three options The ability to always construct an acceptable password favors the last two.