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 the user's passwords with a secret, the user's *super password*. The former are more popular, perhaps because password calculators are vulnerable to an offline guessing attack.

A guessing attack to determine a password calculator user's super password starts with an adversary learning the generated password and other inputs to the calculation for a single site. The easiest way to get this information is to induce the user to create an account at a site controlled by the adversary. Once the adversary has found the super password that results in the known site password, the adversary can then generate all the user's 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, the user's 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 can guess 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 user is likely to choose the last of these for the adversary's site. Guessing this super password doesn't give the adversary access to the user's 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 a 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 a user first enters a super password on a given machine, PassPet recursively hashes it for several seconds and caches the result on the machine. When the user asks for a password, PassPet hashes the cached value with the other inputs to the calculation, which is fast. An adversary trying to guess a 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 the user has more than one. It also give an adversary who learns the cached value another vehicle for guessing the user's super password.

Yet another mitigation is MFKDF, which uses a second factor in the computation. In this case, learning a super password does not give the adversary the ability to compute the user's site passwords. One issue with this approach is incorporating it into different implementations of a password calculator. For example, those that are provided as a browser add-on often provide 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 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 the user's 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 salt n_i to produce a password p_i for site i having P_i bits, where $p_{ii}=F(h(s,n_i))$. Here h is an H-bit hash function, and F is a function

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

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 by inducing the user to create an account at a site controlled by the adversary. 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 the user's sites. If there is more than one guess that produces p_i, the adversary must test the guesses online at least one of the user's other sites.

Assume that the offline guessing attack will generate a candidate in a reasonable amount of time because people have trouble remembering hard to guess super passwords. This candidate is either *s* or a pre-image. The guessing attack becomes impractical if the probability the guess is *s* is sufficiently small.

The simplest way to generate pre-images is with $S>P$. Unfortunately, the adversary can make $S-P$ small by setting the password rules at the adversary's site to require a large *P*. In this case, the adversary controls *P* while *S* is limited by the user's ability to remember *s*. The password manager could limit the values of *P* that it will generate passwords for, but then the site passwords will be short and are likely to be easy to guess.

A better approach is $H>P$. There will be 2^{H-P} guesses that produce p_i, only one of which is *s.*[2] The rest are pre-images. Therefore, we want $H-P>>1$, which is easy to do. The adversary can choose a large *P* in an attempt to make $H-P$ small, but the calculation algorithm can choose *H* large enough that it can refuse to produce a password larger than some limit without making the passwords generated for legitimate sites too weak. Limiting the site password to 100 characters is not unreasonable.

The adversary can weaken this 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, making them distinguishable from most pre-images. Increasing $H-P$ protects against this attack, because it results in more pre-images and more that contain a dictionary word.

Consider two examples. The probability that a 20-character random string chosen from an alphabet of 70 characters contains any word of length between 5 and 10 characters out of a 100,000 word dictionary is 0.001.[3] An algorithm with 1B pre-images ($H-P=30$) means the adversary has to try 1M of them online. Shorter super passwords have a lower probability of containing a dictionary word, 1 in 1M for a 12-

[2] That's not quite true. Not every bit pattern is a valid character. If the super password alphabet has 64 characters, the exponent is $3(H-P)/4$, a difference small enough to ignore.

[3] This estimate is conservative, since password guessers typically use much larger dictionaries.

character super password, but there are many more pre-images (*H-P=50*). The likelihood of success is so low that there is no point in mounting an offline guessing attack.

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.

One criticism of PBKDF is that it is not memory intensive, allowing an adversary to run many independent guessing attacks on a single machine. Choosing one iteration with the largest possible key size mitigates this criticism. For example, computing a 2 Kb key with 50,000 iterations takes about 150 ms on a standard laptop. Doing 1 iteration to produce a 64 Mb key takes about the same amount of time while using much more memory.

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

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

the best the adversary can achieve than using a JavaScript implementation of another hash function.

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 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 at a time. 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 bits are used with a 64-character alphabet. In this example, there are 4 different byte values that produce the same character instead of one. In other words, there are 2^{2L} times as many guesses that will produce the same L-character site password. Using more bits of the hash for the index dramatically increase this factor, *e.g.*, using 16 bits results in a factor of 2^{10L} more pre-images.

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. Many 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 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. 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. Unlike the other algorithms, 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.

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 three. 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 introduces additional pre-images. If the rules state that the password must start with a letter, the generated password could be yOmLOM3cfjlh. This value could have been the first candidate password for an incorrect guess of the super password. What the adversary doesn't know is that the first candidate password was actually 4NaxjmnuSVEO. In other words, the generated password came from a number of iterations unknown to the adversary. The adversary has to try online all such combinations that match the known site password to know if a guess is the correct super password.[5]

[5] This argument doesn't apply to the 4th method, another reason not to use it.