Making Password Calculator Guessing Attacks Impractical

Alan H. Karp

July 8, 2024

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.

1 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.

An offline 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 from a breach. Once the adversary has found a super password that results in the known site password, the adversary can then generate the password for another site. If that password works, there is a high probability that the adversary can generate all your passwords. The question is how to make this attack impractical.

2 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 your subscription super password doesn't give the adversary access to your money. This approach adds friction to logging in, so it might not be used. There is also a risk that users will pick super passwords that allow an adversary to figure out the others after learning any one of them.

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 moderately weak 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 *collisions*, 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 collision 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.

3 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 $p_i = F(h(s, n_i)$. Here h is an H-bit hash function, and F is a function 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

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

least one of your other sites. This candidate is either s or a collision. The guessing attack becomes impractical if the probability the guess is s is sufficiently small even if it doesn?t take long to find each collision.

Collisions are generated when $S > P$, where S and P are the entropies in bits of s and p_i, respectively. The number of guesses for s that generate p_i is 2^{S-P} for $S - P > 0$, 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 typically has entropy $|A|^L$, where $|A|$ is the number of characters in the alphabet. Since s must be memorable, its entropy will be less than that of a random string of the same length. If a user generated string has the entropy of a K-character random string, then the number of collisions is $2^{a(K-L)}$, where $a = log_2|A|$.

Consider some examples with a 64-character alphabet. With a good super password having entropy $K = 16$ characters (96 bits) and a strong site password with $L = 12$ (72 bits), there will be 2^{24} collisions, which is more than enough collisions for adequate guessing protection. Weaker super passwords don't get this much protection unless the site password is also weaker. A super password with $K = 12$ would be adequately safe from an offline guessing attack with a 10-character site password, which itself is moderately secure against guessing when stored using a slow hash function but not otherwise. A shorter super password would only be safe from a guessing attack when the site password is very weak.

There is no need for the adversary to try all possible strings. Most of the collisions will be effectively random strings, but the adversary knows that human memorable strings typically contain one or more dictionary words or variants, such as capitalization, reversals, and LEET-speak substitutions. Measurements using zxcvbn show that about 7% (13%) of 12(24)-character random strings contain at least one substring that a person might choose. Therefore, we need to choose parameters that generate about 10 times as many collisions than it first appears.

So far the salt has been assumed to be public, but that's not entirely the case. In some password calculators, it consists of two terms, your userid for the site and an 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 it easy to guess. Unfortunately, usability constraints require that the nickname be folded to lower case before using it in the calculation, but adding a fixed string, such as your birth year, to every nickname makes the adversary's job harder.

There is another source of collisions. 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 collision. In other words, an initial hash result that looks nothing like the known site password was generated by a guess that is actually a collision. This protection is limited because fewer than 10% of computed passwords fail to satisfy the rules for most sites.

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 $S - P$. 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 super 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.

4 Conclusions

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. Making sure the calculation generates a large number of collisions makes this attack impractical.

Appendices

Other considerations that influence the algorithm are discussed here.

A 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."[2]

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.

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

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, such as scrypt which is more memory intensive, and Argon2 which is also GPU resistant. However, neither is built into browsers.

B Converting the Key to Characters

There are a few ways to generate a password from the hash. A convenient option 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 those characters 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 one 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 compute $R/|A|$, where R is the range of the random values, $R = 256$ when using a byte, and $|A|$ is the number of character in the alphabet. Another 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-character alphabet will appear one more time than the others. The resulting modulo bias is the same for both approaches. The document ModuloBias shows that the loss of entropy due to this factor is small enough to ignore..

C 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 bytes of the hash to characters than needed for the password, and convert the first L bytes into the candidate password. If it doesn't pass, drop the first of those bytes 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 a time limited by the latency requirement.
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. 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. There are four variants.
   (a) Use the same bytes for both selecting characters and for shuffling. This approach makes the site password more guessable since the adversary knows the random numbers used for the shuffle.
   (b) Use one additional byte from the hash per character for shuffling.
   (c) Use one additional byte from the hash as the seed for a pseudo-random number generator.
   (d) Use one byte 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. Select another byte if that location is occupied. Experiments show it uses 3-5 additional bytes per character in the calculated password. As a result, the hash must return more bytes than may be needed, reducing the number of iterations an adversary needs to do.

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.

The possibility that the password derived from the initial hash doesn't meet the site's rules, which results in more collisions, favors using one of the first three options The ability to always construct an acceptable site password favors one of the variants of the last one. A compromise solution is to use 4(a) only when 3 fails to produce an acceptable password, which protects your super password at the cost of making the site password more guessable. Note that 3 has never failed in over 10 years' use and testing on 100s of additional sites.

D Further Discussion of 4

Consider how approach 4 works for the case where the password rules require one each of upper case, lower case, digit, and special character. These will appear in the first four positions before the shuffle. The adversary knows that the first character was selected from a set of 52; the second, a set of 26; the third a set of 10; and the fourth from a set specified in the password rules, say 8. In other words, about 100 times fewer guesses are needed if the adversary can un-shuffle the characters. That task is simplified in 4a because the adversary knows the bytes to use for the shuffle.

Approach 4b uses additional bytes from the hash to do the shuffle, either one byte as the seed for a pseudo-random number generator or one additional byte per character in the site password. In either case, the adversary needs fewer online tests because more bytes go into computing the site password. To see this assume that a guess has all the characters that appear in a known site password but not necessarily in the right order. In addition, assume the guess has produced a site password structured as it would be before the shuffle. The adversary then uses the next byte(s) in the hash to shuffle the characters, and rejects the guess if the result doesn't match the known site password. As a result, the exponent of the number of collisions that must be tested online is reduced from $S - P$ to $S - P - 8N$, where $N = L$ if a byte per character is used for the shuffle or $N = 1$ if one byte is used as a random number seed.

E Strength Meters

Users are more likely to pick strong super passwords if there is a meter to warn them of weak ones. The strength estimator of zxcvbn provides one that looks for a number of patterns that can reduce the number of tries to guess the password. It falls back on brute force for characters that aren't in any other pattern and uses $10M$, where M is the number of character in the brute force segment. The rationale is that there are words that don't appear in standard dictionaries but might show up in others. Their paper gives Teiubesc, which means ”I love you” in Romanian as an example. Better to be conservative they say.

This value is supported by the work of Shannon, who estimated that there are between 1 and 2 bits of entropy per character in English text. That value is certainly an underestimate for someone constructing a password. It is more likely that there are at least 4 bits of entropy per character in a human chosen password. Using 4 instead of $log_2|A|$ for the substrings for which zxcvbn uses brute force is likely to dramatically underestimate the password's strength.

The problem with using the more conservative value for the site password is that it encourages people to use much longer site passwords than they need. As a result, the entropy difference between it and the super password is likely to be small or even negative, resulting in offline guessing attacks being tractable.

Using a conservative estimate makes perfect sense for the super password, which is picked by a person and matches the zxcvbn use case, but not the site password,which is calculated with a good hash function. As shown earlier, the likelihood of generating guessablesubstrings is small but non-zero, hence the need to have a meter. Calculating the zxcvbn score with $log_2|A|$ for the brute force substrings is more appropriate for the site password. There is still the chance to produce words like Teiubesc, but that chance is vanishingly small compared to the probability of a person choosing it.

One way to encourage people to pick super passwords that have more entropy than their site passwords is to report that the super password has less entropy than the modified zxcvbn calculation. A simple heuristic that can encourage choosing super passwords for which offline guessing is impractical is to reduce the calculated strength by some number of bits. In this case, the super password strength meter will report Strong only when the super password has approximately that many more bits of entropy than a site password that has just enough entropy for its meter to report Strong. Decreasing the calculated entropy by 16 bits will result in enough collisions to make offline guessing impractical.

F A Key Assumption

Many of the arguments made in this document hinge on the assumption that the adversary needs fewer online tests the more bytes go into creating the site password. Given an alphabet of size A and random values in the range $0 \leq r < R$, the number of values of r that produce the same value of $\mathrm{mod}(R, A)$ is either $\lfloor \mathrm{mod}(R, A) \rfloor$ or $\lceil \mathrm{mod}(R, A) \rceil$. Say that the adversary knows the first character of a site password, say a, A guess for the super password produces bytes with a value of $0 \leq B < R$.