ABSTRACT

In the past few years, security breaches of the networks of large online companies have exposed millions of user passwords. Though typically stored in an incomprehensible ‘hashed’ format, researchers (and bad actors) can recover the original, plaintext passwords from leaks by applying well-known cracking techniques. When this occurs, the original, single-company data breach increases in scope; no longer are accounts exclusively relevant to the source company insecure, but since users frequently use the same email/password combinations on multiple websites, attackers are able to gain access to accounts on services not directly affected by a security inconsistency. Such spillover effects can affect any website that uses a user account system.

Generally, the companies exploited have employed either insufficient or imperfect protections of user data, either by storing passwords in plaintext, using hashing algorithms not meant for password storage, or by applying poorly-selected encryption to passwords.

In this review, we introduce the question of user authentication and discuss common password storage techniques. An examination of common hashing algorithms used in password storage follows. We close with possible mitigation measures users can utilize in order to minimize the effect a data breach has on their online presence as a whole.

INTRODUCTION

When a company decides it wants to provide more user customization on a website, it has a few options. One is to use cookies, which can track user behavior over different browsing sessions and can store information over extended periods of time. However, cookies are ephemeral (clients can erase them), unreliable (they do not persist over different browsers), and cannot be trusted (they are stored client-side; a user can modify cookies to whatever they want). Even with these drawbacks, cookies are a popular way to provide simple customization on a website.

However, when an online property wants a more robust system with many capabilities, they turn to the concept of users. A user can then customize a website according to his or her pleasure, and data between different users is separated for privacy. The question then becomes, “How can I identify and authenticate an individual user?”

Overwhelmingly, website operators have chosen to utilize the concept of a password associated with a user account or email address. This is works reasonably well at keeping out unwanted users and validating legitimate ones. Over time, however, numerous drawbacks to this system have been discovered.

One such problem is user password memory. If there are only a handful of passwords a user is expected to remember, this is tenable. But as websites become more numerous and users gain in the amount of accounts they must maintain, it becomes difficult to keep track of the different username/password combinations a person has used. Users counter this by re-using passwords across services. This is a legitimate approach, but becomes problematic if an account on any service is compromised; now, all
accounts on services that share credentials with the initial breach can be accessed.

Additionally, passwords that are easy to remember are generally weak passwords. Passwords based on English words, words relating to the specific website in question, and proper nouns are particularly poor choices. Length also has much to do with the strength of a password, and longer passwords are more difficult to remember. In the end, the only true ‘secure’ password is a long, randomly-generated string of characters. These types of passwords are the most difficult to remember.

Another problem deals with password storage and validation. How should a website operator validate a user when logging in? Generally, this is done through some comparison of the given password to the password that is found in the database (which was saved at account creation). Unfortunately, storing passwords in plaintext is not good security practice, so methods have been developed that try to mitigate the impact of a password breach. The different options available and the ones companies ultimately choose are detailed here.

**PASSWORD STORAGE**

There are many ways to store passwords in an online database. The leaks provided here exemplify typical storage schemes companies utilize. Unfortunately, none of them are considered safe and none conform to currently-accepted best practices. Below, an examination of the various techniques is given.

**PLAINTEXT**

Storage of passwords in plaintext is exactly what it sounds like – the password for each user is stored in the database with no encryption, hashing, or transformations of any kind. This means that if the database is ever leaked, the passwords will be exposed without any additional work. Additionally, anyone that has access to the database can see what password each individual user has selected for this site. In a system that adheres to best practices, this should not be possible.

Validation is performed with a simple comparison of the attempted string with the stored password string. If they match, the user is authenticated. This is by far the worst option for storing passwords. No company should be using this storage scheme because it puts users at a high risk and forces them not only to trust the company’s database security but also the database administrators themselves. Since the passwords are viewable by anyone with access to the database tables, a rogue administrator could steal passwords and misuse them.

**HASHED PASSWORDS**

In this storage scheme, a password is put through a one-way hash function before being stored in the database. In general, a hash function is an algorithm that maps data of a variable length to data of a fixed length (“Cryptographic hash function,” 2013). Different inputs to the hash function should always produce differing hash values, and
the same input to the hash function should consistently produce the same value.

Hash functions are frequently used for simple data verification. For example, when a user is downloading a file from a server, the website administrator will provide a hash value for that file. When the download is complete, the user can run the local file through the same hash function to see if it produces a value that matches the one posted. If the values match, then the download completed successfully and no modification of the file has been detected. However, if the values do not match, the user has downloaded a file that is in some way different from the one that the website administrator has posted. This could mean that the download did not complete successfully, or, if it did, that the website has been compromised and the true file has been replaced by an alternative one.

This same concept applied to passwords forms the next level of password storage. There are many different hashing functions, and selection of a correct hashing function is critical to the secure storage of passwords.

Typically, hash functions are designed to be as fast and simple as possible – since they are used for things such as data validation (as in the example above), a faster algorithm means less resource usage and a better user experience. Functions of this nature include MD5 and SHA-1, two common algorithms that are used for storage of hashed passwords.

Unfortunately, when applied to password hashing, the algorithm that is chosen should be one designed to be slow, not fast. The reason has to do with what happens when a data breach does occur. When a cracker gets access to a list of millions of hashed user passwords, the next goal is to recover the plaintext version of these passwords. However, hash functions are meant to be one-way; their output does not suggest anything about the original input. This means that the attacker must continually “guess” various passwords, running them through the hash function each time and then comparing the value to the list of hashed passwords. If two hashes match up, the plaintext version of the password corresponds to the one the attacker just used to create this hash value. Additionally, since the same string will always hash to the same value, users with the same password hash will also have the same originating password. Through billions of attempts and comparisons, an attacker can eventually expect to recover 80-95% of such a list, depending on how persistent the attack is and what password policy the website in question had in place.

With hashes, a user is authenticated by comparing hash values. The user enters the password, the system passes the password through the chosen hashing algorithm, and compares it to the hash stored in the database. If they match, this user has provided a valid password.

Hashing also shields users from prying database administrators. Since the hash value does not reveal anything about the original plaintext password, a person with access to the database would have to employ a similar password-cracking strategy to recover a user’s original password.

**HASHED AND SALTED PASSWORDS**

A step up from simple password hashing is the current best-practice, salting and then hashing. This approach is similar to hashing-only, but includes adding a cryptographic “salt” to the password before passing it through a hash function. Typically, this involves concatenating a random string to the beginning or end of the original password. Salting a password does not provide any additional security on a per-user basis, since the salt is known and will be leaked along with the hashed password in a data breach. However, it increases the amount of time an attacker has to dedicate to cracking the entire database. Since each hashed password now includes random additional characters, the attacker cannot simply compare his guess to the entire list and crack
multiple passwords at once, as he was able to with hashing-only. In other words, if two users share the same password, they will not hash to the same value, unlike with simple hashing (assuming that the salt is unique).

Since the salt is known and should be unique, one approach that has been considered is simply using the username as the salt. This way, the salt will always be unique (since two users can’t have the same username) and the benefits of using a salt will (apparently) still apply. This is almost true, and using this system is better than hashing-only, but is not without drawbacks.

Firstly, one of the main attacks on passwords hashed through any hashing algorithm are so-called pre-computed “rainbow tables”. A rainbow table is a table that contains pre-computed hashes for specific inputs. This means that the password cracker can now use a rainbow table to compare the leaked hashes he has received and find the plaintext passwords without additional computations. Salting is meant to prevent this; now, rainbow tables cannot be pre-computed for common passwords because the salt changes the password itself.

Using a username as the salt, however, makes rainbow tables again effective. For almost all websites, there are common usernames like “admin” or “root”. Rainbow tables can be computed using this as the salt and the same attack can be mounted on these users.

Additionally, if a user has the same username across multiple websites, the database will have the same hash value computed if they are using the username as the salt, the user re-uses their password, and the same hashing algorithm is used for computation. This is a very specific case, but nevertheless is a possible result of not using random salts. Salts should be chosen from a truly random space to provide the most benefit.

**Insecure Algorithms**

An arbitrarily-chosen hash function is likely to be insecure. Here, two common functions in use today are discussed, MD5 and SHA-1. Neither should be used for storing password hashes.

**MD5**

MD5 is a message-digest algorithm that takes an arbitrary-length input and produces a 128-bit output (Pornin, 2011). Designed in 1991 by Ron Rivest (Rivest, 1992), it is an update to MD4, an earlier hash function. MD5 is widely utilized and is oftentimes used as the only hashing function that protects plaintext passwords. Since its introduction, MD5 has shown to be vulnerable to collisions (different inputs can result in the same hash value), among a host of other issues. In fact, the CMU Software Engineering Institute considers it “cryptographically broken and unsuitable for further use.”

Although these problems are troubling, the greatest weakness for password storage is MD5’s speed. Two ATI Radeon 7970 graphics cards can compute over 23 billion MD5 hashes per second. As mentioned, the more hashes that can be computed per second, the easier it will be to crack a list of passwords.

MD5 should never be used to store hashed passwords, even if salted. The only passwords that resist cracking with this algorithm are extremely long, random-character passwords, which are not typical.

**HASHING ALGORITHM SELECTION**

As was mentioned previously, selection of which specific hashing algorithm is used is a central concern if a company decides to utilize password hashing. Since any one-way hashing algorithm will ‘work’ in practice, companies can get away with selecting ones that are not ideal for password storage. Most often, companies fail to select correct hashing algorithms due to ignorance of best practices or due to the difficulty of changing an authentication system once it is set up. When a company is young, the focus likely is not on password security. As the business grows, companies may be hesitant to change authentication systems because they will inconvenience users at no immediate benefit. Ideally, best practices would be followed from the beginning of a company’s life and updated upon obsolescence. An examination of a few popular and ideal algorithms follows.
SHA-1

SHA-1 (secure hash function 1) was designed by the United States National Security Agency in 1995 as a successor to SHA-0 ("Federal Information Processing Standards," 2002). It produces a 160-bit output to a variable-length input. SHA-1 is the most widely used of the SHA family of hash functions, but attacks have been found (Biham et al., 2005) and NIST now recommends that federal agencies utilize SHA-2, its successor.

SHA-1 is also frequently used for password hashing, and is only slightly better than MD5. The same graphics cards as mentioned before can compute over 8 billion hashes of SHA-1 per second, roughly 1/3 of the speed of MD5. This is still an unacceptably high rate of hashing for passwords, however. SHA-1, when applied to passwords, suffers the same vulnerabilities that any fast hash function does. As such, it should not be used.

Secure Algorithms

There are currently a few algorithms accepted as best practices. The focus here is on PBKDF2 and bcrypt, which are both key derivation functions designed to be slow (and, therefore, more secure).

PBKDF2

The Password-Based Key Derivation Function 2 was introduced by RSA Laboratories in 2000, meant to replace PBKDF1 (Kaliski, 2000). It produces a variable-length key and is widely implemented as part of programming language libraries. PBKDF2 relies on an underlying hash function (such as MD5 or SHA-1) and a salt value to compute a derived key. It continually applies the hashing function, with a variable length of iterations.

PBKDF2 is secure because of the iterations it includes. Instead of passing a password through one iteration of MD5, PBKDF2 passes it through the algorithm at least 1000 times. This means that the amount of iterations that an attacker can perform per second fall precipitously.

bcrypt

Designed by Niels Provos and David Mazieres in 1999, bcrypt is a key derivation function that is similar to, but stronger than, PBKDF2 (Provos, 1999). bcrypt is based on the Blowfish cipher, which relies on constant changes to the underlying table structure during key derivation. This means that GPU-based acceleration techniques become less effective, and the speed of guessing is greatly reduced (Pornin, 2011). Current benchmarks place bcrypt on 5,000 to 10,000 on comparable CPU/graphics-card combinations, though speeds of up to 500,000 are possible through specialized set-ups ("John the Ripper benchmarks," 2013).

Although there are other, more recent algorithms available for secure hash storage, bcrypt and PBKDF2 are two competent choices. Brute-force guessing attacks on passwords stored with these algorithms will be able to crack only the most basic of choices, and other types of attacks will still be very limited by the attacker’s computation power. If a data breach does happens within a service, these algorithms will allow ample time for the organization to assess the damage and reset affected passwords without having accounts compromised. Additionally, through communication with users and other companies, the usual worrisome spillover effects of password leakage can be corrected.

USER PASSWORD SELECTION

While changes in the way an Internet service stores passwords have the biggest effect on overall password security, an individual user cannot themselves make such change happen. If they wish to utilize a service, they are forced to put their trust in the website’s security architects. As recent data breaches have shown, such trust is, in many cases, misplaced. In the past two years alone, there have been data breaches of tens of universally-recognizable online brands. With an average user utilizing just 5 passwords spread across 26 online accounts (Waugh, 2012), a simple back-of-the-envelope calculation suggests that hackers can expect to access 4 or 5 accounts on different websites whenever a breach happens. This is unacceptable, as improved password selection on the part of users can mitigate all but the most egregious of password storage mistakes by web developers.

Recommendations are given in increasing difficulty and/or decreasing importance. Users would be well-served in a data breach by implementing many of these suggestions.
PASSWORD REUSE

To begin with, there should be no re-use of passwords across online services. With each password re-use, the vector for attack for hackers increases, and they become more likely to breach an account. In the worst case, a user uses just one password for all websites. In the best case, there is no repetition of any password on any website. At the very least, unique passwords should be used for websites of high importance, such as for online banking accounts.

This ensures that if a breach happens at a relatively insignificant service, such as a news website, its negative effects do not spill over to more important accounts. Since smaller services are also more likely to have poor password-storage techniques, using the same password for two accounts of varying importance is not a good idea.

PASSWORD LENGTH AND CHARACTER SPACE

The most frequent way passwords are stored are with a weak hashing algorithm, such as MD5 or SHA1. Even with best-practice password storage techniques, particularly weak password selection can cause accounts to be compromised. Though there is a wealth of information online about successful password selection, there are two things that affect password strength: length and character space (for brute-force techniques).

Character space refers to the possible characters that are used within a password: for example, a password such as “bumper33” is alpha-numerical. Adding a special character will make it resistant to attacks based only on letters and numbers.

In general, the length of a password is its most important feature in brute-force attacks. The difference between the amounts of time it takes to crack MD5 passwords of length 6 versus 8 is immense, whereas cracking a 6-character password that includes only letters and one with letters, numbers, and special characters is comparable.

Dictionary-based attacks, where the password cracker utilizes a dictionary of English words and inserts common substitutions/modifications such as “0” for “o”, are widespread. This is why a password such as “bumper33” is not a good choice: “bumper” is a common English word and the additional “33” is just a small modification. However, even using all lower-case characters with four common words as a password, such as “correcthorsebatterystaple” (Findkle & Saba, 2012), results in much better security. This is primarily due to the length of the password.

Character space also has an impact on password strength, but it is not as great as length. If there is no additional work involved for adding all character types in a password, then using as many different characters as possible (upper case, lower case, numeric, special) offers the best security. Users should strive to utilize the maximum character space that a website’s password policy allows.

PASSWORD STORAGE

Password storage is of huge importance, but best practices are not always easy to implement. In the worst case, a user tries to remember all passwords. This is not a good storage scheme because the brain is not good at remembering multiple unique, long strings. Eventually, users revert to either re-using passwords or utilizing mental shortcuts to achieve quasi-security.

One common compromise is to select a ‘base’ password that is difficult to remember and make slight modifications to it for different websites based upon the website name. This way, all passwords are still unique and the user may retain the ability to ‘remember’ everything, since they are only forced to remember one string of random characters. This can be a reasonable solution for users that do not want to pursue any additional security, but the approach will break down during a targeted attack. If a cracker is not simply using automated tools to try and re-use passwords across services, they can recognize this special string (assuming they get the plaintext password) and base new passwords off of this. This is an unlikely situation, but a possibility nevertheless.

Although many online resources recommend against it, storing passwords on physical paper is not a terrible choice. The vast majority of unauthorized account accesses through stolen passwords are the result of a data breach in a service, which an individual user
has no effect on. The probability of a home burglary (in which the intruder is interested in stealing passwords) is usually much lower than the probability of a password leak. As long as such a list is hidden reasonably well, this type of password storage can prove effective. It is certainly better than password reuse.

Many users utilize the built-in password storage schemes included by browser vendors. Using this service is not a good idea; although this provides for a very good user experience, such passwords can not only be stolen by anyone with physical access to your computer (as with the physical password list), but browser and computer insecurities can expose the same information. And since these passwords are not typically stored in any encrypted fashion, such exploits are not difficult to carry out.

Ideally, a user will keep an encrypted database of passwords and refer to it when a password needs to be retrieved. There are many services available for this storage scheme, the most popular of which is likely KeePass. With KeePass, an encrypted database can be kept on a flash drive which contains all usernames and passwords to each website. Then, when a login screen requires password retrieval, the user can utilize the program to copy the password over. This type of password storage is extremely secure and very unlikely to be exploited by other programs. And since the information is encrypted, even loss of the flash drive will not result in a problem (assuming the encryption password is strong enough).

But this approach has drawbacks. It is not user-friendly, because it forces the user to go into a different program to retrieve any username or password, a significant step-down for users accustomed to the auto-filling that a browser password management scheme offers. Also, it is inflexible: in situations that the flash drive is unavailable or cannot be used, the user is helpless.

A service that solves these problems with minor trade-offs is LastPass. LastPass is an online password management service that includes extensions for all major browsers. This allows them to have the same auto-filling capability that a native browser implementation does, but also retain the cryptographic strength of storage with encryption that KeePass offers. Also, there are mobile and web apps that offer back-up ways to look up passwords if LastPass is not installed on a computer.

**TRULY RANDOM PASSWORDS**

The simple truth is that the only truly secure password is a very long, truly random, maximum-character-space password. Using a randomly generated, extended-length password (such as 16 characters) which includes all of the possible characters in its character space is the best possible selection. Using different random passwords for each web service and storing them with either KeePass or LastPass offers a huge improvement in security.

Even with a supercomputer, no cracker could expect to recover random, 16-character length passwords. Of course, if a web service with passwords stored in plaintext experiences a breach, the user’s password will be exposed – but such a situation is not truly problematic, since the password makes no indication of a password selection ‘style’ (as in the common-base technique described earlier) and provides no additional information about the user. At worst, the individual account that was leaked will be breached – but in this case, the company in question has made so many mistakes that no password selection strategy could save the user.

**OTHER NOTES**

Finally, users should be suspect of services that ask to “connect” to other accounts by providing passwords. A breach in one service could now affect any of the other connected services. Although some websites may claim not to store these passwords, always assume that this is untrue and that they are indeed retaining the information.

**ACKNOWLEDGMENTS**

Thanks to Ars Technica for providing the spark to pique my interest in password security.

**REFERENCES**

Click here to view references at the end of the journal.