Password Storage Cheat Sheet

Introduction

It is essential to store passwords in a way that prevents them from being obtained by an attacker even if the application or database is compromised. The majority of modern languages and frameworks provide built-in functionality to help store passwords safely.

After an attacker has acquired stored password hashes, they are always able to brute force hashes offline. As a defender, it is only possible to slow down offline attacks by selecting hash algorithms that are as resource intensive as possible.

This cheat sheet provides guidance on the various areas that need to be considered related to storing passwords. In short:

- **Use Argon2id with a minimum configuration of 19 MiB of memory, an iteration count of 2, and 1 degree of parallelism.**
- **If Argon2id is not available, use scrypt with a minimum CPU/memory cost parameter of (2^17), a minimum block size of 8 (1024 bytes), and a parallelization parameter of 1.**
- **For legacy systems using bcrypt, use a work factor of 10 or more and with a password limit of 72 bytes.**
- **If FIPS-140 compliance is required, use PBKDF2 with a work factor of 600,000 or more and set with an internal hash function of HMAC-SHA-256.**
- **Consider using a pepper to provide additional defense in depth (though alone, it provides no additional secure characteristics).**

Background

Hashing vs Encryption

Hashing and encryption both provide ways to keep sensitive data safe. However, in almost all circumstances, **passwords should be hashed, NOT encrypted.**

Hashing is a one-way function (i.e., it is impossible to "decrypt" a hash and obtain the original plaintext value). Hashing is appropriate for password validation. Even if an attacker obtains the hashed password, they cannot enter it into an application's password field and log in as the victim.

Encryption is a two-way function, meaning that the original plaintext can be retrieved.

Encryption is appropriate for storing data such as a user's address since this data is displayed in plaintext on the user's profile. Hashing their address would result in a garbled mess.

In the context of password storage, encryption should only be used in edge cases where it is necessary to obtain the original plaintext password. This might be necessary if the application needs to use the password to authenticate with another system that does not support a modern way to programmatically grant access, such as OpenID Connect (OIDC). Where possible, an alternative architecture should be used to avoid the need to store passwords in an encrypted form.

For further guidance on encryption, see the Cryptographic Storage Cheat Sheet.

How Attackers Crack Password Hashes

Although it is not possible to "decrypt" password hashes to obtain the original passwords, it is possible to "crack" the hashes in some circumstances.

The basic steps are:

- Select a password you think the victim has chosen (e.g. `password1!`)
- Calculate the hash
- Compare the hash you calculated to the hash of the victim. If they match, you have correctly "cracked" the hash and now know the plaintext value of their password.

This process is repeated for a large number of potential candidate passwords. Different methods can be used to select candidate passwords, including:

- Lists of passwords obtained from other compromised sites
- Brute force (trying every possible candidate)
- Dictionaries or wordlists of common passwords

While the number of permutations can be enormous, with high speed hardware (such as GPUs) and cloud services with many servers for rent, the cost to an attacker is relatively small to do successful password cracking especially when best practices for hashing are not followed.

Strong passwords stored with modern hashing algorithms and using hashing best practices should be effectively impossible for an attacker to crack. It is your responsibility as an application owner to select a modern hashing algorithm.

Password Storage Concepts

Salting

A salt is a unique, randomly generated string that is added to each password as part of the

hashing process. As the salt is unique for every user, an attacker has to crack hashes one at a time using the respective salt rather than calculating a hash once and comparing it against every stored hash. This makes cracking large numbers of hashes significantly harder, as the time required grows in direct proportion to the number of hashes.

Salting also protects against an attacker pre-computing hashes using rainbow tables or database-based lookups. Finally, salting means that it is impossible to determine whether two users have the same password without cracking the hashes, as the different salts will result in different hashes even if the passwords are the same.

Modern hashing algorithms such as Argon2id, bcrypt, and PBKDF2 automatically salt the passwords, so no additional steps are required when using them.

Peppering

A pepper can be used in addition to salting to provide an additional layer of protection. The purpose of the pepper is to prevent an attacker from being able to crack any of the hashes if they only have access to the database, for example, if they have exploited a SQL injection vulnerability or obtained a backup of the database.

One of several peppering strategies is to hash the passwords as usual (using a password hashing algorithm) and then HMAC or encrypt the hashes with a symmetrical encryption key before storing the password hash in the database, with the key acting as the pepper. Peppering strategies do not affect the password hashing function in any way.

- The pepper is **shared between stored passwords**, rather than being *unique* like a salt.
- Unlike a password salt, the pepper **should not be stored in the database**.
- Peppers are secrets and should be stored in "secrets vaults" or HSMs (Hardware Security Modules). See the Secrets Management Cheat Sheet for more information on securely storing secrets.
- Like any other cryptographic key, a pepper rotation strategy should be considered.

Work Factors

The work factor is essentially the number of iterations of the hashing algorithm that are performed for each password (usually, it's actually `2^work` iterations). The purpose of the work factor is to make calculating the hash more computationally expensive, which in turn reduces the speed and/or increases the cost for which an attacker can attempt to crack the password hash. The work factor is typically stored in the hash output.

When choosing a work factor, a balance needs to be struck between security and performance. Higher work factors will make the hashes more difficult for an attacker to crack but will also make the process of verifying a login attempt slower. If the work factor is too high, this may

degrade the performance of the application and could also be used by an attacker to carry out a denial of service attack by making a large number of login attempts to exhaust the server's CPU.

There is no golden rule for the ideal work factor - it will depend on the performance of the server and the number of users on the application. Determining the optimal work factor will require experimentation on the specific server(s) used by the application. As a general rule, calculating a hash should take less than one second.

Upgrading the Work Factor

One key advantage of having a work factor is that it can be increased over time as hardware becomes more powerful and cheaper.

The most common approach to upgrading the work factor is to wait until the user next authenticates and then to re-hash their password with the new work factor. This means that different hashes will have different work factors and may result in hashes never being upgraded if the user doesn't log back into the application. Depending on the application, it may be appropriate to remove the older password hashes and require users to reset their passwords next time they need to login in order to avoid storing older and less secure hashes.

Password Hashing Algorithms

There are a number of modern hashing algorithms that have been specifically designed for securely storing passwords. This means that they should be slow (unlike algorithms such as MD5 and SHA-1, which were designed to be fast), and how slow they are can be configured by changing the work factor.

Websites should not hide which password hashing algorithm they use. If you utilize a modern password hashing algorithm with proper configuration parameters, it should be safe to state in public which password hashing algorithms are in use and be listed here.

The main three algorithms that should be considered are listed below:

Argon2id

Argon2 is the winner of the 2015 Password Hashing Competition. There are three different versions of the algorithm, and the Argon2id variant should be used, as it provides a balanced approach to resisting both side-channel and GPU-based attacks.

Rather than a simple work factor like other algorithms, Argon2id has three different parameters that can be configured. Argon2id should use one of the following configuration settings as a base minimum which includes the minimum memory size (m), the minimum number of iterations (t) and the degree of parallelism (p).

- m=47104 (46 MiB), t=1, p=1 (Do not use with Argon2i)
- m=19456 (19 MiB), t=2, p=1 (Do not use with Argon2i)
- m=12288 (12 MiB), t=3, p=1
- m=9216 (9 MiB), t=4, p=1
- m=7168 (7 MiB), t=5, p=1

These configuration settings are equivalent in the defense they provide. The only difference is a trade off between CPU and RAM usage.

scrypt

scrypt is a password-based key derivation function created by Colin Percival. While Argon2id should be the best choice for password hashing, scrypt should be used when the former is not available.

Like Argon2id, scrypt has three different parameters that can be configured. scrypt should use one of the following configuration settings as a base minimum which includes the minimum CPU/memory cost parameter (N), the blocksize (r) and the degree of parallelism (p).

- N=2^17 (128 MiB), r=8 (1024 bytes), p=1
- N=2^16 (64 MiB), r=8 (1024 bytes), p=2
- N=2^15 (32 MiB), r=8 (1024 bytes), p=3
- N=2^14 (16 MiB), r=8 (1024 bytes), p=5
- N=2^13 (8 MiB), r=8 (1024 bytes), p=10

These configuration settings are equivalent in the defense they provide. The only difference is a trade off between CPU and RAM usage.

bcrypt

The bcrypt password hashing function should be the best choice for password storage in legacy systems or if PBKDF2 is required to achieve FIPS-140 compliance.

The work factor should be as large as verification server performance will allow, with a minimum of 10.

Input Limits

bcrypt has a maximum length input length of 72 bytes for most implementations. To protect against this issue, a maximum password length of 72 bytes (or less if the implementation in use has smaller limits) should be enforced when using bcrypt.

Pre-Hashing Passwords

An alternative approach is to pre-hash the user-supplied password with a fast algorithm such as SHA-256, and then to hash the resulting hash with bcrypt (i.e., `bcrypt(base64(hmac-sha256(data:$password, key:$pepper)), $salt, $cost)`). This is a dangerous (but common) practice that **should be avoided** due to password shucking and other issues when combining bcrypt with other hash functions.

PBKDF2

PBKDF2 is recommended by NIST and has FIPS-140 validated implementations. So, it should be the preferred algorithm when these are required.

PBKDF2 requires that you select an internal hashing algorithm such as an HMAC or a variety of other hashing algorithms. HMAC-SHA-256 is widely supported and is recommended by NIST.

The work factor for PBKDF2 is implemented through an iteration count, which should set differently based on the internal hashing algorithm used.

- PBKDF2-HMAC-SHA1: 1,300,000 iterations
- PBKDF2-HMAC-SHA256: 600,000 iterations
- PBKDF2-HMAC-SHA512: 210,000 iterations

Parallel PBKDF2

- PPBKDF2-SHA512: cost 2
- PPBKDF2-SHA256: cost 5
- PPBKDF2-SHA1: cost 10

These configuration settings are equivalent in the defense they provide. (Number as of december 2022, based on testing of RTX 4000 GPUs)

PBKDF2 Pre-hashing

When PBKDF2 is used with an HMAC, and the password is longer than the hash function's block size (64 bytes for SHA-256), the password will be automatically pre-hashed. For example, the password "This is a password longer than 512 bits which is the block size of SHA-256" is converted to the hash value (in hex): `fa91498c139805af73f7ba275cca071e78d78675027000c99a9925e2ec92eedd`.

A good implementation of PBKDF2 will perform pre-hashing before the expensive iterated hashing phase, but some implementations perform the conversion on each iteration. This can make hashing long passwords significantly more expensive than hashing short passwords. If a user can supply very long passwords, there is a potential denial of service vulnerability, such as the one published in Django in 2013. Manual pre-hashing can reduce this risk but requires

adding a salt to the pre-hash step.

Upgrading Legacy Hashes

For older applications built using less secure hashing algorithms such as MD5 or SHA-1, these hashes should be upgraded to modern password hashing algorithms as described above. When the user next enters their password (usually by authenticating on the application), it should be re-hashed using the new algorithm. It would also be good practice to expire the users' current password and require them to enter a new one so that any older (less secure) hashes of their password are no longer useful to an attacker.

However, this approach means that old (less secure) password hashes will be stored in the database until the user logs in. Two main approaches can be taken to avoid this dilemma.

One method is to expire and delete the password hashes of users who have been inactive for an extended period and require them to reset their passwords to login again. Although secure, this approach is not particularly user-friendly. Expiring the passwords of many users may cause issues for support staff or may be interpreted by users as an indication of a breach.

An alternative approach is to use the existing password hashes as inputs for a more secure algorithm. For example, if the application originally stored passwords as `md5($password)`, this could be easily upgraded to `bcrypt(md5($password))`. Layering the hashes avoids the need to know the original password; however, it can make the hashes easier to crack. These hashes should be replaced with direct hashes of the users' passwords next time the user logs in.

Assume that whatever password hashing method is selected will have to be upgraded in the future. Ensure that upgrading your hashing algorithm is as easy as possible. For a transition period, allow for a mix of old and new hashing algorithms. Using a mix of hashing algorithms is easier if the password hashing algorithm and work factor are stored with the password using a standard format, for example, the modular PHC string format.

International Characters

Ensure your hashing library is able to accept a wide range of characters and is compatible with all Unicode codepoints. Users should be able to use the full range of characters available on modern devices, in particular mobile keyboards. They should be able to select passwords from various languages and include pictograms. Prior to hashing the entropy of the user's entry should not be reduced. Password hashing libraries need to be able to use input that may contain a NULL byte.