Password Storage Cheat Sheet

Introduction

This cheat sheet advises you on the proper methods for storing passwords for authentication. When passwords are stored, they must be protected from an attacker even if the application or database is compromised. Fortunately, a majority of modern languages and frameworks provide built-in functionality to help store passwords safely.

Passwords should never be stored in plain text. Instead, they must be protected using strong, slow hashing algorithms such as Argon2id, bcrypt, or PBKDF2. A unique salt must be added to each password to prevent attackers from using precomputed lookup tables like rainbow tables. Fast hashing algorithms such as SHA-256 are not suitable for password storage because they allow attackers to perform large numbers of guesses quickly. Using slow, memory-hard algorithms makes brute-force attacks significantly more difficult, expensive, and time-consuming.

To sum up our recommendations:

- **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 can keep sensitive data safe, but in almost all circumstances, **Passwords should be securely hashed using modern, adaptive hashing algorithms (e.g., Argon2id, bcrypt, or**

PBKDF2), rather than encrypted or stored in plaintext.

Because **hashing is a one-way function** (i.e., it is impossible to "decrypt" a hash and obtain the original plaintext value), it is the most appropriate approach for password validation. Even if an attacker obtains the hashed password, they cannot use it to log in as the victim.

Since **encryption is a two-way function**, attackers can retrieve the original plaintext from the encrypted data. It can be used to store 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.

The only time encryption should be used in passwords is 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). Wherever 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.

When Password Hashes Can Be Cracked

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.

However, there are some situations where an attacker can "crack" the hashes in some circumstances by doing the following:

- Selecting a password you think the victim has chosen (e.g. `password1!`)
- Calculating the hash
- Comparing 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.

Usually, the attacker will repeat this process with a list of large number of potential candidate passwords, such as:

- 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.

Methods for Enhancing Password Storage

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's 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.

At the algorithm and specification level, modern password hashing functions such as Argon2id, bcrypt, and PBKDF2 require the caller to provide a salt. However, most widely used implementations and libraries automatically generate and manage salts internally, so application developers typically do not need to handle salt generation manually when using these libraries correctly.

Peppering

Peppering is a class of strategies that can be used in addition to salting to provide an additional layer of protection. It prevents 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.

Common requirements for peppering strategies

- A pepper is **shared between stored passwords**, rather than being *unique* to an individual password like a password salt.
- Unlike a password salt, the pepper should not be public and **should not be stored along with the generated hash**. The pepper should be stored separately from the password 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.

- In the event of a pepper's compromise, the pepper will have to be changed. Peppers cannot be changed without knowledge of a user's password. Therefore changing a pepper will require forcing all users whose passwords were protected by the previous pepper to reset their passwords.

Pre-hashing peppers

In this strategy, a pepper is added to a password before being hashed by a password hashing algorithm. The computed hash is then stored in the database. In this case the pepper should be a random value generated securely. See the Cryptographic_Storage_Cheat_Sheet for more information on securely generating random values.

Post-hashing peppers

In this strategy, a password is hashed as usual using a password hashing algorithm. The resulting password hash is then hashed again using an HMAC (e.g., HMAC-SHA256, HMAC-SHA512, depending on the desired output length) before storing the resulting hash in the database. In this case the pepper is acting as the HMAC key and should be generated as per requirements of the HMAC algorithm.

Using Work Factors

The work factor is the number of iterations of the hashing algorithm that are performed for each password (usually, it's actually `2^work` iterations). The work factor is typically stored in the hash output. It makes 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.

When you choose a work factor, strike a balance between security and performance. Though higher work factors make hashes more difficult for an attacker to crack, they will slow down the process of verifying a login attempt. If the work factor is too high, the performance of the application may be degraded, which could be used by an attacker to carry out a denial of service attack by exhausting the server's CPU with a large number of login attempts.

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, then re-hash their password with the new work factor. The different hashes will have different work factors and hashes may never be 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

Some modern hashing algorithms have been specifically designed to securely store passwords. This means that they should be slow (unlike algorithms such as MD5 and SHA-1, which were designed to be fast), and you can change how slow they are by changing the work factor.

You do not need to hide which password hashing algorithm is used by an application. 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.

When selecting a password hashing algorithm, developers should prefer modern algorithms that are designed to resist both GPU-based and memory-based attacks. Where available, newer algorithms should be chosen for new applications, while older algorithms may still be acceptable for legacy systems with appropriate configuration.

Three hashing algorithms that should be considered.

Argon2id

Argon2 was the winner of the 2015 Password Hashing Competition. Out of the three Argon2 versions, use the Argon2id variant since 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: the base minimum of the minimum memory size (m), the minimum number of iterations (t), and the degree of parallelism (p). We recommend the following configuration settings:

These parameters control how computationally expensive it is to compute a password hash. Increasing memory usage, iteration count, or parallelism makes password cracking attempts significantly slower and more costly for attackers, while still remaining practical for legitimate authentication requests when tuned appropriately.

- 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 provide an equal level of defense, and 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 parameters that can be configured: the minimum memory cost parameter (N), the blocksize (r), and the degree of parallelism (p). Use one of the following settings:

- 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 provide a similar minimal level of defense, with the main trade-off between parallelism and RAM usage.

bcrypt

The bcrypt password hashing function **should only** be used for password storage in legacy systems where Argon2 and scrypt are not available.

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

Input Limits of bcrypt

bcrypt has a maximum length input length of 72 bytes for most implementations, so you should enforce a maximum password length of 72 bytes (or less if the bcrypt implementation in use has smaller limits).

Pre-Hashing Passwords with bcrypt

An alternative approach is to pre-hash the user-supplied password with a fast algorithm such as SHA-2, HMAC, or BLAKE3 and then to hash the resulting hash value with bcrypt (i.e., `bcrypt(H($password)), $salt, $cost)`.. This can be **dangerous** because of null bytes in the hash output value and because of password shucking.

The original bcrypt expects a null terminated password string, this means that the hash value will only be used to the first null byte in the hash value. (`bcrypt(H($password)), $salt, $cost) == bcrypt("", $salt, $cost)` if `H($password)[0] == 0`) This increases the chance of finding a collision when combining bcrypt with other hash functions and can be avoided by encoding the hash value to printable string with something like base64. base64 can increases the length of the hash value above 72 characters and so there is a bit of truncation for large hash values from hashes like SHA-512, this is negligible.

Password shucking uses the fact, that it is easy to check if `bcrypt(base64(H($password))), $salt, $cost) == bcrypt(base64($leaked_hash), $salt, $cost)`. If the inner hash function `H` is used with the same password somewhere else and known to an attacker cracking the password can be reduced to breaking the hash function `H`. Just using pure SHA-512, (i.e. `bcrypt(base64(sha512($password))), $salt, $cost)`) is a **dangerous practice** and is as secure as just using pure SHA-512. Password shucking only works if a leaked hash is known to the attacker, either through a breach database or rainbow tables. To mitigate password shucking a pepper can be used.

To summarize if bcrypt has to be used and the password should to be pre-hashed you should do `bcrypt(base64(hmac-sha384(data:$password, key:$pepper)), $salt, $cost)` and store the pepper not in the database.

PBKDF2

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

The PBKDF2 algorithm 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-SHA256: 600,000 iterations (recommended)
- PBKDF2-HMAC-SHA512: 220,000 iterations

- PBKDF2-HMAC-SHA1: 1,400,000 iterations — **legacy only**, do not select for new systems. NIST SP 800-131A Rev. 2 disallows SHA-1 for new use after 2030.

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`.

Good implementations of PBKDF2 perform pre-hashing before the expensive iterated hashing phase. However, some implementations perform the conversion on each iteration, which can make hashing long passwords significantly more expensive than hashing short passwords. When users supply very long passwords, a potential denial of service vulnerability could occur, such as the one published in Django during 2013. Manual pre-hashing can reduce this risk but requires adding a salt to the pre-hash step.

Upgrading Legacy Hashes

Older applications that use less secure hashing algorithms, such as MD5 or SHA-1, can be upgraded to modern password hashing algorithms as described above. When the users enter their password (usually by authenticating on the application), that input should be re-hashed using the new algorithm. Defenders should 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 means that old (less secure) password hashes will be stored in the database until the user logs in. You can take one of two approaches to avoid this dilemma.

Upgrade Method One: 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.

Upgrade Method Two: 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.

Remember that once your password hashing method is selected, it will have to be upgraded in the future, so ensure that upgrading your hashing algorithm is as easy as possible. During the 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

Your hashing library must be able to accept a wide range of characters and should be compatible with all Unicode codepoints, so users can use the full range of characters available on modern devices - especially 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, and password hashing libraries need to be able to use input that may contain a NULL byte.