

# PASSWORD ENCRYPTION

## I. Introduction & Problem statement

The term “authentication” is often associated with a scenario where the participating parties wish to verify the identity of others and detect the presence of an impersonator. On a high-level understanding, in order to prove one’s authenticity, an individual should be required to provide information specific to verification.

Within the diverse range of formats on which such information can take, passwords have proven to be most popular, employed in websites and applications of all scales. In essence, a password is a string of some length, made up of characters from a specified alphabet. From the user’s perspective, setting and using one’s passwords is intuitive - when logging in, if the password is correct, i.e. similar to the one registered in the system, the user’s identity is verified.

However, from the system’s perspective, its responsibility is substantially more challenging. Intuitively, passwords are to be stored in the system’s database, in some fashion, so that it can permit or prevent an attempt to log in. However, it is this password storage that gives rise to the most vulnerable component of the system.

In recent history, there have been cases where even seemingly secure and large-scale sites suffered from database breaches, leading to massive leaks in users’ information. For example, in June 2020, almost 270 million records of the amateur literary site Wattpad were leaked, shared, and sold publicly on a hacking forum. Compromised data include an extensive set of personal information: bios, date of birth, email addresses, genders, geographic locations, IP addresses, names, passwords, social media profiles, etc.

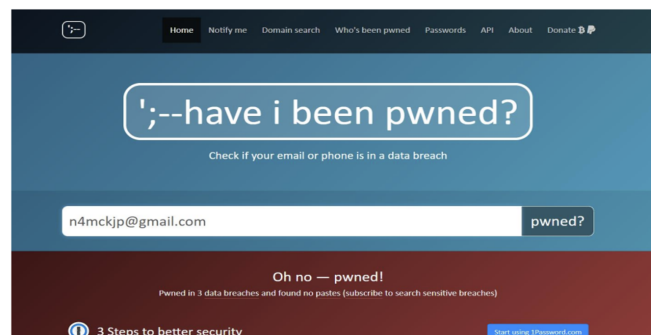


Image from: <https://haveibeenpwned.com/>

## II. Project overview

Clearly, the responsibility to safely store users' information, primarily passwords, is of utmost importance to systems and websites. In this report, we set out to survey on the most prominent approaches to password storage and encryption among companies, the most common password attacks, and potential solutions to such threats.

In the first part, we give an overview of the theoretical knowledge behind password storage, including modes of storage (Section III), potential attacks (Section IV), and common solutions (Section V). Section V also introduces the an additional idea of peppering. In the second part, we will delve into the workings of some hashing algorithms, including their properties, and reasons backing their security. In addition, we will talk more about password security schemes, including special hashing functions, techniques such as salting and peppering, or multilayer encryption, etc. We will also demonstrate the implementation of password authentication in code, using common libraries.

The ideas we present here build from the knowledge within the course on Information Security, specifically in Chapter 6: Authentication. Within the textbook, the general idea behind storage using hashing algorithms is already mentioned, along with the salting technique used before hashing. However, the details behind a specific hashing function are overlooked, and more advanced techniques such as bcrypt, scrypt, and multilayer encryption are not included. We set out to expand on this topic and present our findings. These expansions are located in Section VI.

## III. Password storage

### Database leaks

Generally, database breaches may come from SQL injection attacks, misconfigured servers, or just a disgruntled employee or a corrupted official. However, a compromised database does not automatically entail a breach of personal passwords. In reality, this depends on the mode of password that the system implements.

### Storing passwords in plaintext

Intuitively, this is the simplest approach to password storage. Whenever there is an attempt to log in, the provided password is compared to the stored entry in the database, and if the password matches, the user is allowed to access the system.

However, this practice is frowned upon due to its obvious drawbacks in security. In the case of a slight mistake that allows an attacker to access the database, all passwords will be breached immediately. Even if this database is locked with some key, this key runs a similar risk of falling into

the hands of an adversary. In addition, the negative implications of a breached plaintext password is multiplied as realistically, many people habitually use one password for all services they may register.


At a glance, this mode of storage seems insecure altogether enough that one may expect that no company would utilize this approach. However, data breaches in the past have shown that this storage mode is actually utilized, sometimes by even well-known and supposedly secure sites.



## League of Legends

In June 2012, the multiplayer online game League of Legends suffered a data breach. At the time, the service had more than 32 million registered accounts and the breach affected various personal data attributes including "encrypted" passwords. In 2018, a 339k record subset of the data emerged with email addresses, usernames and **plain text** passwords, likely cracked from the original cryptographically protected ones.

**Breach date:** 11 June 2012  
**Date added to HIBP:** 28 July 2018  
**Compromised accounts:** 339,487  
**Compromised data:** Email addresses, Passwords, Usernames  
[Permalink](#)



## Yahoo

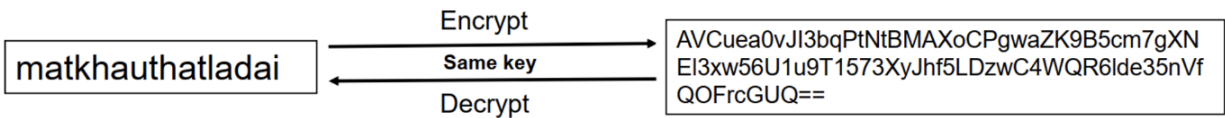
In July 2012, Yahoo! had their online publishing service "Voices" compromised via a SQL injection attack. The breach resulted in the disclosure of nearly half a million usernames and passwords stored in **plain text**. The breach showed that of the compromised accounts, a staggering 59% of people who also had accounts in the Sony breach reused their passwords across both services.

**Breach date:** 11 July 2012  
**Date added to HIBP:** 4 December 2013  
**Compromised accounts:** 453,427  
**Compromised data:** Email addresses, Passwords  
[Permalink](#)

## Storing passwords with two-way encryption

Up one security level, this mode is based on secret-key cryptography such as DES, or RSA. Given an appropriate key management scheme, this approach is more secure than the previous storage method. Specifically, when registering a password, this password is passed through a secret-key encryption algorithm, and the ciphertext itself is stored in the system's server, rather than the plaintext.

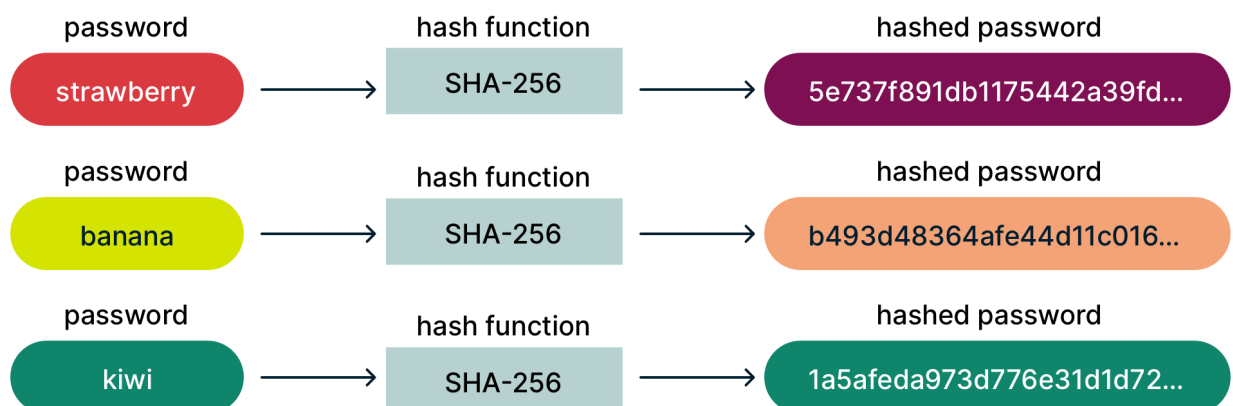
However, underneath this ciphertext is still the password in plaintext, and the key weakness to this storage mode is the fact that if the encryption key is known, it is possible to reverse the process of encryption, i.e. decrypt, in order to restore the password in plaintext. Obviously, secret-key encryption is more suitable in the scenario where data is to be sent and read, rather than just stored.



## Storing passwords using hash functions

In mathematics, a hashing algorithm is a complex math function that takes as input a string of text, and returns as output a seemingly random string of fixed length. Generally, a well-designed hashing algorithm displays some desirable qualities. First, if the input to the hashing algorithm changes slightly, the output changes significantly and in a seemingly randomized way, which is needed so that attacks to detect the patterns are futile. Second, a good hashing algorithm is a one-way encryption - the forward process is straightforward and fast, but the reverse, going from the hash to restore the original string, is infeasible.

It is these properties that make hashing algorithms suitable for hashing passwords - similar to the previous mode, the hashed password is stored in the server instead of the plaintext, but obtaining the hash theoretically cannot lead to knowing the original password. The process of logging in follows the previous pattern - the user logs in with a password, which goes through the hash. The hash is compared with the one stored in the database, and logging in is successful when the hash matches.



Nowadays, using hashing functions for password storage has increasingly become the norm. The following section thus deals with the most popular attacks on hashed passwords.

## IV. Hashed password attacks

### Brute force attack

A crucial weakness to using hashing functions is that they are innately vulnerable to brute force attacks because of their optimized-for-speed design, while simultaneously modern GPUs are becoming faster. For example, the VGA GTX 1660 GPU can perform 7.31 million hashes per second, so a 6-character password (with  $26^6$  possible passwords) will be broken in under a minute.

Brute force attacks, as the name suggests, involve trying every possible combination of characters, putting them through a known hash function, and testing against the target hash.

### Rainbow table attack

A hash function, no matter how complex, is a function, meaning that two similar inputs should return two similar outputs. This, combined with the observed high usage of some specific passwords in real life, means that the hash of these passwords can be precomputed and stored in some database. Later, given a hash, instead of having to try all possible passwords, an attacker can see whether the hash is already in the pre-computed database, and if so, the plaintext password automatically becomes known.


---

### Free Password Hash Cracker

Enter up to 20 non-salted hashes, one per line:

7085e6b4fb5bf71436221f6ccd1af40c

I'm not a robot

  
reCAPTCHA  
[Privacy](#) · [Terms](#)

Crack Hashes

**Supports:** LM, NTLM, md2, md4, md5, md5(md5\_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1 sha1\_bin)), QubesV3.1BackupDefaults

Hash	Type	Result
7085e6b4fb5bf71436221f6ccd1af40c	md5	anhyeuem

**Color Codes:** Green: Exact match, Yellow: Partial match, Red: Not found.

[Download CrackStation's Wordlist](#)

<https://crackstation.net/>

### Dictionary attack

Instead of trying all possible passwords in a brute force manner, the choice of passwords to test is made with a specific strategy. For example, common passwords can be altered slightly (e.g. with capitalization) in order to create a list of passwords to test out. Also, personal information can also be used to create a dictionary of suspicious passwords to test out. For example, common first names (e.g. about 100) can be combined with all dates possible (365) to create passwords in the format “name + birthday”

In all of these attacks, because one password leads to only one hash, the attacker can recognize known hashes, and once a password is cracked, all users using the password are compromised. In addition, even before cracking the hash, a similar hashed password already indicates that two users have the same password, which might facilitate the attacker in some ways.

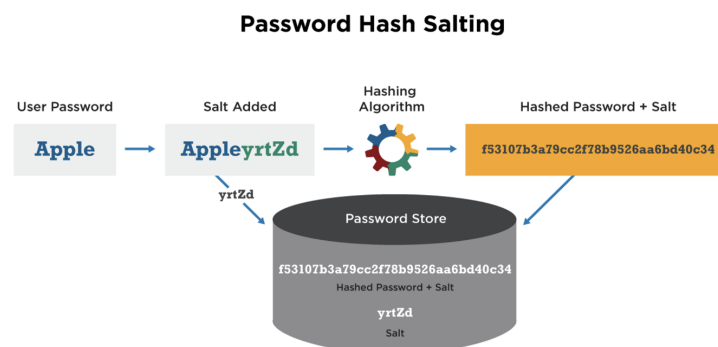
## V. Potential solutions

### Salting

In the context of password encryption, salt is a short, fixed-length random set of characters, appended to the password before hashing. A randomly generated salt is unique for each user.

For this reason, salting helps in preventing rainbow table attacks, since two users with the exact same password still have different salts appended to their passwords, leading to entirely different hashes. In addition, even if one password is cracked, other users who share the password are not compromised. The attacker shall also not know if two users share a password.

From the user's perspective, one does not have to know which salt is being used - salt is automatically added by the system storing the hashed password. In order to achieve this, the database is formatted to keep track of the username, password hash, and the salt in plaintext, and when verifying, the salt is appended before hashing. However, because of the stored plaintext salt, brute force attacks can still happen.

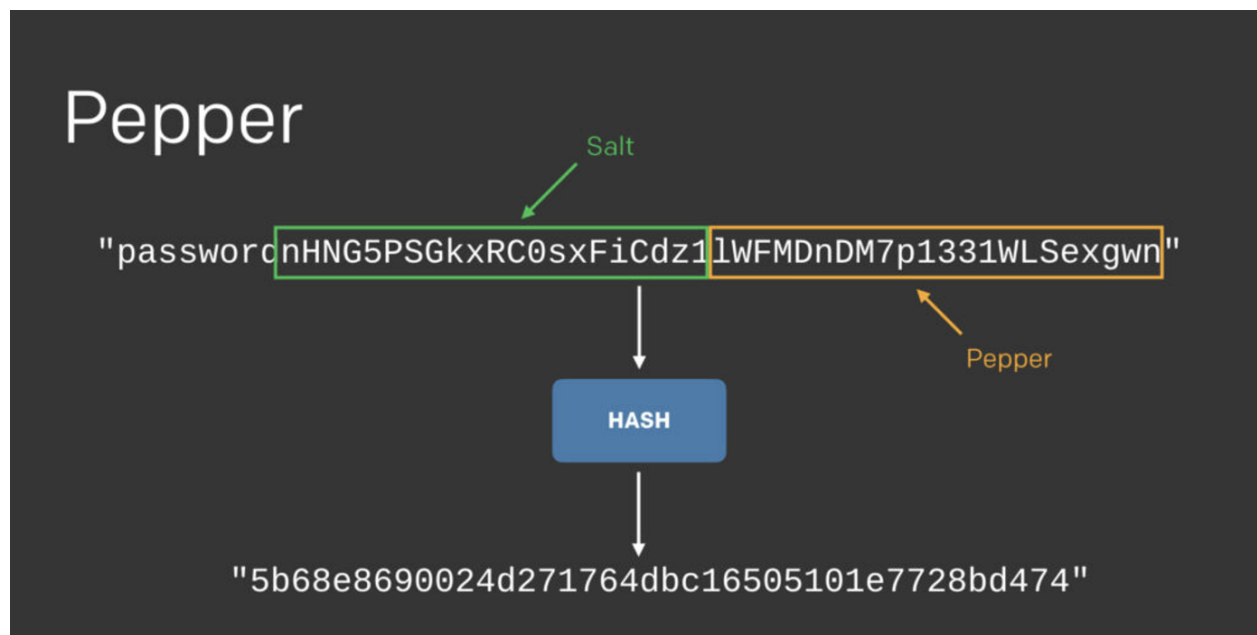


# Peppering

Also in the context of password encryption, pepper is a very short random string, also added to the password before hashing. Similar to salt, pepper is also automatically added to the user's password, and its value lies outside the scope of what the user needs to know. The core difference between salt and pepper is the fact that the latter is not stored in the system's server or database. On a high-level understanding, an attacker with only access to the database, must perform brute force attack significantly slower because of a lack of knowledge regarding the pepper, i.e. all peppers must be tried.

There are two significant approaches to using pepper. On the one hand, pepper can be a security choice by the site or system owner to be secure enough, thereby static-wide, i.e. applied to all passwords in the database. The storing of the pepper is kept secret, not in the database but in a secure and separate part of the application. On the other hand, the pepper might not be stored at all. In this case, pepper is a very short string (maybe just 52 uppercase and lowercase English letters), and the system has to cycle through all possible peppers when verifying an attempt to log in.

It is also notable that pepper is generally an experimental technique, and since cybersecurity engineers should use open-source well-proven authentication systems instead of developing one by themselves, the choice of using pepper is still limited.



## Other hash functions

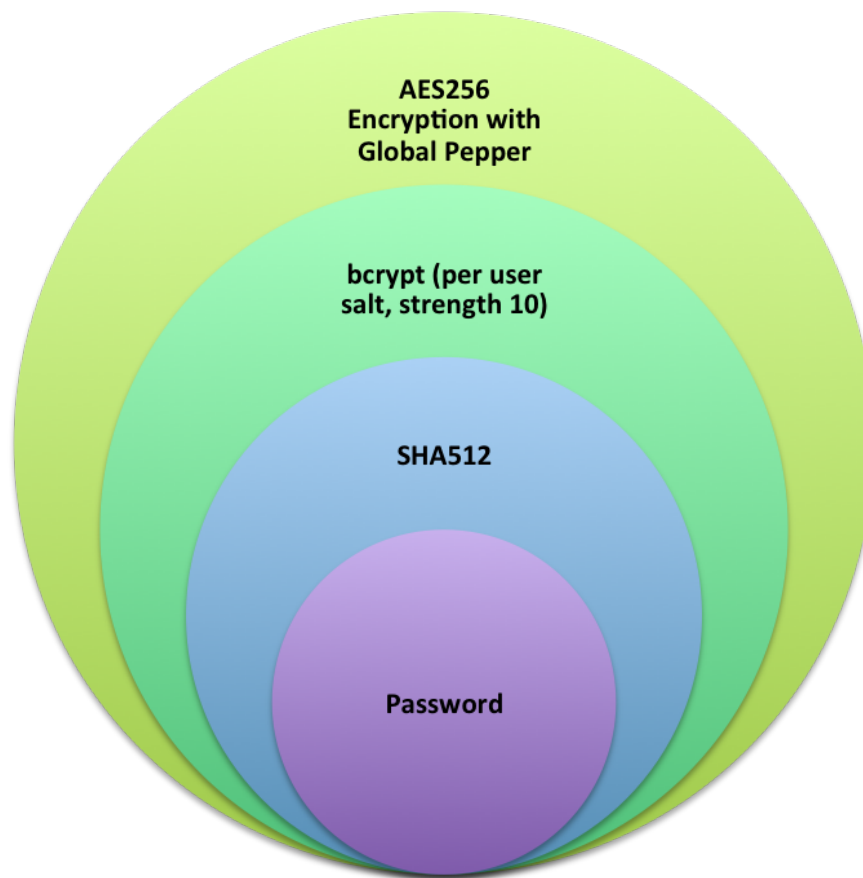
Since hash functions are designed for speed, they also allow attackers to perform brute force attacks efficiently. To counter this phenomenon, other hash functions, which are deliberately slowed down,

might be used. Examples include bcrypt, scrypt, and argon2. These are designed to completely neutralize brute-force attacks.

This effect comes from the working of these algorithms. For example, bcrypt takes as input a cost, simply understood as the number of rounds the algorithm has to go through, in order to slow down computation. As modern machines become faster, the cost parameter is increased accordingly.

Section VI will elaborate on this topic.

## Multilayer approach



A password may be hashed without salt, then passed through bcrypt, with salt, and a cost of 10, before being encrypted with AES-256 with global pepper (the encryption key is stored somewhere else, not in the database). As a result, this creates layers of security protecting a password.

## VI. Elaboration



In the realm of cybersecurity, a myriad of methods exist to thwart hackers and safeguard sensitive information through encryption and hashing techniques. However, not all methods are equally effective, and some may introduce a trade-off between security and efficiency. This section delves into the intricate world of password encryption, with a primary focus on methods that have been widely employed from the past to the present day.

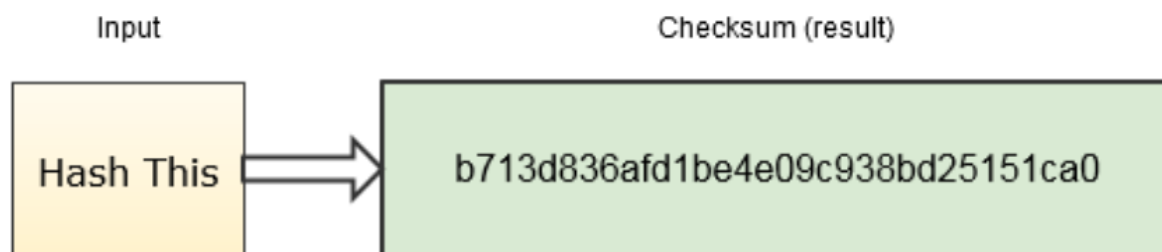
To gain a comprehensive understanding of the fundamental concepts related to password encryption, an analysis on the strengths and weaknesses of different algorithms, as well as their rooms for improvement, is needed. We focus on the MD5 hash algorithm, a cornerstone in the field of password encryption.

## MD5

### Introduction

Cryptographic hash functions take data input (or message) and generate a fixed size result (or digest). It is almost impossible to regenerate the input from the result of the hash function - fundamentally, this is not encryption.

One of the most widely used Cryptographic hash Function is **MD5** or "message digest 5". MD5 creates a 128-bit message digest from the data input which is typically expressed in 32 digits hexadecimal number. MD5 hashes are unique for different inputs regardless of the size of the input. MD5 hashes look like this:



It is widely used to make sure that the transferred file in a software has arrived safely. For example when you download a file from the Internet/Server, it might be corrupted, or there might be data loss due to connection loss, virus, hack attack or some other reason. In the context of password storage, it is also used in databases to store passwords as hashes instead of the original plaintext.

### Algorithm

The MD5 algorithm works by processing input data through a series of steps:

1. **Padding:** The input message is padded with bits so that its length is a multiple of 512 bits. This ensures that the message can be divided into equal blocks for further processing.

2. **Message block division:** The padded message is divided into 512-bit blocks, each containing 16 32-bit words.
3. **Initialization:** Four 32-bit words are used to initialize the MD5 algorithm's internal state. These values are constants and remain unchanged throughout the processing.
4. **Processing rounds:** Each message block is processed through four rounds of complex mathematical operations. These operations involve bit shifts, logical operations, additions, and nonlinear functions. The goal of these rounds is to "scramble" the data and create a unique fingerprint that is sensitive to any changes in the input.
5. **State update:** After each round, the internal state of the MD5 algorithm is updated by combining it with the results of the previous round.
6. **Output generation:** After all message blocks have been processed, the final state of the MD5 algorithm is combined to generate a 128-bit hash value. This hash value is the unique fingerprint of the original message.

Algorithm
<p><b>Input:</b> Message of any length</p> <ol style="list-style-type: none"><li>1. Padding</li><li>2. Divide message into blocks</li><li>3. Initialize internal state</li><li>4. Process each block through four rounds</li><li>5. Update internal state</li><li>6. Generate <b>128-bit</b> hash value</li></ol> <p><b>Output:</b> Unique fingerprint of the original message</p>

Properties such as sensitivity to input changes and one-wayness in the MD5 algorithm arise from its design components. MD5 employs message padding, a compression function with non-linear operations, an initialization vector, and constants. The algorithm's iterative structure, consisting of 64 rounds, ensures that even small alterations in the input lead to significant changes in the output. These properties are crucial for cryptographic hash functions, making it computationally infeasible to reverse the hash and providing resistance against finding patterns or collisions in the hashed values.

## bcrypt

### Introduction

As demonstrated earlier, MD5 was once considered a viable option for enhancing security measures. However, the rapid advancement of computer technology, coupled with the widespread use of MD5, has rendered it less secure in the face of modern threats. In fact, the popularity of MD5 has reached a point where numerous websites can decode its output in a matter of seconds. Furthermore, its

susceptibility to collision attacks, inherent speed, and absence of salting have further eroded its effectiveness as a password hashing mechanism.

Recognizing the critical need for more robust security practices, bcrypt emerged as a compelling alternative to MD5. Specifically, bcrypt was developed to address these vulnerabilities by introducing a set of essential features that significantly bolster password hashing security.

This mainly comes from the idea of key stretching, a design choice to make hashing computationally intensive. It introduces a configurable cost factor, which mandates a specific number of iterations during the hashing process. This deliberate complexity makes it considerably slower to compute hashes compared to MD5. As a result, attackers face a daunting challenge when attempting brute-force attacks, as each iteration significantly increases the time and computational resources required to test potential passwords, thereby increasing resilience against modern Graphics Processing Units (GPUs) and Application-Specific Integrated Circuits (ASICs). In addition, bcrypt incorporates salt as an inherent part of its hashing process. This means that even if two users have the same password, their resulting hashes will be distinct due to the inclusion of unique salt values. This simple yet effective measure thwarts the effectiveness of rainbow table attacks, which rely on precomputed hash values for commonly used passwords.

## Principle

The bcrypt algorithm is a password hashing function designed for security and resistance against brute-force attacks. Based on the Blowfish cipher, bcrypt incorporates several key features that enhance its security for password storage. Below is a high-level overview of how the bcrypt algorithm works, based on the simplified code and concepts we discussed:

- 1. Password and Salt Input:** Bcrypt starts by taking a user's password and a cryptographic salt.
- 2. Cost Factor:** bcrypt uses a cost factor (work factor), which determines how computationally intensive the hash computation will be. The cost factor is essentially a measure of the number of iterations (or rounds) the key setup phase will undergo, making the hashing process adaptable to future increases in computational power.
- 3. Key Setup with EksBlowfishSetup:** Bcrypt begins with a standard Blowfish key setup but modifies it to create an "expensive" key setup (EksBlowfishSetup). The EksBlowfishSetup involves initializing the Blowfish cipher's P-array and S-boxes with a predefined value (like the hexadecimal digits of  $\pi$ ). The password and salt are then mixed into the P-array and S-boxes through a series of operations, including XOR and encryption steps. This process is repeated ( $2^{\text{cost}}$ ) times, making it computationally expensive.
- 4. Encryption of Constant Text:** Bcrypt then encrypts a 24-byte constant string 64 times using the Blowfish cipher with the P-array and S-boxes set up in the previous step. This process doesn't depend on the password or the salt; it's a fixed operation using the modified state from the EksBlowfishSetup.
- 5. Formation of Hash:** The final state after the repeated encryption of the constant string forms the basis of the password hash. This hashed value is combined with the salt and the cost factor to form the final bcrypt hash.

**6. Output Format:** The output of the bcrypt function is typically formatted as a string that includes the algorithm version, the cost factor, the salt, and the hashed value. The format is generally ``$2a$[cost]$(22 character salt)[31 character hash]``. The salt and hash are usually encoded in a modified base64 format.

**7. Verification:** To verify a password, the same process is run with the stored salt and cost, and the resulting hash is compared to the stored hash. If they match, the password is correct.

Algorithm
<p><b>Input:</b> String of text (password)</p> <ol style="list-style-type: none"><li>1. Concatenate salt to password</li><li>2. Determine cost factor</li><li>3. Choose key setup, taking cost factor into account</li><li>4. Encrypt constant text</li><li>5. Form hashing algorithm with hashed constant, salt, and cost factor</li><li>6. Hash salted password</li></ol> <p><b>Output:</b> Hashed password, generally formatted <code>`\$2a\$[cost]\$(22 character salt)[31 character hash]`</code></p>

The strength of bcrypt lies in its use of salt to prevent precomputed hash attacks and its adaptive nature due to the cost factor, which can be increased as computational capabilities grow. This makes bcrypt an effective tool for securely hashing passwords.

## Implementation

From our high-level understanding of MD5 and crypt, we set out to explore details in the implementation of these hashing algorithms. The code (to be updated) can be found in our public GitHub repository below.

<https://github.com/thiemcun203/Password-Encryption.git>