Salted Password Hashing - Doing it Right

If you're a web developer, you've probably had to make a user account system. The most important aspect of a user account system is how user passwords are protected. User account databases are hacked frequently, so you absolutely must do something to protect your users' passwords if your website is ever breached. The best way to protect passwords is to employ **salted password hashing**. This page will explain why it's done the way it is.

There are a lot of conflicting ideas and misconceptions on how to do password hashing properly, probably due to the abundance of misinformation on the web. Password hashing is one of those things that's so simple, but yet so many people get wrong. With this page, I hope to explain not only the correct way to do it, but why it should be done that way.

IMPORTANT WARNING: If you are thinking of writing your own password hashing code, **please don't!**. It's too easy to screw up. No, that cryptography course you took in university doesn't make you exempt from this warning. This applies to everyone: **DO NOT WRITE YOUR OWN CRYPTO!** The problem of storing passwords has already been solved. Use either use either phpass, the PHP, C#, Java, and Ruby implementations in defuse/password-hashing, or libsodium.

If for some reason you missed that big red warning note, please go read it now. Really, this guide is **not** meant to walk you through the process of writing your own storage system, it's to explain the reasons why passwords should be stored a certain way.

You may use the following links to jump to the different sections of this page.

What is password hashing?

```
hash("hello") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hbllo") = 58756879c05c68dfac9866712fad6a93f8146f337a69afe7dd238f3364946366
hash("waltz") = c0e81794384491161f1777c232bc6bd9ec38f616560b120fda8e90f383853542
```

Hash algorithms are one way functions. They turn any amount of data into a fixed-length "fingerprint" that cannot be reversed. They also have the property that if the input changes by even a tiny bit, the resulting hash is completely different (see the example above). This is great for protecting passwords, because we want to store passwords in a form that protects them even if the password file itself is compromised, but at the same time, we need to be able to verify that a user's password is correct.

The general workflow for account registration and authentication in a hash-based account system is as follows:

1. The user creates an account.
2. Their password is hashed and stored in the database. At no point is the plain-text (unencrypted) password ever written to the hard drive.
3. When the user attempts to login, the hash of the password they entered is checked against the hash of their real password (retrieved from the database).
4. If the hashes match, the user is granted access. If not, the user is told they entered invalid login credentials.
5. Steps 3 and 4 repeat every time someone tries to login to their account.

In step 4, never tell the user if it was the username or password they got wrong. Always display a generic message like "Invalid username or password." This prevents attackers from enumerating valid usernames without knowing their passwords.

It should be noted that the hash functions used to protect passwords are not the same as the hash functions you may have seen in a data structures course. The hash functions used to implement data structures such as hash tables are designed to be fast, not secure. Only **cryptographic hash functions** may be used to implement password hashing. Hash functions like SHA256, SHA512, RipeMD, and WHIRLPOOL are cryptographic hash functions.

It is easy to think that all you have to do is run the password through a cryptographic hash function and your users' passwords will be secure. This is far from the truth. There are many ways to recover passwords from plain hashes very quickly. There are several easy-to-implement techniques that make these "attacks" much less effective. To motivate the need for these techniques, consider this very website. On the front page, you can submit a list of hashes to be cracked, and receive results in less than a second. Clearly, simply hashing the password does not meet our needs for security.

The next section will discuss some of the common attacks used to crack plain password hashes.

How Hashes are Cracked

Dictionary and Brute Force Attacks

```
      Dictionary Attack

Trying apple        : failed
Trying blueberry    : failed
Trying justinbeiber : failed
             ...
Trying letmein      : failed
Trying s3cr3t       : success!
```

```
  Brute Force Attack

Trying aaaa : failed
Trying aaab : failed
Trying aaac : failed
          ...
Trying acdb : failed
Trying acdc : success!
```

The simplest way to crack a hash is to try to guess the password, hashing each guess, and checking if the guess's hash equals the hash being cracked. If the hashes are equal, the guess is the password. The two most common ways of guessing passwords are **dictionary attacks** and **brute-force attacks**.

A dictionary attack uses a file containing words, phrases, common passwords, and other strings that are likely to be used as a password. Each word in the file is hashed, and its hash is compared to the password hash. If they match, that word is the password. These dictionary files are constructed by extracting words from large bodies of text, and even from real databases of passwords. Further processing is often applied to dictionary files, such as replacing words with their "leet speak" equivalents ("hello" becomes "h3110"), to make them more effective.

A brute-force attack tries every possible combination of characters up to a given length. These attacks are very computationally expensive, and are usually the least efficient in terms of hashes cracked per processor time, but they will always eventually find the password. Passwords should be long enough that searching through all possible character strings to find it will take too long to be worthwhile.

There is no way to prevent dictionary attacks or brute force attacks. They can be made less effective, but there isn't a way to prevent them altogether. If your password hashing system is secure, the only way to crack the hashes will be to run a dictionary or brute-force attack on each hash.

Lookup Tables

```
Searching: 5f4dcc3b5aa765d61d8327deb882cf99: FOUND: password5
Searching: 6cbe615c106f422d23669b610b564800:  not in database
Searching: 630bf032efe4507f2c57b280995925a9: FOUND: letMEin12
Searching: 386f43fab5d096a7a66d67c8f213e5ec: FOUND: mcd0nalds
Searching: d5ec75d5fe70d428685510fae36492d9: FOUND: p@ssw0rd!
```

Lookup tables are an extremely effective method for cracking many hashes of the same type very quickly. The general idea is to **pre-compute** the hashes of the passwords in a password dictionary and store them, and their corresponding password, in a lookup table data structure. A good implementation of a lookup table can process hundreds of hash lookups per second, even when they contain many billions of hashes.

If you want a better idea of how fast lookup tables can be, try cracking the following sha256 hashes with CrackStation's free hash cracker.

```
c11083b4b0a7743af748c85d343dfee9fbb8b2576c05f3a7f0d632b0926aadfc
08eac03b80adc33dc7d8fbe44b7c7b05d3a2c511166bdb43fcb710b03ba919e7
e4ba5cbd251c98e6cd1c23f126a3b81d8d8328abc95387229850952b3ef9f904
5206b8b8a996cf5320cb12ca91c7b790fba9f030408efe83ebb83548dc3007bd
```

Reverse Lookup Tables

```
Searching for hash(apple) in users' hash list...     : Matches [alice3, 0bob0, charles8]
Searching for hash(blueberry) in users' hash list... : Matches [usr10101, timmy, john91]
Searching for hash(letmein) in users' hash list...   : Matches [wilson10, dragonslayerX,
joe1984]
Searching for hash(s3cr3t) in users' hash list...    : Matches [bruce19, knuth1337, john87]
Searching for hash(z@29hjja) in users' hash list...  : No users used this password
```

This attack allows an attacker to apply a dictionary or brute-force attack to many hashes at the same time, without having to pre-compute a lookup table.

First, the attacker creates a lookup table that maps each password hash from the compromised user account database to a list of users who had that hash. The attacker then hashes each password guess and uses the lookup table to get a list of users whose password was the attacker's guess. This attack is especially effective because it is common for many users to have the same password.

- **Rainbow Tables**

  Rainbow tables are a time-memory trade-off technique. They are like lookup tables, except that they sacrifice hash cracking speed to make the lookup tables smaller. Because they are smaller, the solutions to more hashes can be stored in the same amount of space, making them more effective. Rainbow tables that can crack any md5 hash of a password up to 8 characters long exist.

Next, we'll look at a technique called salting, which makes it impossible to use lookup tables and rainbow tables to crack a hash.

Adding Salt

```
hash("hello")                    = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
hash("hello" + "QxLUF1bgIAdeQX") = 9e209040c863f84a31e719795b2577523954739fe5ed3b58a75cff2127075ed1
hash("hello" + "bv5PehSMfV11Cd") = d1d3ec2e6f20fd420d50e2642992841d8338a314b8ea157c9e18477aaef226ab
hash("hello" + "YYLmfY6IehjZMQ") = a49670c3c18b9e079b9cfaf51634f563dc8ae3070db2c4a8544305df1b60f007
```

Lookup tables and rainbow tables only work because each password is hashed the exact same way. If two users have the same password, they'll have the same password hashes. We can prevent these attacks by randomizing each hash, so that when the same password is hashed twice, the hashes are not the same.

We can randomize the hashes by appending or prepending a random string, called a **salt**, to the password before hashing. As shown in the example above, this makes the same password hash into a completely different string every time. To check if a password is correct, we need the salt, so it is usually stored in the user account database along with the hash, or as part of the hash string itself.

The salt does not need to be secret. Just by randomizing the hashes, lookup tables, reverse lookup tables, and rainbow tables become ineffective. An attacker won't know in advance what the salt will be, so they can't pre-compute a lookup table or rainbow table. If each user's password is hashed with a different salt, the reverse lookup table attack won't work either.

In the next section, we'll look at how salt is commonly implemented incorrectly.

The WRONG Way: Short Salt & Salt Reuse

The most common salt implementation errors are reusing the same salt in multiple hashes, or using a salt that is too short.

Salt Reuse

A common mistake is to use the same salt in each hash. Either the salt is hard-coded into the program, or is generated randomly once. This is ineffective because if two users have the same password, they'll still have the same hash. An attacker can still use a reverse lookup table attack to run a dictionary attack on every hash at the same time. They just have to apply the salt to each password guess before they hash it. If the salt is hard-coded into a popular product, lookup tables and rainbow tables can be built for that salt, to make it easier to crack hashes generated by the product.

A new random salt must be generated each time a user creates an account or changes their password.

Short Salt

If the salt is too short, an attacker can build a lookup table for every possible salt. For example, if the salt is only three ASCII characters, there are only 95x95x95 = 857,375 possible salts. That may seem like a lot, but if each lookup table contains only 1MB of the most common passwords, collectively they will be only 837GB, which is not a lot considering 1000GB hard drives can be bought for under $100 today.

For the same reason, the username shouldn't be used as a salt. Usernames may be unique to a single service, but they are predictable and often reused for accounts on other services. An attacker can build lookup tables for common usernames and use them to crack username-salted hashes.

To make it impossible for an attacker to create a lookup table for every possible salt, the salt must be long. A good rule of thumb is to use a salt that is the same size as the output of the hash function. For example, the output of SHA256 is 256 bits (32 bytes), so the salt should be at least 32 random bytes.

The WRONG Way: Double Hashing & Wacky Hash Functions

This section covers another common password hashing misconception: wacky combinations of hash algorithms. It's easy to get carried away and try to combine different hash functions, hoping that the result will be more secure. In practice, though, there is very little benefit to doing it. All it does is create interoperability problems, and can sometimes even make the hashes less secure. Never try to invent your own crypto, always use a standard that has been designed by experts. Some will argue that using multiple hash functions makes the process of computing the hash slower, so cracking is slower, but there's a better way to make the cracking process slower as we'll see later.

Here are some examples of poor wacky hash functions I've seen suggested in forums on the internet.

- `md5(sha1(password))`
- `md5(md5(salt) + md5(password))`
- `sha1(sha1(password))`
- `sha1(str_rot13(password + salt))`
- `md5(sha1(md5(md5(password) + sha1(password)) + md5(password)))`

Do not use any of these.

Note: This section has proven to be controversial. I've received a number of emails arguing that wacky hash functions are a good thing, because it's better if the attacker doesn't know which hash function is in use, it's less likely for an attacker to have pre-computed a rainbow table for the wacky hash function, and it takes longer to compute the hash function.

An attacker cannot attack a hash when he doesn't know the algorithm, but note Kerckhoffs's principle, that the attacker will usually have access to the source code (especially if it's free or open source software), and that given a few password-hash pairs from the target system, it is not difficult to reverse engineer the algorithm. It does take longer to compute wacky hash functions, but only by a small constant factor. It's better to use an iterated algorithm that's designed to be extremely hard to parallelize (these are discussed below). And, properly salting the hash solves the rainbow table problem.

If you really want to use a standardized "wacky" hash function like HMAC, then it's OK. But if your reason for doing so is to make the hash computation slower, read the section below about key stretching first.

Compare these minor benefits to the risks of accidentally implementing a completely insecure hash function and the interoperability problems wacky hashes create. It's clearly best to use a standard and well-tested algorithm.

Hash Collisions

Because hash functions map arbitrary amounts of data to fixed-length strings, there must be some inputs that hash into the same string. Cryptographic hash functions are designed to make these collisions incredibly difficult to find. From time to time, cryptographers find "attacks" on hash functions that make finding collisions easier. A recent example is the MD5 hash function, for which collisions have actually been found.

Collision attacks are a sign that it may be more likely for a string other than the user's password to have the same hash. However, finding collisions in even a weak hash function like MD5 requires a lot of dedicated computing power, so it is very unlikely that these collisions will happen "by accident" in practice. A password hashed using MD5 and salt is, for all practical purposes, just as secure as if it were hashed with SHA256 and salt. Nevertheless, it is a good idea to use a more secure hash function like SHA256, SHA512, RipeMD, or WHIRLPOOL if possible.

The RIGHT Way: How to Hash Properly

This section describes exactly how passwords should be hashed. The first subsection covers the basics—everything that is absolutely necessary. The following subsections explain how the basics can be augmented to make the hashes even harder to crack.

The Basics: Hashing with Salt

Warning: Do not just read this section. You absolutely must implement the stuff in the next section: "Making Password Cracking Harder: Slow Hash Functions".

We've seen how malicious hackers can crack plain hashes very quickly using lookup tables and rainbow tables. We've learned that randomizing the hashing using salt is the solution to the problem. But how do we generate the salt, and how do we apply it to the password?

Salt should be generated using a **Cryptographically Secure Pseudo-Random Number Generator** (CSPRNG). CSPRNGs are very different than ordinary pseudo-random number generators, like the "C" language's `rand()` function. As the name suggests, CSPRNGs are designed to be cryptographically secure, meaning they provide a high level of randomness and are completely unpredictable. We don't want our salts to be predictable, so we must use a CSPRNG. The following table lists some CSPRNGs that exist for some popular programming platforms.

Platform	CSPRNG
PHP	mcrypt_create_iv, openssl_random_pseudo_bytes
Java	java.security.SecureRandom
Dot NET (C#, VB)	System.Security.Cryptography.RNGCryptoServiceProvider

Ruby	SecureRandom
Python	secrets
Perl	Math::Random::Secure
C/C++ (Windows API)	CryptGenRandom
Any language on GNU/Linux or Unix	Read from /dev/random or /dev/urandom

The salt needs to be unique per-user per-password. Every time a user creates an account or changes their password, the password should be hashed using a new random salt. Never reuse a salt. The salt also needs to be long, so that there are many possible salts. As a rule of thumb, make your salt is at least as long as the hash function's output. The salt should be stored in the user account table alongside the hash.

To Store a Password

1. Generate a long random salt using a CSPRNG.
2. Prepend the salt to the password and hash it with a **standard** password hashing function like Argon2, bcrypt, scrypt, or PBKDF2.
3. Save both the salt and the hash in the user's database record.

To Validate a Password

1. Retrieve the user's salt and hash from the database.
2. Prepend the salt to the given password and hash it using the same hash function.
3. Compare the hash of the given password with the hash from the database. If they match, the password is correct. Otherwise, the password is incorrect.

In a Web Application, always hash on the server

If you are writing a web application, you might wonder *where* to hash. Should the password be hashed in the user's browser with JavaScript, or should it be sent to the server "in the clear" and hashed there?

Even if you are hashing the user's passwords in JavaScript, you still have to hash the hashes on the server. Consider a website that hashes users' passwords in the user's browser without hashing the hashes on the server. To authenticate a user, this website will accept a hash from the browser and check if that hash exactly matches the one in the database. This seems more secure than just hashing on the server, since the users' passwords are never sent to the server, but it's not.

The problem is that the client-side hash logically *becomes* the user's password. All the user needs to do to authenticate is tell the server the hash of their password. If a bad guy got a user's *hash* they could use it to authenticate to the server, without knowing the user's password! So, if the bad guy somehow steals the database of hashes from this hypothetical website, they'll have immediate access to everyone's accounts without having to guess any passwords.

This isn't to say that you *shouldn't* hash in the browser, but if you do, you absolutely have to hash on the server too. Hashing in the browser is certainly a good idea, but consider the following points for your implementation:

- Client-side password hashing is **not** a substitute for HTTPS (SSL/TLS). If the connection between the browser and the server is insecure, a man-in-the-middle can modify the JavaScript code as it is downloaded to remove the hashing functionality and get the user's password.

- Some web browsers don't support JavaScript, and some users disable JavaScript in their browser. So for maximum compatibility, your app should detect whether or not the browser supports JavaScript and emulate the client-side hash on the server if it doesn't.

- You need to salt the client-side hashes too. The obvious solution is to make the client-side script ask the server for the user's salt. Don't do that, because it lets the bad guys check if a username is valid without knowing the password. Since you're hashing and salting (with a good salt) on the server too, it's OK to use the username (or email) concatenated with a site-specific string (e.g. domain name) as the client-side salt.

Making Password Cracking Harder: Slow Hash Functions

Salt ensures that attackers can't use specialized attacks like lookup tables and rainbow tables to crack large collections of hashes quickly, but it doesn't prevent them from running dictionary or brute-force attacks on each hash individually. High-end graphics cards (GPUs) and custom hardware can compute billions of hashes per second, so these attacks are still very effective. To make these attacks less effective, we can use a technique known as **key stretching**.

The idea is to make the hash function very slow, so that even with a fast GPU or custom hardware, dictionary and brute-force attacks are too slow to be worthwhile. The goal is to make the hash function slow enough to impede attacks, but still fast enough to not cause a noticeable delay for the user.

Key stretching is implemented using a special type of CPU-intensive hash function. Don't try to invent your own– simply iteratively hashing the hash of the password isn't enough as it can be parallelized in hardware and executed as fast as a normal hash. Use a standard algorithm like PBKDF2 or bcrypt. You can find a PHP implementation of PBKDF2 here.

These algorithms take a security factor or iteration count as an argument. This value determines how slow the hash function will be. For desktop software or smartphone apps, the best way to choose this parameter is to run a short benchmark on the device to find the value that makes the hash take about half a second. This way, your program can be as secure as possible without affecting the user experience.

If you use a key stretching hash in a web application, be aware that you will need extra computational resources to process large volumes of authentication requests, and that key stretching may make it easier to run a Denial of Service (DoS) attack on your website. I still recommend using key stretching, but with a lower iteration count. You should calculate the iteration count based on your computational resources and the expected maximum authentication request rate. The denial of service threat can be eliminated by making the user solve a CAPTCHA every time they log in. Always design your system so that the iteration count can be increased or decreased in the future.

If you are worried about the computational burden, but still want to use key stretching in a web application, consider running the key stretching algorithm in the user's browser with JavaScript. The Stanford JavaScript Crypto Library includes PBKDF2. The iteration count should be set low enough that the system is usable with slower clients like mobile devices, and the system should fall back to server-side computation if the user's browser doesn't support JavaScript. Client-side key stretching does not remove the need for server-side hashing. You must hash the hash generated by the client the same way you would hash a normal password.

Impossible-to-crack Hashes: Keyed Hashes and Password Hashing Hardware

As long as an attacker can use a hash to check whether a password guess is right or wrong, they can run a dictionary or brute-force attack on the hash. The next step is to add a **secret key** to the hash so that only someone who knows the key can use the hash to validate a password. This can be accomplished two ways. Either the hash can be encrypted using a cipher like AES, or the secret key can be included in the hash using a keyed hash algorithm like HMAC.

This is not as easy as it sounds. The key has to be kept secret from an attacker even in the event of a breach. If an attacker gains full access to the system, they'll be able to steal the key no matter where it is stored. The key must be stored in an external system, such as a physically separate server dedicated to password validation, or a special hardware device attached to the server such as the YubiHSM.

I highly recommend this approach for any large scale (more than 100,000 users) service. I consider it necessary for any service hosting more than 1,000,000 user accounts.

If you can't afford multiple dedicated servers or special hardware devices, you can still get some of the benefits of keyed hashes on a standard web server. Most databases are breached using SQL Injection Attacks, which, in most cases, don't give attackers access to the local filesystem (disable local filesystem access in your SQL server if it has this feature). If you generate a random key and store it in a file that isn't accessible from the web, and include it into the salted hashes, then the hashes won't be vulnerable if your database is breached using a simple SQL injection attack. Don't hard-code a key into the source code, generate it randomly when the application is installed. This isn't as secure as using a separate system to do the password hashing, because if there are SQL injection vulnerabilities in a web application, there are probably other types, such as Local File Inclusion, that an attacker could use to read the secret key file. But, it's better than nothing.

Please note that keyed hashes do not remove the need for salt. Clever attackers will eventually find ways to compromise the keys, so it is important that hashes are still protected by salt and key stretching.

Other Security Measures

Password hashing protects passwords in the event of a security breach. It does not make the application as a whole more secure. Much more must be done to prevent the password hashes (and other user data) from being stolen in the first place.

Even experienced developers must be educated in security in order to write secure applications. A great resource for learning about web application vulnerabilities is The Open Web Application Security Project (OWASP). A good introduction is the OWASP Top Ten Vulnerability List. Unless you understand all the vulnerabilities on the list, do not attempt to write a web application that deals with sensitive data. It is the employer's responsibility to ensure all developers are adequately trained in secure application development.

Having a third party "penetration test" your application is a good idea. Even the best programmers make mistakes, so it always makes sense to have a security expert review the code for potential vulnerabilities. Find a trustworthy organization (or hire staff) to review your code on a regular basis. The security review process should begin early in an application's life and continue throughout its development.

It is also important to monitor your website to detect a breach if one does occur. I recommend hiring at least one person whose full time job is detecting and responding to security breaches. If a breach goes undetected, the attacker can make your website infect visitors with malware, so it is extremely important that breaches are detected and responded to promptly.

Frequently Asked Questions

What hash algorithm should I use?

DO use:

- Well-designed key stretching algorithms such as PBKDF2, bcrypt, and scrypt.
- OpenWall's Portable PHP password hashing framework
- My implementations of PBKDF2 in PHP, C#, Java, and Ruby.
- Secure versions of crypt ($2y$, 5, 6)

DO NOT use:

- Fast cryptographic hash functions such as MD5, SHA1, SHA256, SHA512, RipeMD, WHIRLPOOL, SHA3, etc.
- Insecure versions of crypt (1, 2, $2x$, 3).
- Any algorithm that you designed yourself. Only use technology that is in the public domain and has been well-tested by experienced cryptographers.

Even though there are no cryptographic attacks on MD5 or SHA1 that make their hashes easier to crack, they are old and are widely considered (somewhat incorrectly) to be inadequate for password storage. So I don't recommend using them. An exception to this rule is PBKDF2, which is frequently implemented using SHA1 as the underlying hash function.

How should I allow users to reset their password when they forget it?

It is my personal opinion that all password reset mechanisms in widespread use today are insecure. If you have high security requirements, such as an encryption service would, do not let the user reset their password.

Most websites use an email loop to authenticate users who have forgotten their password. To do this, generate a random **single-use** token that is strongly tied to the account. Include it in a password reset link sent to the user's email address. When the user clicks a password reset link containing a valid token, prompt them for a new password. Be sure that the token is strongly tied to the user account so that an attacker can't use a token sent to his own email address to reset a different user's password.

The token must be set to expire in 15 minutes or after it is used, whichever comes first. It is also a good idea to expire any existing password tokens when the user logs in (they remembered their password) or requests another reset token. If a token doesn't expire, it can be forever used to break into the user's account. Email (SMTP) is a plain-text protocol, and there may be malicious routers on the internet recording email traffic. And, a user's email account (including the reset link) may be compromised long after their password has been changed. Making the token expire as soon as possible reduces the user's exposure to these attacks.

Attackers will be able to modify the tokens, so don't store the user account information or timeout information in them. They should be an unpredictable random binary blob used only to identify a record in a database table.

Never send the user a new password over email. Remember to pick a new random salt when the user resets their password. Don't re-use the one that was used to hash their old password.

What should I do if my user account database gets leaked/hacked?

Your first priority is to determine how the system was compromised and patch the vulnerability the attacker used to get in. If you do not have experience responding to breaches, I highly recommend hiring a third-party security firm.

It may be tempting to cover up the breach and hope nobody notices. However, trying to cover up a breach makes you look worse, because you're putting your users at further risk by not informing them that their passwords and other personal information may be compromised. You must inform your users as soon as possible—even if you don't yet fully understand what happened. Put a notice on the front page of your website that links to a page with more detailed information, and send a notice to each user by email if possible.

Explain to your users exactly how their passwords were protected—hopefully hashed with salt—and that even though they were protected with a salted hash, a malicious hacker can still run dictionary and brute force attacks on the hashes. Malicious hackers will use any passwords they find to try to login to a user's account on a different website, hoping they used the same password on both websites. Inform your users of this risk and recommend that they change their password on any website or service where they used a similar password. Force them to change their password for your service the next time they log in. Most users will try to "change" their password to the original password to get around the forced change quickly. Use the current password hash to ensure that they cannot do this.

It is likely, even with salted slow hashes, that an attacker will be able to crack some of the weak passwords very quickly. To reduce the attacker's window of opportunity to use these passwords, you should require, in addition to the current password, an email loop for authentication until the user has changed their password. See the previous question, "How should I allow users to reset their password when they forget it?" for tips on implementing email loop authentication.

Also tell your users what kind of personal information was stored on the website. If your database includes credit card numbers, you should instruct your users to look over their recent and future bills closely and cancel their credit card.

What should my password policy be? Should I enforce strong passwords?

If your service doesn't have strict security requirements, then don't limit your users. I recommend showing users information about the strength of their password as they type it, letting them decide how secure they want their password to be. If you have special security needs, enforce a minimum length of 12 characters and require at least two letters, two digits, and two symbols.

Do not force your users to change their password more often than once every six months, as doing so creates "user fatigue" and makes users less likely to choose good passwords. Instead, train users to change their password whenever they feel it has been compromised, and to never tell their password to anyone. If it is a business setting, encourage employees to use paid time to memorize and practice their password.

If an attacker has access to my database, can't they just replace the hash of my password with their own hash and login?

Yes, but if someone has access to your database, they probably already have access to everything on your server, so they wouldn't need to login to your account to get what they want. The purpose of password hashing (in the context of a website) is not to protect the website from being breached, but to protect the passwords if a breach does occur.

You can prevent hashes from being replaced during a SQL injection attack by connecting to the database with two users with different permissions. One for the 'create account' code and one for the 'login' code. The 'create account' code should be able to read and write to the user table, but the 'login' code should only be able to read.

Why do I have to use a special algorithm like HMAC? Why can't I just append the password to the secret key?

Hash functions like MD5, SHA1, and SHA2 use the Merkle–Damgård construction, which makes them vulnerable to what are known as length extension attacks. This means that given a hash H(X), an attacker can find the value of H(pad(X) + Y), for any other string Y, without knowing X. pad(X) is the padding function used by the hash.

This means that given a hash H(key + message), an attacker can compute H(pad(key + message) + extension), without knowing the key. If the hash was being used as a message authentication code, using the key to prevent an attacker from being able to modify the message and replace it with a different valid hash, the system has failed, since the attacker now has a valid hash of message + extension.

It is not clear how an attacker could use this attack to crack a password hash quicker. However, because of the attack, it is considered bad practice to use a plain hash function for keyed hashing. A clever cryptographer may one day come up with a clever way to use these attacks to make cracking faster, so use HMAC.

Should the salt come before or after the password?

It doesn't matter, but pick one and stick with it for interoperability's sake. Having the salt come before the password seems to be more common.

Why does the hashing code on this page compare the hashes in "length-constant" time?

Comparing the hashes in "length-constant" time ensures that an attacker cannot extract the hash of a password in an on-line system using a timing attack, then crack it off-line.

The standard way to check if two sequences of bytes (strings) are the same is to compare the first byte, then the second, then the third, and so on. As soon as you find a byte that isn't the same for both strings, you know they are different and can return a negative response immediately. If you make it through both strings without finding any bytes that differ, you know the strings are the same and can return a positive result. This means that comparing two strings can take a different amount of time depending on how much of the strings match.

For example, a standard comparison of the strings "xyzabc" and "abcxyz" would immediately see that the first character is different and wouldn't bother to check the rest of the string. On the other hand, when the strings "aaaaaaaaaaB" and "aaaaaaaaaaZ" are compared, the comparison algorithm scans through the block of "a" before it determines the strings are unequal.

Suppose an attacker wants to break into an on-line system that rate limits authentication attempts to one attempt per second. Also suppose the attacker knows all of the parameters to the password hash (salt, hash type, etc), except for the hash and (obviously) the password. If the attacker can get a precise measurement of how long it takes the on-line system to compare the hash of the real password with the hash of a password the attacker provides, he can use the timing attack to extract part of the hash and crack it using an offline attack, bypassing the system's rate limiting.

First, the attacker finds 256 strings whose hashes begin with every possible byte. He sends each string to the on-line system, recording the amount of time it takes the system to respond. The string that takes the longest will be the one whose hash's first byte matches the real hash's first byte. The attacker now knows the first byte, and can continue the attack in a similar manner on the second byte, then the third, and so on. Once the attacker knows enough of the hash, he can use his own hardware to crack it, without being rate limited by the system.

It might seem like it would be impossible to run a timing attack over a network. However, it has been done, and has been shown to be practical. That's why the code on this page compares strings in a way that takes the same amount of time no matter how much of the strings match.

How does the SlowEquals code work?

The previous question explains why SlowEquals is necessary, this one explains how the code actually works.

```
private static boolean slowEquals(byte[] a, byte[] b)
{
    int diff = a.length ^ b.length;
    for(int i = 0; i < a.length && i < b.length; i++)
        diff |= a[i] ^ b[i];
    return diff == 0;
}
```

The code uses the XOR "^" operator to compare integers for equality, instead of the "==" operator. The reason why is explained below. The result of XORing two integers will be zero if and only if they are exactly the same. This is because 0 XOR 0 = 0, 1 XOR 1 = 0, 0 XOR 1 = 1, 1 XOR 0 = 1. If we apply that to all the bits in both integers, the result will be zero only if all the bits matched.

So, in the first line, if `a.length` is equal to `b.length`, the diff variable will get a zero value, but if not, it will get some non-zero value. Next, we compare the bytes using XOR, and OR the result into diff. This will set diff to a non-zero value if the bytes differ. Because ORing never un-sets bits, the only way diff will be zero at the end of the loop is if it was zero before the loop began (a.length == b.length) and all of the bytes in the two arrays match (none of the XORs resulted in a non-zero value).

The reason we need to use XOR instead of the "==" operator to compare integers is that "==" is usually translated/compiled/interpreted as a branch. For example, the C code "`diff &= a == b`" might compile to the following x86 assembly:

```
MOV EAX, [A]
CMP [B], EAX
JZ equal
JMP done
equal:
AND [VALID], 1
done:
AND [VALID], 0
```

The branching makes the code execute in a different amount of time depending on the equality of the integers and the CPU's internal branch prediction state.

The C code "`diff |= a ^ b`" should compile to something like the following, whose execution time does not depend on the equality of the integers:

```
MOV EAX, [A]
XOR EAX, [B]
OR [DIFF], EAX
```

Why bother hashing?

Your users are entering their password into your website. They are trusting you with their security. If your database gets hacked, and your users' passwords are unprotected, then malicious hackers can use those passwords to compromise your users' accounts on other websites and services (most people use the same password everywhere). It's not just your security that's at risk, it's your users'. You are responsible for your users' security.

Article written by Defuse Security.