Fichier 1 : explications sur les salted password

Salting hashes sounds like one of the steps of a hash browns recipe, but in cryptography, the expression refers to adding random data to the input of a hash function to guarantee a unique output, the hash, even when the inputs are the same. Consequently, the unique hash produced by adding the salt can protect us against different attack vectors, such as hash table attacks, while slowing down dictionary and brute-force offline attacks.

However, there are limitations in the protections that a salt can provide. If the attacker is hitting an online service with a credential stuffing attack, a subset of the brute force attack category, salts won't help at all because the legitimate server is doing the salting+hashing for you.

Note: Never tell anyone using your registration forms that their selected password is not unique. A system like that in place will allow hackers to crack passwords in record time!

Hashed passwords are not unique to themselves due to the deterministic nature of hash function: when given the same input, the same output is always produced. If Alice and Bob both choose dontpwnme4 as a password, their hash would be the same:

| username | hash |
| --- | --- |
| alice | 4420d1918bbcf7686defdf9560bb5087d20076de5f77b7cb4c3b40bf46ec428b |
| jason | 695ddccd984217fe8d79858dc485b67d66489145afa78e8b27c1451b27cc7a2b |
| mario | cd5cb49b8b62fb8dca38ff2503798eae71bfb87b0ce3210cf0acac43a3f2883c |
| teresa | 73fb51a0c9be7d988355706b18374e775b18707a8a03f7a61198eefc64b409e8 |
| bob | 4420d1918bbcf7686defdf9560bb5087d20076de5f77b7cb4c3b40bf46ec428b |
| mike | 77b177de23f81d37b5b4495046b227befa4546db63cfe6fe541fc4c3cd216eb9 |

As we can see, alice and bob have the same password as we can see that both share the same hash: 4420d1918bbcf7686defdf9560bb5087d20076de5f77b7cb4c3b40bf46ec428b.

The attacker can better predict the password that legitimately maps to that hash. Once the password is known, the same password can be used to access all the accounts that use that hash.

Can you find what is jason's password based on the hash 695ddccd984217fe8d79858dc485b67d66489145afa78e8b27c1451b27cc7a2b?

Attacker gets DB. Sees duplicate hashes. Attacker can arrive to conclusion that there's no salts or using a weak algo to hash the passwords. If they find a lot of the same hashes, sign that server has a default password and every new acct has a default password. The kinds of attacks we're talking about here are offline attacks against compromised/exfiltrated data.

## **Attacking Unsalted Passwords**

To start, the attacker could try a [dictionary attack](https://en.wikipedia.org/wiki/Dictionary_attack). Using a pre-arranged listing of words, such as the entries from the English dictionary, with their computed hash, the attacker easily compares the hashes from a stolen passwords table with every hash on the list. If a match is found, the password then can be deduced.



Two *different* hash functions can produce the same hash; however, the risk of this happening is extremely low. But, how do attackers know which hash function to use? [It's not too hard](https://security.stackexchange.com/a/180536).

Fortunately, despite choosing the same password, alice and bob chose a password that is not easily found in a dictionary: dontpwnme4. Our friend mike, on the other hand, chose friendship as his password which is a direct entry in the English dictionary. mike is at high risk of being breached through a dictionary attack; the risk for alice and bob is no different. To come up with a password such as dontpwnme4, the attacker could use special dictionaries such as [leetspeak](https://en.wikipedia.org/wiki/Leet) to crack the password.

Both dictionary attacks and brute-force attacks require the real-time computation of the hash. Since a good password hash function is *slow*, this would take a lot of time.

Dictionary -> use lists from a dictionary Brute force -> using random characters

What kind of password profiling they are trying to make.

### Cracking Unsalted Hashes with Tables

An attacker has two types of tools at disposal: hash table and rainbow table. Definition of both and how they can help with cracking table. Hash tables to be exhausted first. Additional results use a rainbow.

Hash tables = fast lookup, but long computation (if you were building one from scratch), more space. Rainbow table = slow lookup because you have to run through the hash algorithms many times, less space.

A *hash table* can make the exploitation of unsalted passwords easier. A hash table is essentially a *pre-computed* database of hashes. Dictionaries and random strings are run through a selected hash function and the input/hash mapping is stored in a table. The attacker can then simply do a password reverse lookup by using the hashes from a stolen password database.

The main difference between a hash table attack and a dictionary and brute-force attack is *pre-computation*. Hash table attacks are fast because the attacker doesn't have to spend any time computing any hashes. The trade-off for the speed gained is the immense amount of space required to host a hash table. We could say that a hash table attack is a pre-computed dictionary and/or brute-force attack.

Since time and space are limited, the attacker that designs and computes the hash table may want to process the most commonly used passwords first. Here is where alice and bob could be at a much higher risk if dontpwnme4 is in that common-password list. Large common-password databases are created using frequency analysis across passwords collected from different publicly leaked breaches.

The strength of hash tables comes from volume not computation speed and the volume is huge! Each data breach adds to this volume. For a list of companies that have been breached visit the [pwned websites list of haveibeenpwned.com](https://haveibeenpwned.com/PwnedWebsites).

*"There are often 'breaches' announced by attackers which in turn are exposed as hoaxes. There is a balance between making data searchable early and performing sufficient due diligence to establish the legitimacy of the breach."* - [Troy Hunt](https://twitter.com/troyhunt)

Faster CPUs and GPUs, distributed computations, and weak algorithms are making cracking a password much easier. However, because cracking password hashes these days is more challenging than credential stuffing, it is always a good idea to use [MFA (Multi-factor Authentication)](https://auth0.com/docs/mfa).

## **Mitigating Password Attacks with Salt**

To mitigate the damage that a hash table or a dictionary attack could do, we salt the passwords. According to [OWASP Guidelines](https://owasp.org/www-project-cheat-sheets/cheatsheets/Password_Storage_Cheat_Sheet.html#salting), a salt is a value generated by a [cryptographically secure function](https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html) that is added to the input of hash functions to create unique hashes for every input, regardless of the input not being unique. A salt makes a hash function look non-deterministic, which is good as we don't want to reveal duplicate passwords through our hashing.

Let’s say that we have password farm1990M0O and the salt f1nd1ngn3m0. We can salt that password by either appending or prepending the salt to it. For example: farm1990M0Of1nd1ngn3m0 or f1nd1ngn3m0farm1990M0O are valid salted passwords. Once the salt is added, we can then hash it. Let's see this in action:

### Prepending the Salt

Password: farm1990M0O

Salt: f1nd1ngn3m0

Salted input: f1nd1ngn3m0farm1990M0O

Hash (SHA-256): 7528ed35c6ebf7e4661a02fd98ab88d92ccf4e48a4b27338fcc194b90ae8855c

### Appending the Salt

Password: farm1990M0O

Salt: f1nd1ngn3m0

Salted input: farm1990M0Of1nd1ngn3m0

Hash (SHA-256): 07dbb6e6832da0841dd79701200e4b179f1a94a7b3dd26f612817f3c03117434

The hashes were calculated using the following Python code:

import hashlib

string = "saltedpassword"

hashlib.sha256(string.encode()).hexdigest()

This demonstrates the importance of using unique salts. Let’s say that we decide to always append the salt to the passwords. If two users use the same password, we are just creating longer passwords that won’t be unique in our database. Both salted passwords would hash to the *same value*. But, if we choose another salt for the same password, we get two unique *and* longer passwords that hash to a different value. Let's visualize this through an example:

Alice and Bob decide to use both the same password, farm1990M0O. For Alice, we'll use f1nd1ngn3m0 again as the salt. However, for Bob, we'll use f1nd1ngd0ry as the salt:

### Hashing and Salting Alice's Password

User: Alice

Password: farm1990M0O

Salt: f1nd1ngn3m0

Salted input: farm1990M0Of1nd1ngn3m0

Hash (SHA-256): 07dbb6e6832da0841dd79701200e4b179f1a94a7b3dd26f612817f3c03117434

### Hashing and Salting Bob's Password

User: Bob

Password: farm1990M0O

Salt: f1nd1ngd0ry

Salted input: farm1990M0Of1nd1ngd0ry

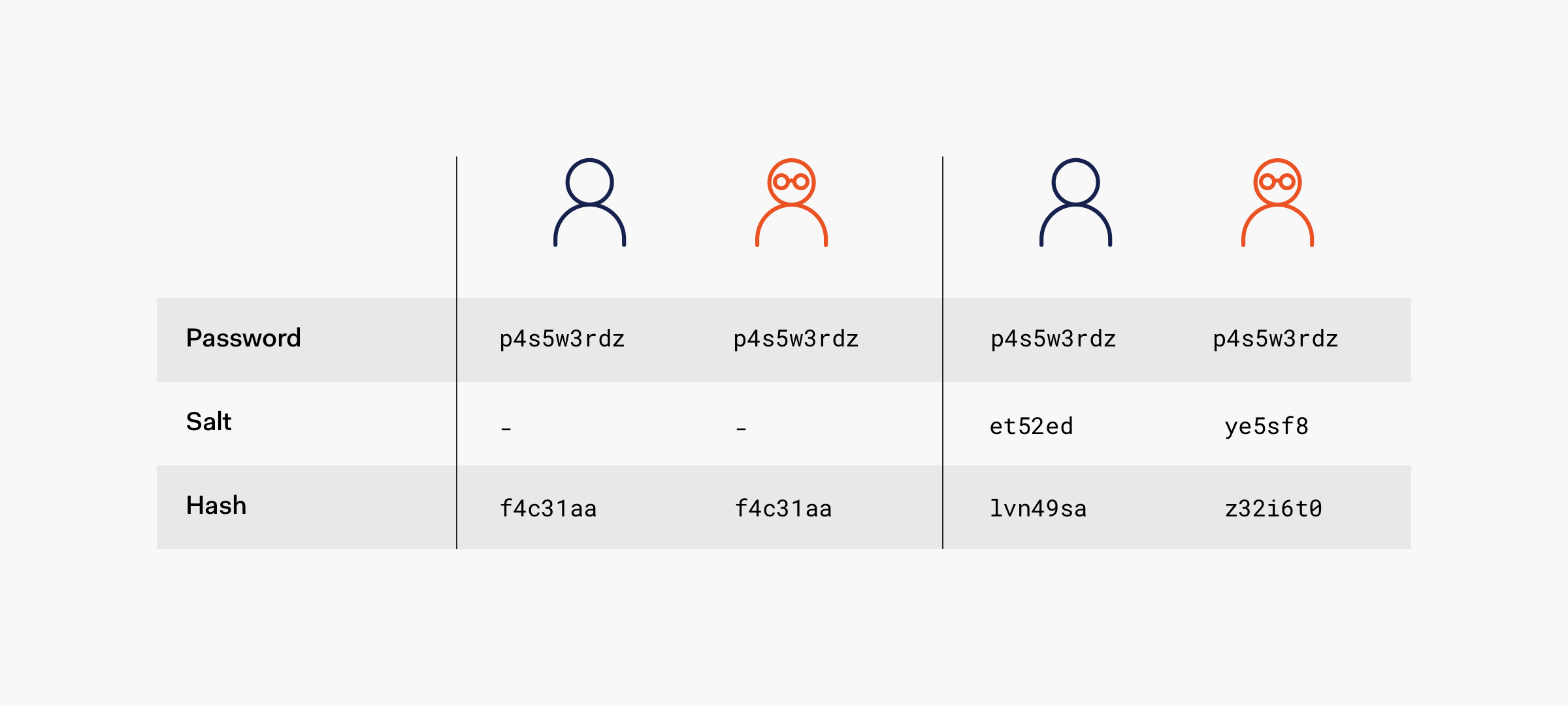
Hash (SHA-256): 11c150eb6c1b776f390be60a0a5933a2a2f8c0a0ce766ed92fea5bfd9313c8f6

Different users, same password. Different salts, different hashes. If someone looked at the full list of password hashes, no one would be able to tell that Alice and Bob both use the same password. Each unique salt extends the password farm1990M0O and transforms it into a unique password. Additionally, when a user changes their password, the service should also generate a new salt.



In practice, we store the salt in cleartext along with the hash in our database. We would store the salt f1nd1ngn3m0, the hash 07dbb6e6832da0841dd79701200e4b179f1a94a7b3dd26f612817f3c03117434, and the username together so that when the user logs in, we can lookup the username, append the salt to the provided password, hash it, and then verify if the stored hash matches the computed hash.

Now we can see why it is very important that each input is salted with unique random data. When the salt is unique for each hash, we inconvenience the attacker by now having to compute a hash table for each user hash. This creates a big bottleneck for the attacker. Ideally, we want the salt to be truly random and unpredictable to bring the attacker to a halt.



While the attacker may be able to crack one password, cracking all passwords will be unfeasible. Regardless, when a company experiences a data breach, it is impossible to determine which passwords could have been cracked and therefore all passwords must be considered compromised. A request to all users to change their passwords should be issued by the company right away. Upon password change, a new salt should be generated for each user as well.

["If someone breaches into a company database, the company must react as if all passwords were cracked, even if hashing the passwords involved using a salt."](https://twitter.com/intent/tweet?text=%22If%20someone%20breaches%20into%20a%20company%20database%2C%20the%20company%20must%20react%20as%20if%20all%20passwords%20were%20cracked%2C%20even%20if%20hashing%20the%20passwords%20involved%20using%20a%20salt.%22%20via%20@auth0%20https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords)

**Generating a Good Random Salt**

Is f1nd1ngn3m0 a good salt? When we are adding salts to passwords, we need to add salts that are cryptographically strong and credential-specific.

Following [OWASP Guidelines](https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#salting), to properly implement credential-specific salts, we must:

A system-wide salt is pointless to mitigate attacks; it would just make passwords longer. A system-wide salt also easily allows an attacker to keep using hash tables. We should hash and salt each password created for a user. That is, we should generate a unique salt upon creation of each stored credential (not just per user or system-wide). That includes passwords created during registration or as the result of a password reset. If the user eventually cycles over the same password, we don't want to give away that the password has already been used.

[Cryptographically strong or strong cryptography](https://en.wikipedia.org/wiki/Strong_cryptography) define a cryptographic system that is highly resistant to [cryptanalysis](https://en.wikipedia.org/wiki/Cryptanalysis), which are efforts to decipher the secret patterns of a system. Showing that a cryptographic scheme is resistant to attacks is a complex process that requires a lot of time, extensive testing, reviews, and community engagement. Due to this complexity, security experts recommend that [you don't roll your own cryptography](https://motherboard.vice.com/en_us/article/wnx8nq/why-you-dont-roll-your-own-crypto).

To create such cryptographically-strong random data, we may use a [cryptographically secure pseudorandom number generator](https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator)(CSPRNG) to gather unpredictable input from sources that we cannot observe, such as the Random Generator API of our operating system. Even better, we could use a battle-tested, cryptographic library for that. Most of these libraries include facilities for working with random numbers. As an advice, never roll your own random number generators.

OWASP suggests [SecureRandom](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/security/SecureRandom.html) as an example of cryptographically-strong random data.

As storage permits, use a 32-byte or 64-byte salt with the actual size dependent on the protection function. A longer salt effectively increases the computational complexity of attacking passwords which in turn increases the candidate set exponentially. A longer salt also increases the space required to store hash tables while decreasing the possibility that such a table exists in the wild.

The security of this scheme does not depend on hiding, splitting, or otherwise obscuring the salt. Simply put, *do not mess with the salt*. The salt doesn't need to be encrypted, for example. Salts are in place to prevent someone from cracking passwords at large and can be stored in cleartext in the database. However, do not make the salts readily accessible to the public. For that reason, usernames are bad candidates to use as salts.

["Hashing salts are speed bumps in an attacker's road to breaching your data. It does not matter if they are visible and unencrypted, what matters is that they are in place."](https://twitter.com/intent/tweet?text=%22Hashing%20salts%20are%20speed%20bumps%20in%20an%20attacker%27s%20road%20to%20breaching%20your%20data.%20It%20does%20not%20matter%20if%20they%20are%20visible%20and%20unencrypted%2C%20what%20matters%20is%20that%20they%20are%20in%20place.%22%20via%20@auth0%20https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords)

Based on these guidelines, f1nd1ngn3m0 is not being generated from an unpredictable source. In fact, the salt is a 1337 way of writing findingnemo, a popular animated movie, which could be part of a dictionary-brute-force strategy. f1nd1ngn3m0 doesn't meet the length recommendation to be a salt: it's only 11 bytes long.

Our second salt, f1nd1ngd0ry suffers from the same weaknesses. I chose it based on it being the sequence to the "Finding Nemo" movie, "Finding Dory". Our human imagination to create randomness can only go so far so it's better to delegate that task to the machine.

As we can see, hashing and salting are very complex processes and the security of our systems greatly relies on their successful implementation. While these are no methods to create 100% secure systems, these are methods to create hardy and resilient systems. It's best to leave the creation, maintenance, and operation of such methods and systems to security experts. A misstep in your home-made security strategy may lead to extensive damage to your business, users, and reputation.

["Hashing and salting are complex methods to create hardy and resilient systems. It's best to leave their implementation to security experts. A misstep in a home-made security strategy may lead to damage to a business, its users, and reputation."](https://twitter.com/intent/tweet?text=%22Hashing%20and%20salting%20are%20complex%20methods%20to%20create%20hardy%20and%20resilient%20systems.%20It%27s%20best%20to%20leave%20their%20implementation%20to%20security%20experts.%20A%20misstep%20in%20a%20home-made%20security%20strategy%20may%20lead%20to%20damage%20to%20a%20business%2C%20its%20users%2C%20and%20reputation.%22%20via%20@auth0%20https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords)

You'd want to rely on algorithms such as bcrypt that hash and salt the password for you using strong cryptography. Additionally, you may use a security framework, such as [Spring Security](https://projects.spring.io/spring-security/) for the Java Ecosystem for example. These frameworks offer you abstractions that make the development of your applications safer but also [integrate with reliable identity providers](https://auth0.com/docs/quickstart/webapp/java-spring-security-mvc/01-login), such as Auth0, that make Identity and Access Management much easier.

source : https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/