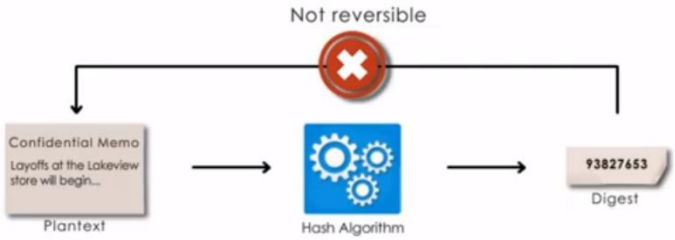
What is a Hashcode?

A **hashcode** is a fixed length code that may be used to verify data integrity, authenticate messages or look up resources. Hashcodes are also a fundamental aspect of encryption. For example, passwords are typically stored as hashcodes.



A class of algorithms known as a hash function can be used to create fixed-length hashcodes from messages of any length.

Ideally, these algorithms have several unique properties:

- **Non-Reversible**

- It is impossible generate a message from its hashcode. Hash functions are known as trap doors because they only go one way from message->hashcode and not the reverse.

- **Avalanche Effect**

- A small change in a message leads to large change in the hashcode. This is known as the avalanche effect.

- **Hash Collision**

- It is exceedingly unlikely that two messages will generate the same hashcode. When this does happen, it is known as hash collision.

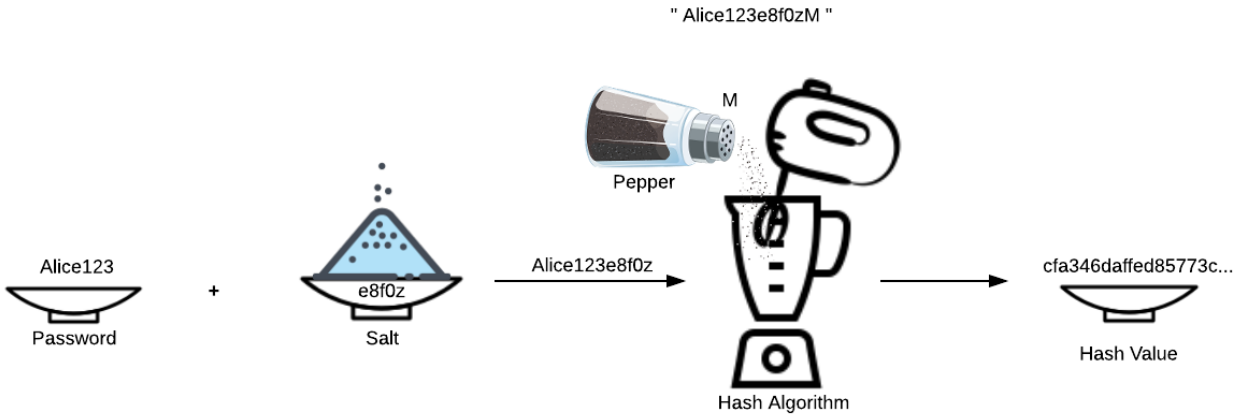
Examples:

An encryption tool stores hashcodes of passwords as opposed to the passwords themselves. If the hashcodes are compromised, the passwords will remain confidential.

What is Cryptographic Salt & Pepper?

Salt is random data that is added to data before it is passed to a hash function. It is a cryptographic technique that makes hash codes more difficult to reverse. It is common to store the salt alongside the hash value.

Pepper is also random data that is added to data before generating a hash code. Unlike salt, pepper is kept secret. In many cases, pepper isn't stored at all. In other cases, it is securely stored separately from the hash code.



Salt, Pepper & Passwords

Passwords are typically converted to a hash value for storage on disk or a database. In this way, if an attacker accesses the passwords, they can't be used to access systems.

Reversing a hash value is extremely difficult. The most common approach that attackers take is to use a **rainbow table** that contains hash codes for common passwords. Adding salt to a password renders a rainbow table useless even if the salt is known to the attacker.

A salt is different for every user but a pepper is common to everyone in the database. Pepper is a site-wise static value.

Advantages:

1. A salt makes passwords more difficult to crack because simply salting makes passwords more complex.
2. Many users tend to use simple passwords, thus resulting in the same passwords like abc123 or password123. Salting would generate a unique password for each user, thus reducing the chance of passwords hacked in bulk.
3. Costly operation for hackers to get each users password. They need to generate the pre-calculated table for each salt individually. Thus, making the process much slower.
4. As peppers are not stored in the database they make the process of hacking even longer and tougher.

Python Code :

```
import hashlib, os

def hash_pwd(password):
    # Hash, salt to be stored in DB

    salt = hashlib.sha256(os.urandom(60)).hexdigest().encode("ascii")
    pepper = b"M"
    password = password.encode("utf-8")

    pwdhash = hashlib.sha512(password + salt + pepper).hexdigest()

    return salt.decode("ascii") + pwdhash


def verify_pwd(salt, stored_password, provided_password):
    pepper = b"M"

    pwd_hash = hashlib.sha512(provided_password.encode("utf8") + salt.encode("ascii") +
      pepper).hexdigest()

    return pwd_hash == stored_password
```

Explanation & Output :

The two functions can be used to hash the user-provided password for storage on disk or into a database (hash_pwd) and to verify the password against the stored one when a user tries to log back in (verify_pwd):

```
# Output :
#
# PS C:\Users\Darlene\Desktop\Sem 6\CSS> python salt-pepper.py
# _____________Salt & Pepper_________________
# 1. Register
# 2.Login
# Enter : 1
# Enter Username : Bob
# Enter Password : 99Pancakes
# --------------Hashing Algo-----------------
# salt: c1c5b3bc1829fe955e327072dfa1c3d66bb11801a5bbef7f7a2e161101310977
# pepper: M
# password: 99Pancakes
# pwdhash: fa7e6aa5738074b43e5c584fb2ccb1a0151f892066503191b7a46dc5d4a87c3b4e909dade60464c
e95188d73d3afb271142ffabba116927023166c424b4c1243
# ----------------------------------------
# _____________Salt & Pepper_________________
# 1. Register
# 2.Login
# Enter : 2
# Enter Username : Bob
# Enter Password : 99pancakes
# User Exists
# --------------Verifying Hash---------------
# stored_password : fa7e6aa5738074b43e5c584fb2ccb1a0151f892066503191b7a46dc5d4a87c3b4e909d
ade60464ce95188d73d3afb271142ffabba116927023166c424b4c1243
# provided_password : 99pancakes
# ----------------------------------------
# Error incorrect password!
# _____________Salt & Pepper_________________
# 1. Register
# 2.Login
# Enter : 2
# Enter Username : Bob
# Enter Password : 99Pancakes
# User Exists
# --------------Verifying Hash---------------
# stored_password : fa7e6aa5738074b43e5c584fb2ccb1a0151f892066503191b7a46dc5d4a87c3b4e909d
ade60464ce95188d73d3afb271142ffabba116927023166c424b4c1243
# provided_password : 99Pancakes
# ----------------------------------------
# Valid User!
# _____________Salt & Pepper_________________
# 1. Register
# 2.Login
# Enter : 3
# ___________________Goodbye__________________
```

How it works…

There are two functions involved here:

- **hash_pwd** : Encodes a provided password in a way that is safe to store on a database or file
- **verify_pwd** : Given an encoded password and a plain text one is provided by the user, it verifies whether the provided password matches the encoded (and thus saved) one.

hash_pwd actually does multiple things; it doesn't just hash the password. The first thing it does is generate some random salt that should be added to the password. That's just the sha256 hash of some random bytes read from os.urandom . It then extracts a string representation of the hashed salt as a set of hexadecimal numbers (hexdigest)

The salt is then provided to **sha512** together with the password itself to hash the password in a randomized way. As sha512 requires bytes as its input, the three strings (password, salt and pepper) are previously encoded in pure bytes. The salt is encoded as plain ASCII, as the hexadecimal representation of a hash will only contain the 0-9 and A-F characters. While the password is encoded as utf-8, it could contain any character.

The resulting hashcode is stored into the database (in this case DB.csv

Once you have your password, **verify_pwd** can then be used to verify provided passwords against it. So it takes three arguments: the hashed password, the salt and the new password that should be verified.
The extracted salt and the password candidate are then provided to **sha512** to compute their hash.

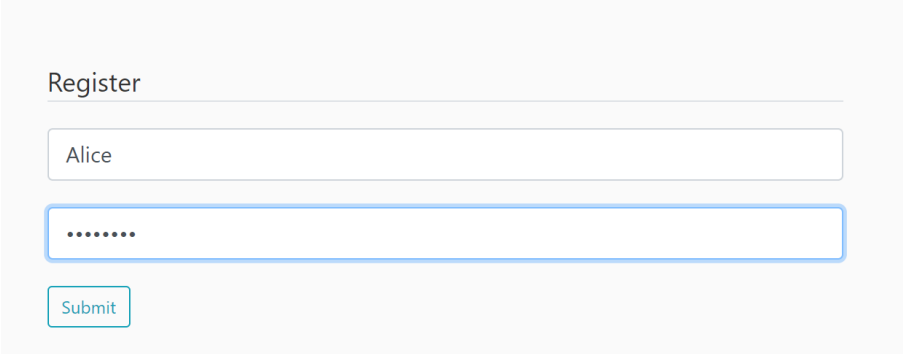
If the resulting hash matches with the previously stored hash password, it means that the two passwords match.If the resulting hash doesn't match, it means that the provided password is wrong. As you can see, it's very important that you make the salt and the password available together, because you'll need it to be able to verify the password and a different salt would result in a different hash and thus you'd never be able to verify the password.
Note:

```
"""
    os.urandom(size) :
    Size: It is the size of string random bytes
    Return Value: This method returns a string which represents random bytes suitable for
cryptographic use.

    encode() :
    Converts the string into bytes to be acceptable by hash function.

    hexdigest() :
    Returns the encoded data in hexadecimal format.
"""
```

Flask SS…



DB.csv

Hack,841d2f230e1c82353f43995ed1b58f571295d7abb8e7a212756264758ad8c353,3531c82edaf506f7b4947507c28efa37cb3c9b49142544e320451ba5cb36fa1302daf962948d5e69dd5e099007e56079123d98d854f87fe4ba07393d3dcaefca

Alice,d1f8dca9fdacc3fa4e8729e046ebe2baad7f1f88a27d1500d57da25da7e70c57,df2f05b7cc3471ac7892a0acf1219853887f994b7ce0a95e0fcdd58442949e04d9671d02bf206edd2328836eea3b274745dc9e58824d27131eb525fb2ece1244