HyperionDev

Introduction to Hashing

Task

Visit our website

Introduction

In this task, we'll be covering the concept of hashing, which is a technique used in cyber security to prevent the misuse or abuse of sensitive data. We'll explore some of the primary differences between hashing and encryption, which are sometimes mistakenly used interchangeably. The amount of data available on the Internet is vast, and this amount is constantly increasing. A large portion of that data needs to be stored in server-friendly ways and compressed. Furthermore, a large portion of that data is sensitive, and user privacy must be protected from threats and malicious actors. To safeguard data, we require fail-safe cyber security. You'll soon appreciate the value of creating unbreakable, hashed passwords and digital signatures, a practice that forms one of the cornerstones of secure digital protocol.

Hashing vs encryption

There are a few key differences between hashing and encryption, both in terms of how they work and in terms of what they are used for.

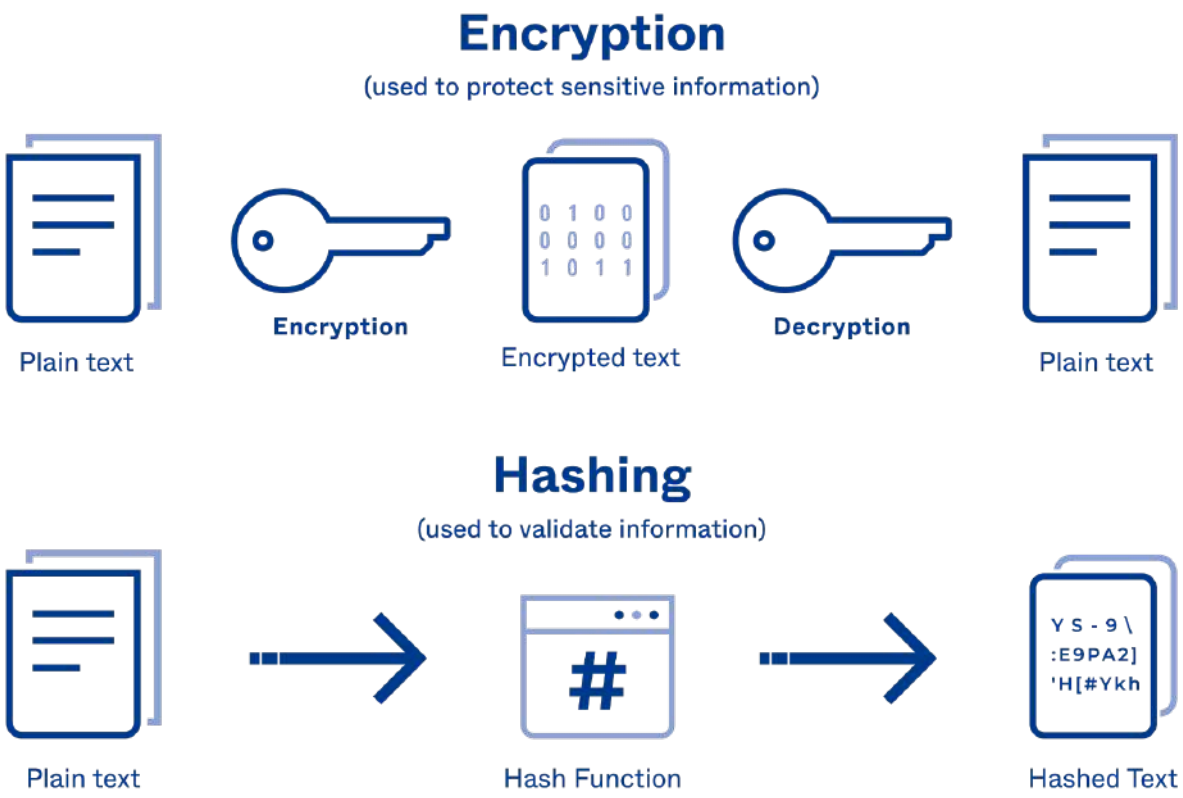
Hashing is a one-way process that converts some data or a message into a fixed-size output called a **hash** or a **hash value**. The main purpose of hashing is to create a unique representation of the original data that cannot be easily reverse-engineered. Hashing is used for a variety of purposes, including data integrity, password storage, and indexing. The primary characteristic of hashes is that they cannot be reversed.

Encryption is a two-way process that converts a message or data into a scrambled, unreadable format called **ciphertext** (or cyphertext). The original message or data, called plaintext, can be recovered by applying a decryption process using a secret key. The main purpose of encryption is to protect the confidentiality of the data by making it unreadable to anyone who does not have the key. This allows the two people at either end of the connection to be able to read each other's messages without a **man in the middle** being able to read them.

The differences between hashing and encryption are listed below:

1. **Purpose:** Hashing is used to create a unique representation of data, while encryption is used to protect the confidentiality of data.
2. **Reversibility:** Hashing is a one-way process that cannot be easily reversed, while encryption is a two-way process that is designed to be reversed using a key.
3. **Key management:** Hashing does not require the use of a key, while encryption requires the use of a secret key to encrypt and decrypt the data.

4. **Output size:** The output of a hash function is typically fixed in size, regardless of the size of the input data. The output of an encryption process is the same size as the input data.



Encryption vs hashing (Okta, 2023)

Developing secure web applications

Best practices are critical when developing secure web applications. As a developer, you are responsible for ensuring that you have done your best to ensure user data is stored and managed securely. There have been many examples over the years of user passwords being leaked (e.g., the 8.4 billion passwords leaked in the **RockYou2021** incident), resulting in extensive personal and commercial damage.

Information security is like an onion with layers. The more layers you add to your security infrastructure, the stronger your security becomes. At the core of this onion is **password hashing**.

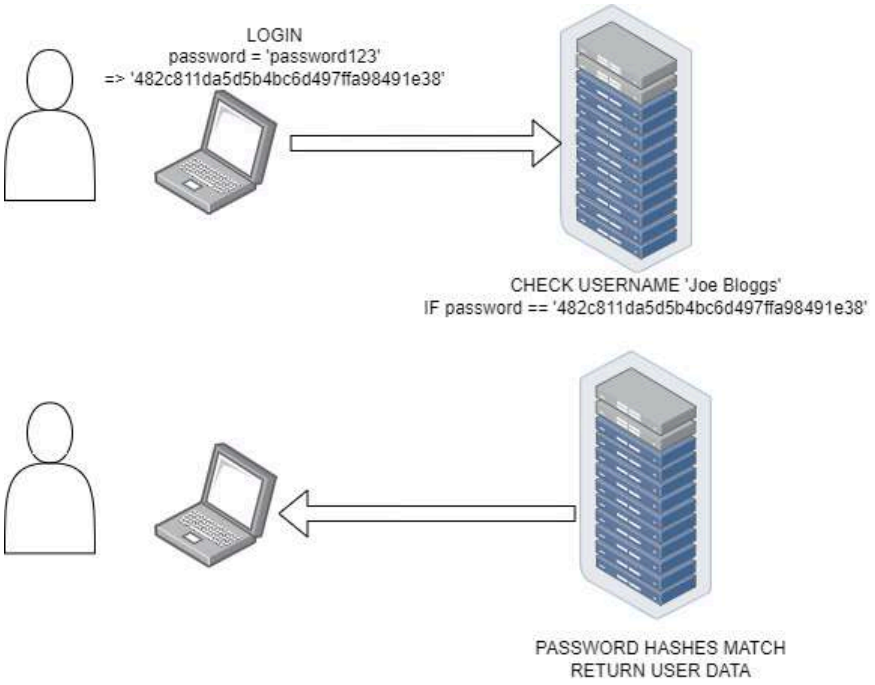
Password hashing is a way to ensure that, even if hackers pull peoples' passwords from your database, they won't be able to read them. Hashing is a specific type of cryptography that only works one way. You can hash a password, but you cannot unhash it. How does this work?

Let's pretend that we have a very simple hash function that converts each letter to its **ASCII (American Standard Code for Information Interchange)** value equivalent and then adds everything together. After this process, you'll end up with a single number. It's impossible to reconstruct the original numbers that were added up by just reading that number. This is just a very simple hashing algorithm for explanatory purposes, so this is definitely not the type of hash that you want to use for password security! There are many more complex algorithms, such as **SHA-256** and **MD5**, that are much better suited to the task. With this in mind, let's look at some of the properties of hashes:

- All hashes are fixed-length, regardless of the length of the password. This way, the hacker is unable to work out how long your password is.
    - However, this can result in something called a **hash collision**: When two or more unique passwords generate the same hash.
    - If your algorithm is good enough, the chances of hash collision are negligibly low.
- Hashes are deterministic: The same password will generate the same hash.
- Hashes are irreversible: There is no inverse function to a hash.
- There is little to no information on the input present in the hash. This means someone can't get hints about your password by reading the hash.

The general idea is that when a user logs in and enters their password, your system will use an appropriate algorithm to hash the entered password and check that it matches the hash stored in the database. Given that the hash function is deterministic, it will generate the same hash if the password is correct. Although collisions are technically possible (see the **pigeonhole principle**), a good hashing algorithm ensures that the chances of this happening are incredibly small.

Below is an example of a user, Joe Bloggs, registering an account with a server. Note that the server stores the hash of the password under the password field. When Joe Bloggs logs into the server, it will hash the password that was given and then compare the two hashes. Because the same password will always produce the same hash, the two hashes will match if he has entered the correct password.



Password hashing

When a hacker gains access to the user passwords, they will be unable to read the passwords due to the fact that the hashes are irreversible. However, a technique called password cracking (or hash cracking) can technically reverse a hash. It is simply a brute-force approach to auto-generating passwords, hashing them, and checking them against all hashes.

Cracking is very computationally expensive and can take hours, days, or a few weeks to find passwords. This, of course, depends on how strong each password is.

Cracking

Two common cracking techniques are dictionary attacks and brute-force attacks.

Dictionary attacks

In a dictionary attack, a malicious actor uses a password-cracking tool that downloads a comprehensive list of commonly used words, phrases, and previously leaked passwords. The tool systematically tries each entry from the list to gain unauthorised access to a system or account. This method leverages the fact that many users choose simple, predictable passwords, often based on dictionary words or common phrases.

This attack can be further enhanced through a technique known as a **rainbow attack**. In this variation, the attacker applies common character substitutions to increase the likelihood of success. For example, they might replace the letter “O” with “0”, “e” with “3”, or “I” with “!” – a practice known as “leet speak”. These substitutions allow the attacker to crack passwords that are not direct dictionary words but are close variants, which many users adopt to meet password complexity requirements while still being easy to remember.

This type of attack will only work on passwords consisting of words that can be found in the dictionary or previously leaked lists of commonly chosen passwords. Even though this approach speeds up the hack a lot, it can still take hours or days.

Did you know?

One technique used to reduce the impact of poor password choice is **salting**. The idea is that, on top of asking the user to enter a password, you generate a random and unique string for each user (called the salt). When hashing the user’s password, you add the salt to the string before hashing.

If you store the salt separately from the user’s information, this adds another layer of difficulty for the wannabe password cracker, as they do not know the salt. They therefore can’t crack the password, even if the password is straightforward.

Brute-force attacks

This technique involves methodically generating every possible combination of letters, numbers, and characters, applying the hashing algorithm to them, and comparing each of those against the stored password hashes.

This is significantly slower than a dictionary attack and can take as long as a few days to a few weeks for even small and relatively insecure passwords. Longer passwords (greater than 12 characters, for example) can take weeks to months to crack.

The following table gives some idea of the effectiveness of a brute-force attack in terms of time taken when looking at password complexity.

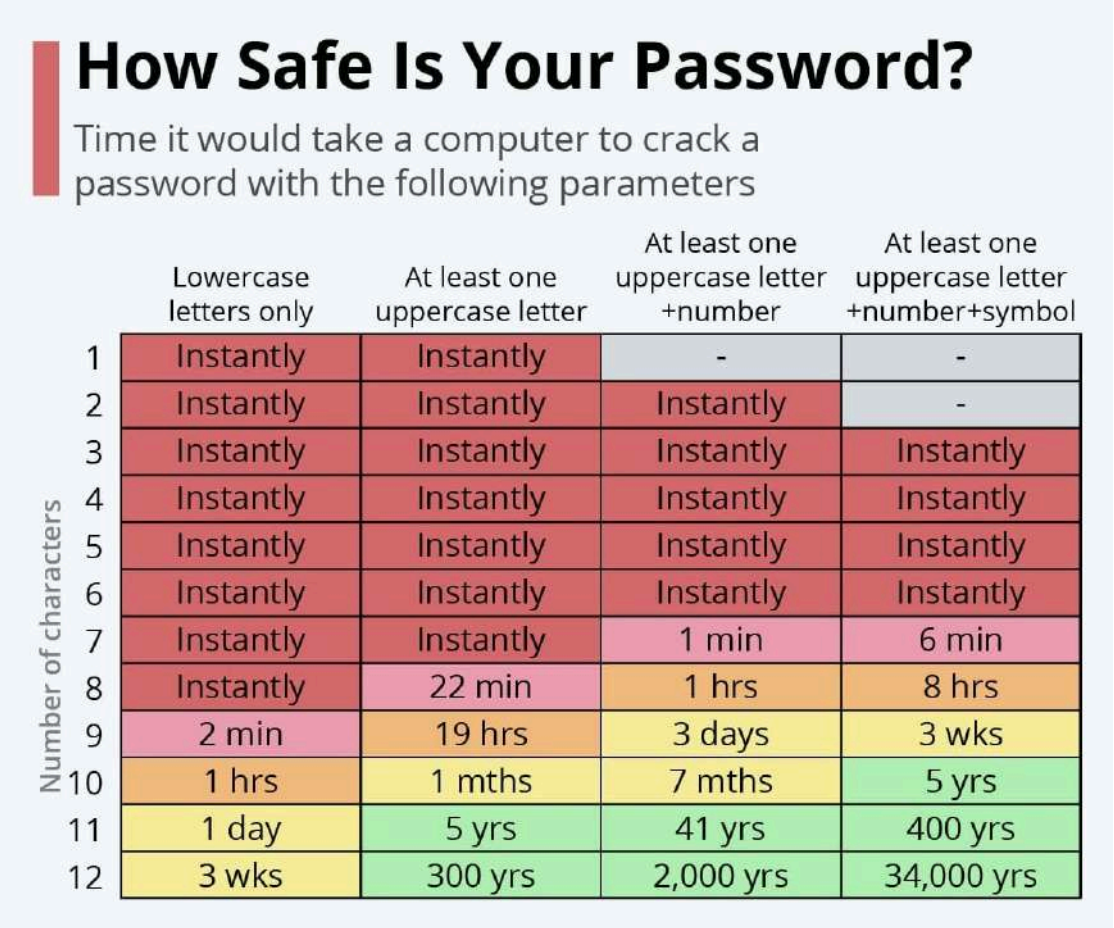


Table depicting password safety (Buchholz, 2021)

Hashing for data retrieval and digital signatures

If this is your first venture into the world of hashing and encryption, you might be wondering where to start. As we've seen above, several different hashing algorithms have become available over the years, with SHA-256 and MD5 being the most complex.

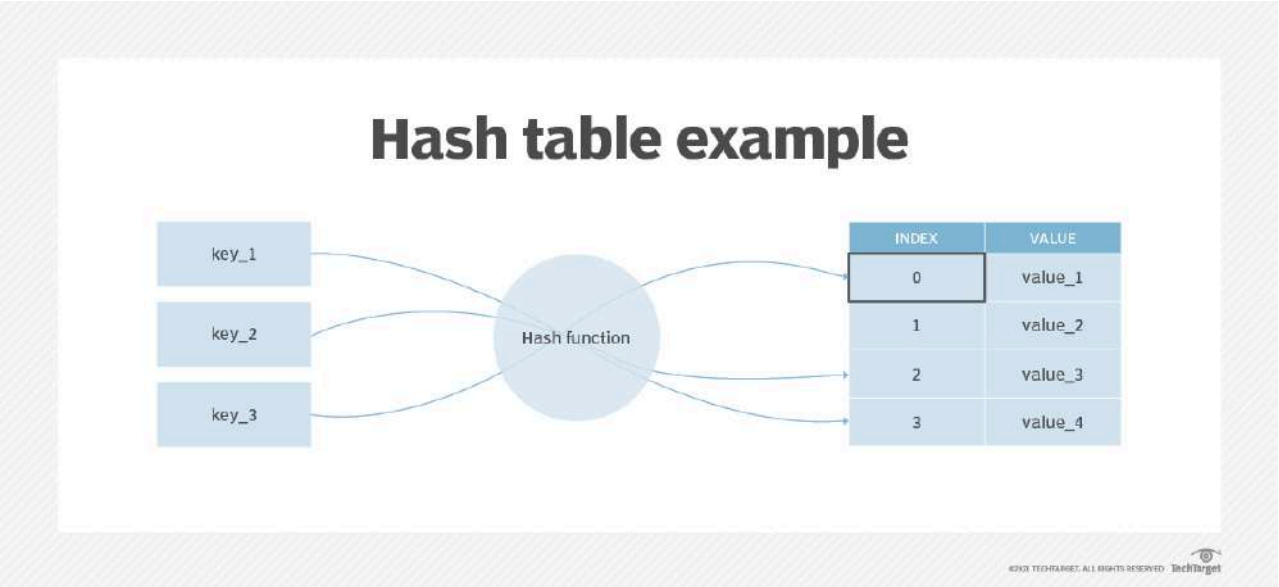
We can start by having a look at how hashing is used for data retrieval and digital signatures.

Data retrieval

Hashing is a process where an object's data, such as a string of text, is transformed into a representative integer value using a specific function or algorithm. This integer, known as a hash code, serves as a unique identifier for the data. Once the hash code is generated, it can be used to quickly locate the original data in a collection, making searches more efficient.

For example, developers often use hash tables to store data like customer records. In a hash table, data is organised into key–value pairs. The "key" is a unique identifier for the data, and the "value" is the actual data being stored. The hash code, derived from the key, is then mapped to a fixed-size location in the hash table. This mapping allows the data to be quickly retrieved later using the key.

The key plays a crucial role in this process: it identifies the data and is used as input to the hashing function. The hashing function then converts the key into a hash code, which determines where the data is stored in the hash table. This way, even large datasets can be searched and accessed efficiently, as the hash code directly points to where the data is located.

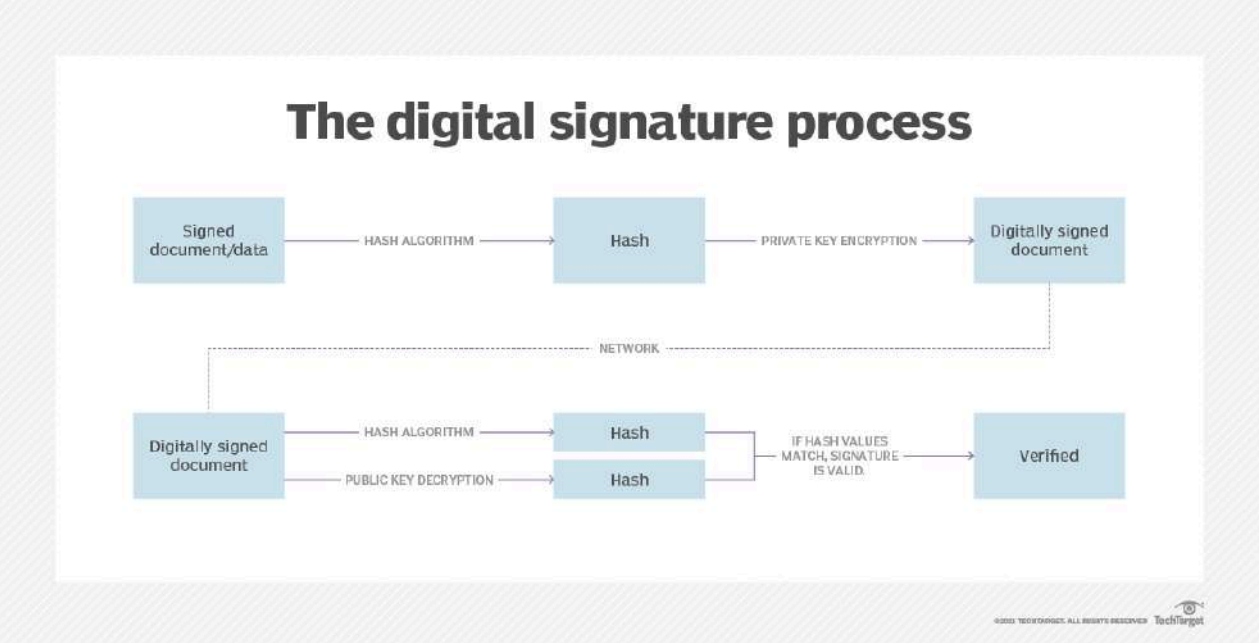


Hash table example (Yasar & Zola, 2024)

Digital signatures

Hashing not only makes data retrieval quick, but it also aids in the encryption and decryption of digital signatures, which verify the sender and recipient of the messages. In this case, the digital signature is first transformed by a hash function before being sent to the recipient in separate transmissions along with the hashed value, also called a message digest.

The message digest is obtained from the signature by the same hash function upon receipt, and it is compared to the message digest that was transmitted to make sure they match. When a value or key is retrieved in a one-way hashing operation, the hash function indexes the original value or key and makes data related to that value or key accessible.



The digital signature process (Hashemi-Pour et al., 2024)

Take note

Python allows us to install and use a wide range of libraries that aren't built in, using a tool called `pip` (Python's package manager).

It's important to note that it's considered a **best practice** in the tech industry to always set up a **virtual environment** when working with new external libraries. Think of the virtual environment as a container where you can install and test different libraries without having to worry about anything affecting your global workspace (just like a sandbox). Once you're done using it, you can simply remove the virtual environment (don't worry, you can always create a new one if you need it).

You will get the chance to set up your own virtual environment for Python in the task at the end of this document. Use the *Additional Reading* document provided in your task folder for instructions to guide you through the process.

Creating a hash function

Take a look at the code below to see how a hash function can be coded in Python.

```
import hashlib


def string_hasher(input_string):
    """
    This function takes an input string, encodes it into a UTF-8 byte format,and
    applies the SHA-256 hashing algorithm to it. The result is then converted
    into a readable hexadecimal string using the hexdigest() method, which can
    be used to securely store or compare sensitive information.
    """
    hashed_string = hashlib.sha256(input_string.encode()).hexdigest()
    return hashed_string


# Call the function with the strings that we would like to hash.
hashed1 = string_hasher("This is sensitive text.")
hashed2 = string_hasher("_password-123_")

# Here we can demonstrate the result of our hashing function.
print(f"\'_password-123_\' \t-->Hash Function-->\t {hashed1}")
```

In this code, we define a function called `string_hasher` that takes an input string and returns its SHA-256 hash. The `hashlib` library is used to access the SHA-256 hashing algorithm. Inside the function, the input string is first encoded into a UTF-8 byte format, as the SHA-256 algorithm works with byte data. Then, the `sha256()` function computes the hash, and `hexdigest()` converts it into a readable hexadecimal string. Finally, we call this function with different strings to demonstrate how their hashes are generated. The result is printed to show the hashed version of the input string.

Take note

The task below is **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

Auto-graded task

Follow these steps:

1. Follow the instructions in the *Additional Reading* document to set up your virtual environment, and then activate it according to the steps provided.
2. Create a file called **password_hash.py**.
3. Referring to the code example provided earlier as a starting point, use the **bcrypt** Python library (hint: `import bcrypt`) to assist you in defining a function that hashes a password string provided by the user.
4. Your function should encode the user's string first before any hashing takes place.
5. Make use of the `bcrypt` library to hash the password while generating a random salt.
6. Add a **requirements.txt** file for your mentor, containing all of the dependencies for your program.
7. Deactivate your virtual environment once you have completed the task.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Optional task

Follow these steps:

1. Follow the instructions in the *Additional Reading* document and create a folder for your virtual environment. Then activate the virtual environment.
2. Do some research on **symmetric encryption**.
3. Create a file called **encrypt.py**.
4. By taking advantage of the `cryptography` Python library, define your function with two parameters: `text` (the string of text to be encrypted) and `key` (the key used to encrypt the text).
    - Hint: `from cryptography.fernet import Fernet`
5. The function above should use the key provided to encrypt the text provided and then return the encrypted text.
6. Using the same library and logic above, create another function that will decrypt any encrypted text using the same key that it was encrypted with.
7. Deactivate your virtual environment once you have completed the task and remember to delete the virtual environment folder (you can always create a new one).

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

Buchholz, K. (2021, December 7). *This chart shows how long it would take a computer to hack your exact password.* World Economic Forum.
https://www.weforum.org/agenda/2021/12/passwords-safety-cybercrime/

Hashemi-Pour, C., Gillis, A. S., & Lutkevich, B. (2024, June). *Digital signature*. TechTarget.
https://www.techtarget.com/searchsecurity/definition/digital-signature

Okta. (2024, August 30). *Hashing vs. encryption: Definitions & differences*.
https://www.okta.com/identity-101/hashing-vs-encryption/

Yasar, K., & Zola, A. (2024, May). *Hashing*. TechTarget.
https://www.techtarget.com/searchdatamanagement/definition/hashing