Introduction to Cryptography

Introduction

The Caesar cipher that was made 2000 years ago is one of the most simplest ciphers used. Caesar cipher shifts the letter by a fixed number

This can be use a key between 1-25. Such as if the key was 5. A would be F. A key of 0 or 26 would lead to no change as 0 will not lead to anything and 26 would be a full rotation, If you are unable to translate it you can always use a brute force method to crack the cipher

Caesar cipher is considered a **substitution cipher** because each letter in the alphabet is substituted with another.

Apart from the Caesar cipher there is also another known as the **Transposition Cipher**, which encrypts the message by changing the order of the letters. After we write the letters of our message by filling one column after the other, we rearrange the columns based on the key and then read the rows.

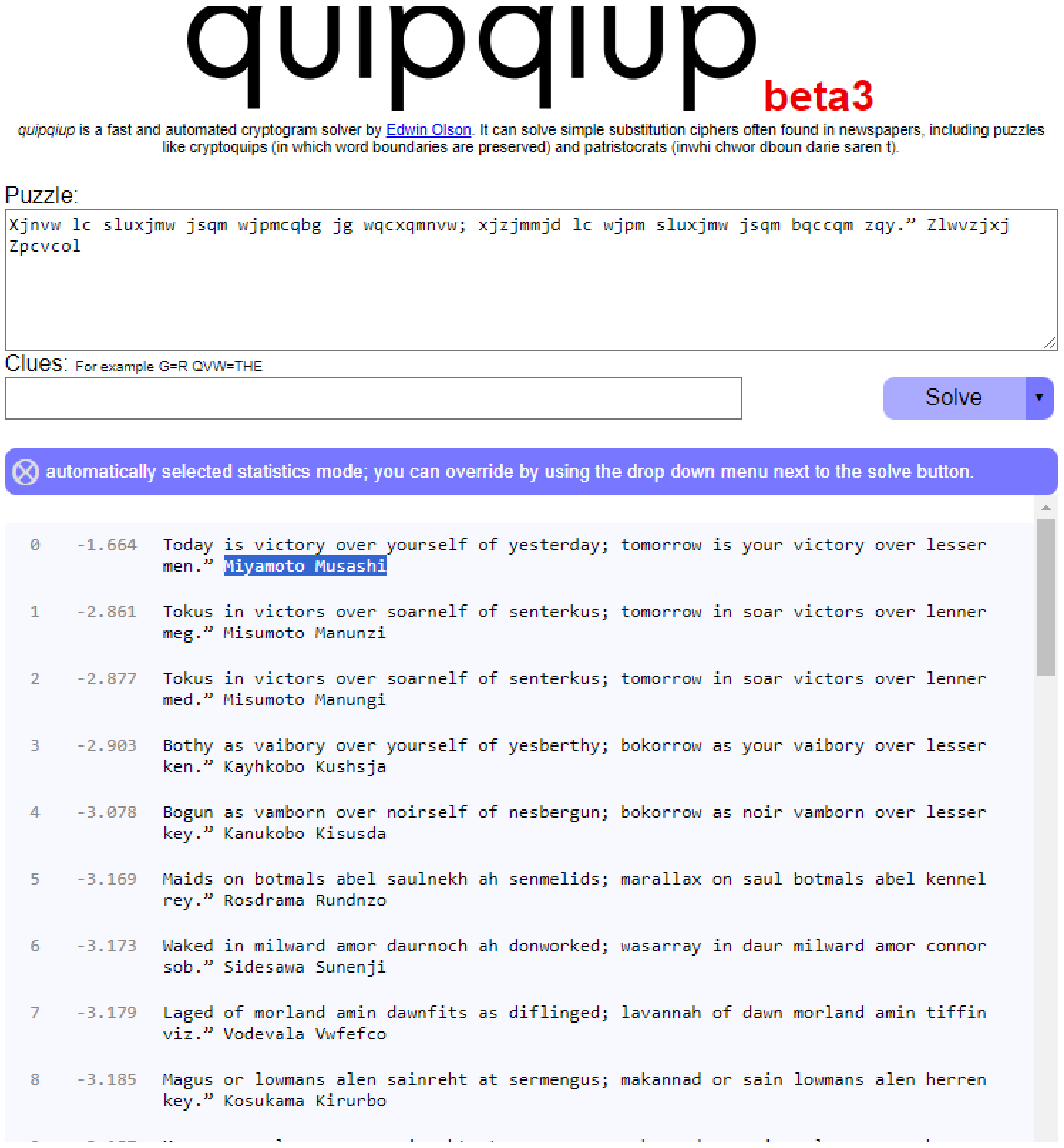
For an encryption algorithm to be considered **secure**, it should be infeasible to recover the original message, i.e., plaintext.

**You have received the following encrypted message:

***"Xjnvw lc sluxjmw jsqm wjpmcqbg jg wqcxqmnvw; xjzjmmjd lc wjpm sluxjmw jsqm bqccqm zqy." Zlwvzjxj Zpcvcol*

You can guess that it is a quote. Who said it?

Go to Quipqiup and put in the text. It will be the first one:



quipqiup is a fast and automated cryptogram solver by Edwin Olson. It can solve simple substitution ciphers often found in newspapers, including puzzles like cryptoquips (in which word boundaries are preserved) and patristocrats (inwhi chwor dboun darie saren t).

Puzzle:

```
Xjnvw lc sluxjmw jsqm wjpmcqbg jg wqcxqmnvw; xjzjmmjd lc wjpm sluxjmw jsqm bqccqm zqy.” Zlwvzjxj Zpcvcol
```

Clues: For example G=R QVW=THE

automatically selected statistics mode; you can override by using the drop down menu next to the solve button.

```
0   -1.664   Today is victory over yourself of yesterday; tomorrow is your victory over lesser
             men.” Miyamoto Musashi

1   -2.861   Tokus in victors over soarnelf of senterkus; tomorrow in soar victors over lenner
             meg.” Misumoto Manunzi

2   -2.877   Tokus in victors over soarnelf of senterkus; tomorrow in soar victors over lenner
             med.” Misumoto Manungi

3   -2.903   Bothy as vaibory over yourself of yesberthy; bokorrow as your vaibory over lesser
             ken.” Kayhkobo Kushsja

4   -3.078   Bogun as vamborn over noirself of nesbergun; bokorrow as noir vamborn over lesser
             key.” Kanukobo Kisusda

5   -3.169   Maids on botmals abel saulnekh ah senmelids; marallax on saul botmals abel kennel
             rey.” Rosdrama Rundnzo

6   -3.173   Waked in milward amor daurnoch ah donworked; wasarray in daur milward amor connor
             sob.” Sidesawa Sunenji

7   -3.179   Laged of morland amin dawnfits as diflinged; lavannah of dawn morland amin tiffin
             viz.” Vodevala Vwfefco

8   -3.185   Magus or lowmans alen sainreht at sermengus; makannad or sain lowmans alen herren
             key.” Kosukama Kirurbo
```

Answer: Miyamoto Musashi

Symmetric Encryption

A symmetric encryption algorithm uses the same key for encryption and decryption. Consequently, the communicating parties need to agree on a secret key before being able to exchange any messages.

Here is what is commonly said when talking about encryption:

- **Cryptographic Algorithm** or **Cipher**: This algorithm defines the encryption and

decryption processes.
- **Key**: The cryptographic algorithm needs a key to convert the plaintext into ciphertext and vice versa.
- **plaintext** is the original message that we want to encrypt
- **ciphertext** is the message in its encrypted form

National Institute of Standard and Technology (NIST) published the Data Encryption Standard (DES) in 1977. DES is a symmetric encryption algorithm that uses a key size of 56 bits. In 1997, a challenge to break a message encrypted using DES was solved. Consequently, it was demonstrated that it had become feasible to use a brute-force search to find the key and break a message encrypted using DES. In 1998, a DES key was broken in 56 hours. These cases indicated that DES could no longer be considered secure.

NIST published the Advanced Encryption Standard (AES) in 2001. Like DES, it is a symmetric encryption algorithm; however, it uses a key size of 128, 192, or 256 bits, and it is still considered secure and in use today. AES repeats the following four transformations multiple times:

1. `SubBytes(state)`: This transformation looks up each byte in a given substitution table (S-box) and substitutes it with the respective value. The `state` is 16 bytes, i.e., 128 bits, saved in a 4 by 4 array.
2. `ShiftRows(state)`: The second row is shifted by one place, the third row is shifted by two places, and the fourth row is shifted by three places. This is shown in the figure below.
3. `MixColumns(state)`: Each column is multiplied by a fixed matrix (4 by 4 array).
4. `AddRoundKey(state)`: A round key is added to the state using the XOR operation.

A block cipher algorithm converts the input (plaintext) into blocks and encrypts each block. A block is usually 128 bits. Here are some examples of Block ciphers:

- **AES, 192, 256** - Keys size of 128, 192 and 256 bits
- **IDEA** - International Data Encryption Algorithm
- **3DES** - Triple DES (Data Encryption Standard) based on DES. 3DES will be deprecated in 2023 and disallowed in 2024.
- **CAST5** - Also known as CAST-128. Some sources state that CASE stands for the names of its authors: Carlisle Adams and Stafford Tavares.
- **Blowfish** - Created by Bruce Schneier

- **Twofish -** Created by the same person as Blowfish and derived from Blowfish
- **CAMELLIA 128, 192, 256 -** Designed by Mitsubishi Electric and NTT in Japan. Its name is derived from the flower camellia japonica.

Symmetric Encryption can help following the security principle of the CIA Triad:

- **Confidentiality**: If Eve intercepted the encrypted message, she wouldn't be able to recover the plaintext. Consequently, all messages exchanged between Alice and Bob are confidential as long as they are sent encrypted.
- **Integrity**: When Bob receives an encrypted message and decrypts it successfully using the key he agreed upon with Alice, Bob can be sure that no one could tamper with the message across the channel. When using secure modern encryption algorithms, any minor modification to the ciphertext would prevent successful decryption or would lead to gibberish as plaintext.
- **Authenticity**: Being able to decrypt the ciphertext using the secret key also proves the authenticity of the message because only Alice and Bob know the secret key.

We can use two tools to help us Encrypt and Decrypt:

- GNU Privacy Guard
- OpenSSL Project

GNU Privacy Guard

The GNU Privacy Guard, also known as GnuPG or GPG, implements the OpenPGP standard.

We can encrypt a file using GnuPG (GPG) using the following command:

`gpg --symmetric --cipher-algo CIPHER message.txt`, where CIPHER is the name of the encryption algorithm. You can check supported ciphers using the command `gpg --version`. The encrypted file will be saved as `message.txt.gpg`.

The default output is in the binary OpenPGP format; however, if you prefer to create an ASCII armoured output, which can be opened in any text editor, you should add the option `--armor`. For example, `gpg --armor --symmetric --cipher-algo CIPHER message.txt`.

You can decrypt using the following command:

`gpg --output original_message.txt --decrypt message.gpg`

OpenSSL Project

The OpenSSL Project maintains the OpenSSL software.

We can encrypt a file using OpenSSL using the following command:

```
openssl aes-256-cbc -e -in message.txt -out encrypted_message
```

We can decrypt the resulting file using the following command:

```
openssl aes-256-cbc -d -in encrypted_message -out original_message.txt
```

To make the encryption more secure and resilient against brute-force attacks, we can add `-pbkdf2` to use the Password-Based Key Derivation Function 2 (PBKDF2); moreover, we can specify the number of iterations on the password to derive the encryption key using `-iter NUMBER`. To iterate 10,000 times, the previous command would become:

```
openssl aes-256-cbc -pbkdf2 -iter 10000 -e -in message.txt -out encrypted_message
```

Consequently, the decryption command becomes:

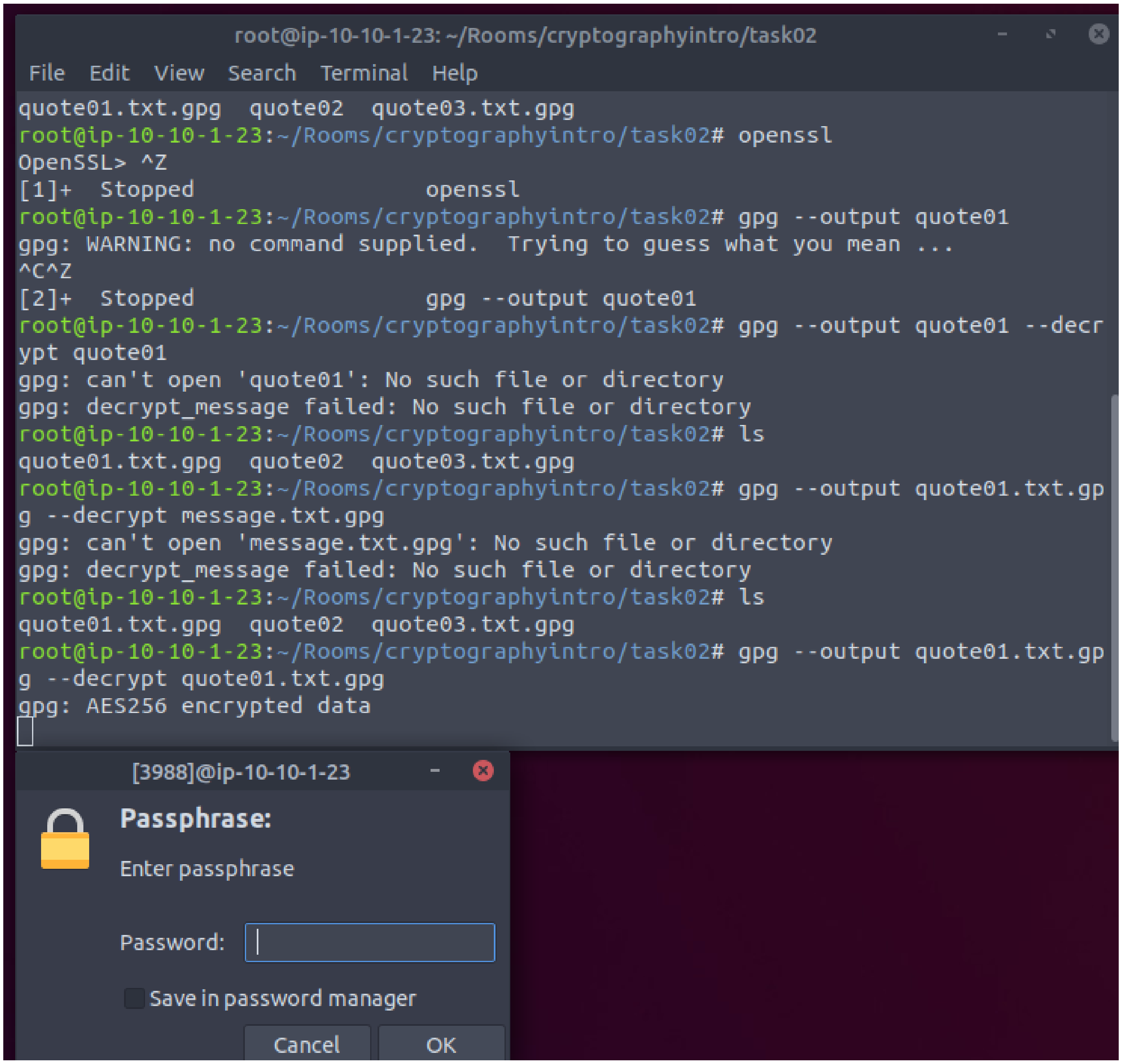
```
openssl aes-256-cbc -pbkdf2 -iter 10000 -d -in encrypted_message -out
original_message.txt
```

Decrypt the file `quote01` encrypted (using AES256) with the key `s!kR3T55` using `gpg`. What is the third word in the file?

We use the following command, which then we put the passphrase in as a pop up appears:

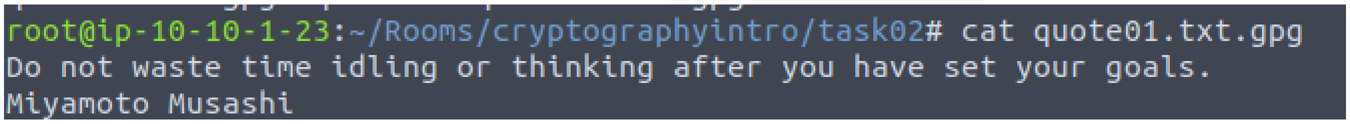
```
gpg --output quote01.txt.gpg --decrypt quote01.txt.gpg
```

When the passphrase box opens up we put in the key `s!kR3T55`


root@ip-10-10-1-23: ~/Rooms/cryptographyintro/task02
File Edit View Search Terminal Help

```
quote01.txt.gpg  quote02  quote03.txt.gpg
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task02# openssl
OpenSSL> ^Z
[1]+  Stopped                 openssl
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task02# gpg --output quote01
gpg: WARNING: no command supplied.  Trying to guess what you mean ...
^C^Z
[2]+  Stopped                 gpg --output quote01
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task02# gpg --output quote01 --decr
ypt quote01
gpg: can't open 'quote01': No such file or directory
gpg: decrypt_message failed: No such file or directory
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task02# ls
quote01.txt.gpg  quote02  quote03.txt.gpg
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task02# gpg --output quote01.txt.gp
g --decrypt message.txt.gpg
gpg: can't open 'message.txt.gpg': No such file or directory
gpg: decrypt_message failed: No such file or directory
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task02# ls
quote01.txt.gpg  quote02  quote03.txt.gpg
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task02# gpg --output quote01.txt.gp
g --decrypt quote01.txt.gpg
gpg: AES256 encrypted data
```

We have decrypted the file now we can run `cat quote01.txt.gpg`



```
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task02# cat quote01.txt.gpg
Do not waste time idling or thinking after you have set your goals.
Miyamoto Musashi
```

Answer: Waste

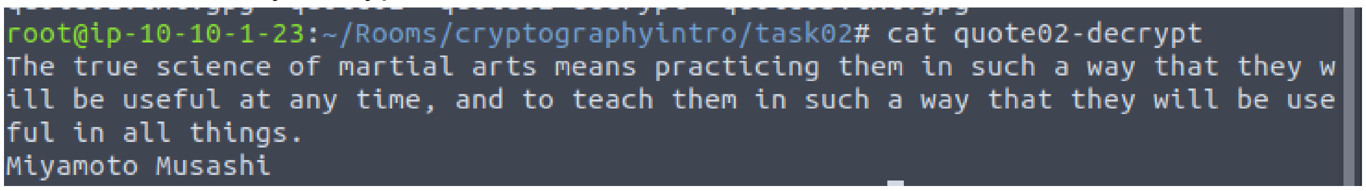
Decrypt the file `quote02` encrypted (using AES256-CBC) with the key `s!kR3T55` using `openssl`. What is the third word in the file?

We use the following command:

```
openssl aes-256-cbc -d -in quote02 -out quote02-decrypt
```

Put in the key we was given: s!kR3T55

Then we have fully decrypted the file:

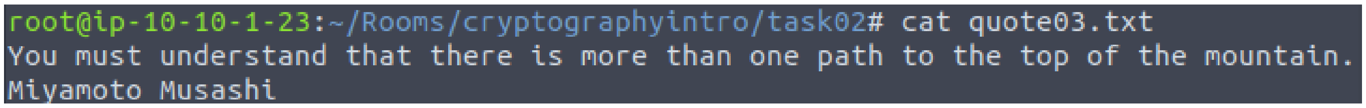


```
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task02# cat quote02-decrypt
The true science of martial arts means practicing them in such a way that they w
ill be useful at any time, and to teach them in such a way that they will be use
ful in all things.
Miyamoto Musashi
```

Answer: Science

Decrypt the file `quote03` encrypted (using CAMELLIA256) with the key `s!kR3T55` using `gpg`. What is the third word in the file?

Following the same steps as the first question and putting in the key with gpg we get our decrypted quote:



```
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task02# cat quote03.txt
You must understand that there is more than one path to the top of the mountain.
Miyamoto Musashi
```

Answer: Understand

Asymmetric Encryption

Symmetric encryption requires the users to find a secure channel to exchange keys. By secure channel, we are mainly concerned with confidentiality and integrity. In other words, we need a channel where no third party can eavesdrop and read the traffic; moreover, no one can change the sent messages and data.

Asymmetric encryption makes it possible to exchange encrypted messages without a secure channel; we just need a reliable channel. By reliable channel, we mean that we are mainly concerned with the channel's integrity and not confidentiality.

When using an asymmetric encryption algorithm, we would generate a key pair: a public key and a private key. The public key is shared with the world, or more specifically, with the people who want to communicate with us securely. The private key must be saved securely, and we must never let anyone access it. Moreover, it is not feasible to derive the private key despite the knowledge of the public key.

RSA:

RSA got its name from its inventors, Rivest, Shamir, and Adleman. It works as follows:

1. Choose two random prime numbers, p and q. Calculate $N = p \times q$.

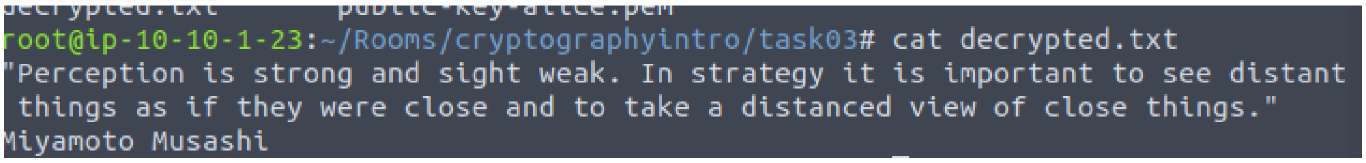
2. Choose two integers *e* and *d* such that *e* × *d* = 1 mod *ϕ*(*N*), where *ϕ*(*N*) = *N* − *p* − *q* + 1. This step will let us generate the public key (*N*,*e*) and the private key (*N*,*d*).
3. The sender can encrypt a value *x* by calculating *y* = *x*__*e* mod *N*. (Modulus)
4. The recipient can decrypt *y* by calculating *x* = *y*__*d* mod *N*. Note that *y*__*d* = *x**ed*** = *x**k**ϕ*(*N*) + 1 = (*x*__*ϕ*(*N*))*k* × *x* = *x*. This step explains why we put a restriction on the choice of *e* and *d*.

RSA security relies on factorization being a hard problem. It is easy to multiply *p* by *q*; however, it is time-consuming to find *p* and *q* given *N*. Moreover, for this to be secure, *p* and *q* should be pretty large numbers, for example, each being 1024 bits (that's a number with more than 300 digits). It is important to note that RSA relies on secure random number generation, as with other asymmetric encryption algorithms. If an adversary can guess *p* and *q*, the whole system would be considered insecure.

Bob has received the file `ciphertext_message` sent to him from Alice. You can find the key you need in the same folder. What is the first word of the original plaintext?

We use the following command to:

```
openssl pkeyutl -decrypt -in ciphertext_message -inkey private-key-bob.pem -out decrypted.txt
```



```
root@ip-10-10-1-23:~/Rooms/cryptographyintro/task03# cat decrypted.txt
"Perception is strong and sight weak. In strategy it is important to see distant
 things as if they were close and to take a distanced view of close things."
Miyamoto Musashi
```

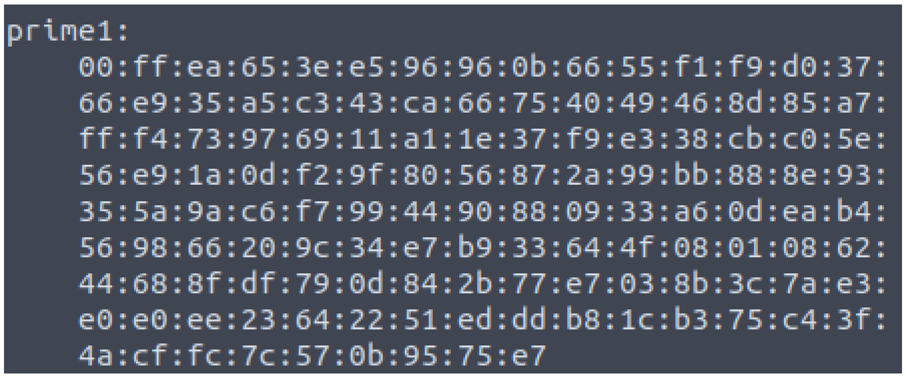
Answer: Perception

Take a look at Bob's private RSA key. What is the last byte of *p*?

We use the following command with Openssl:

```
openssl rsa -in private-key-bob.pem -text -noout
```

As P = Prime1 we scroll down to Prime1 on the output and find the last byte of P

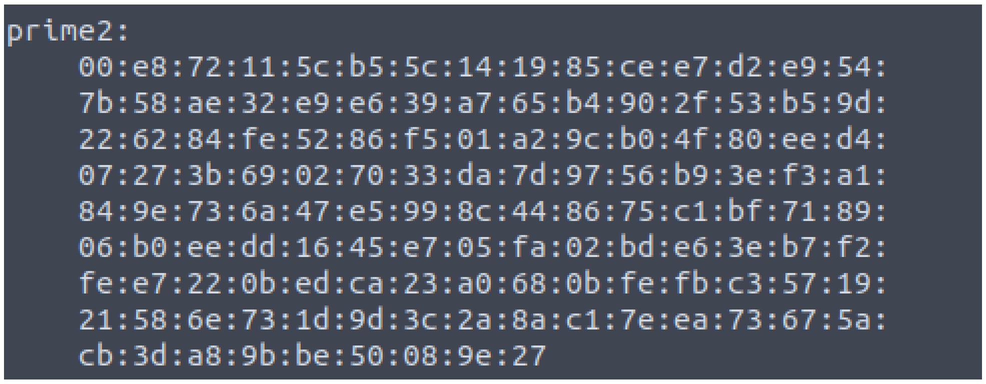


```
prime1:
    00:ff:ea:65:3e:e5:96:96:0b:66:55:f1:f9:d0:37:
    66:e9:35:a5:c3:43:ca:66:75:40:49:46:8d:85:a7:
    ff:f4:73:97:69:11:a1:1e:37:f9:e3:38:cb:c0:5e:
    56:e9:1a:0d:f2:9f:80:56:87:2a:99:bb:88:8e:93:
    35:5a:9a:c6:f7:99:44:90:88:09:33:a6:0d:ea:b4:
    56:98:66:20:9c:34:e7:b9:33:64:4f:08:01:08:62:
    44:68:8f:df:79:0d:84:2b:77:e7:03:8b:3c:7a:e3:
    e0:e0:ee:23:64:22:51:ed:dd:b8:1c:b3:75:c4:3f:
    4a:cf:fc:7c:57:0b:95:75:e7
```

Answer: e7

Take a look at Bob's private RSA key. What is the last byte of *q*?

As q = Prime2 we scroll down to prime2 and look at the last byte



```
prime2:
    00:e8:72:11:5c:b5:5c:14:19:85:ce:e7:d2:e9:54:
    7b:58:ae:32:e9:e6:39:a7:65:b4:90:2f:53:b5:9d:
    22:62:84:fe:52:86:f5:01:a2:9c:b0:4f:80:ee:d4:
    07:27:3b:69:02:70:33:da:7d:97:56:b9:3e:f3:a1:
    84:9e:73:6a:47:e5:99:8c:44:86:75:c1:bf:71:89:
    06:b0:ee:dd:16:45:e7:05:fa:02:bd:e6:3e:b7:f2:
    fe:e7:22:0b:ed:ca:23:a0:68:0b:fe:fb:c3:57:19:
    21:58:6e:73:1d:9d:3c:2a:8a:c1:7e:ea:73:67:5a:
    cb:3d:a8:9b:be:50:08:9e:27
```

Answer: 27

Diffie-Hellman Key Exchange

Diffie-Hellman is an asymmetric encryption algorithm. It allows the exchange of a secret over a public channel

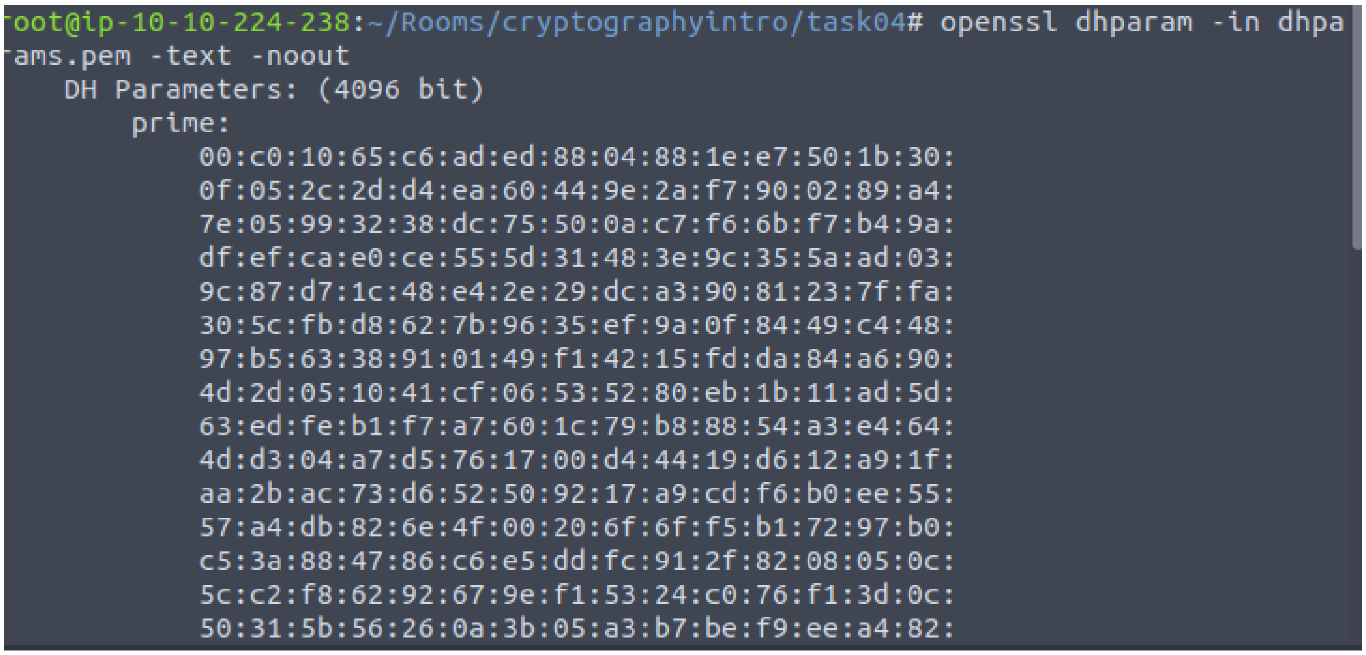
Diffie-Hellman key exchange algorithm allows two parties to agree on a secret over an insecure channel. However, the discussed key exchange is prone to a Man-in-the-Middle (MitM) attack

We can use `openssl` to generate them; we need to specify the option `dhparam` to indicate that we want to generate Diffie-Hellman parameters along with the specified size in bits, such as `2048` or `4096`.

A set of Diffie-Hellman parameters can be found in the file `dhparam.pem`. What is the size of the prime number in bits?

We run the following command:
`openssl dhparam -in dhparams.pem -text -noout`

At the top of the output we can see how many bits there are:

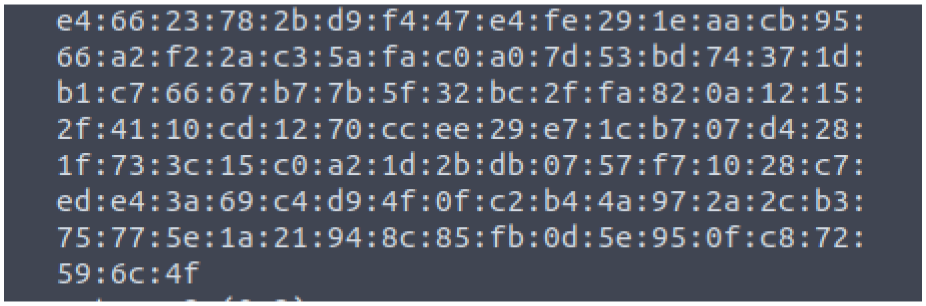


```
root@ip-10-10-224-238:~/Rooms/cryptographyintro/task04# openssl dhparam -in dhpa
rams.pem -text -noout
    DH Parameters: (4096 bit)
        prime:
            00:c0:10:65:c6:ad:ed:88:04:88:1e:e7:50:1b:30:
            0f:05:2c:2d:d4:ea:60:44:9e:2a:f7:90:02:89:a4:
            7e:05:99:32:38:dc:75:50:0a:c7:f6:6b:f7:b4:9a:
            df:ef:ca:e0:ce:55:5d:31:48:3e:9c:35:5a:ad:03:
            9c:87:d7:1c:48:e4:2e:29:dc:a3:90:81:23:7f:fa:
            30:5c:fb:d8:62:7b:96:35:ef:9a:0f:84:49:c4:48:
            97:b5:63:38:91:01:49:f1:42:15:fd:da:84:a6:90:
            4d:2d:05:10:41:cf:06:53:52:80:eb:1b:11:ad:5d:
            63:ed:fe:b1:f7:a7:60:1c:79:b8:88:54:a3:e4:64:
            4d:d3:04:a7:d5:76:17:00:d4:44:19:d6:12:a9:1f:
            aa:2b:ac:73:d6:52:50:92:17:a9:cd:f6:b0:ee:55:
            57:a4:db:82:6e:4f:00:20:6f:6f:f5:b1:72:97:b0:
            c5:3a:88:47:86:c6:e5:dd:fc:91:2f:82:08:05:0c:
            5c:c2:f8:62:92:67:9e:f1:53:24:c0:76:f1:3d:0c:
            50:31:5b:56:26:0a:3b:05:a3:b7:be:f9:ee:a4:82:
```

Answer: 4096

What is the prime number’s last byte (least significant byte)?

Scrolling to the bottom of the output we can see the last hexadecimal:



```
    e4:66:23:78:2b:d9:f4:47:e4:fe:29:1e:aa:cb:95:
    66:a2:f2:2a:c3:5a:fa:c0:a0:7d:53:bd:74:37:1d:
    b1:c7:66:67:b7:7b:5f:32:bc:2f:fa:82:0a:12:15:
    2f:41:10:cd:12:70:cc:ee:29:e7:1c:b7:07:d4:28:
    1f:73:3c:15:c0:a2:1d:2b:db:07:57:f7:10:28:c7:
    ed:e4:3a:69:c4:d9:4f:0f:c2:b4:4a:97:2a:2c:b3:
    75:77:5e:1a:21:94:8c:85:fb:0d:5e:95:0f:c8:72:
    59:6c:4f
```

Answer: 4f

Hashing

A cryptographic hash function is an algorithm that takes data of arbitrary size as its input and returns a fixed size value, called *message digest* or *checksum*, as its output.

Why we need hashing:

- Storing passwords: Instead of storing passwords in plaintext, a hash of the password is stored instead. Consequently, if a data breach occurs, the attacker will get a list of password hashes instead of the original passwords. (In practice, passwords are also “salted”, as discussed in a later task.)

- Detecting modifications: Any minor modification to the original file would lead to a drastic change in hash value, i.e. checksum.

Some older hash functions, such as MD5 (Message Digest 5) and SHA-1, are cryptographically broken. By broken, we mean that it is possible to generate a different file with the same checksum as a given file. This means that we can create a hash collision. In other words, an attacker can create a new message with a given checksum, and detecting file or message tampering won't be possible.

HMAC
Hash-based message authentication code (HMAC) is a message authentication code (MAC) that uses a cryptographic key in addition to a hash function.

According to RFC2104, HMAC needs:

- secret key
- inner pad (ipad) a constant string. (RFC2104 uses the byte `0x36` repeated B times. The value of B depends on the chosen hash function.)
- outer pad (opad) a constant string. (RFC2104 uses the byte `0x5C` repeated B times.)

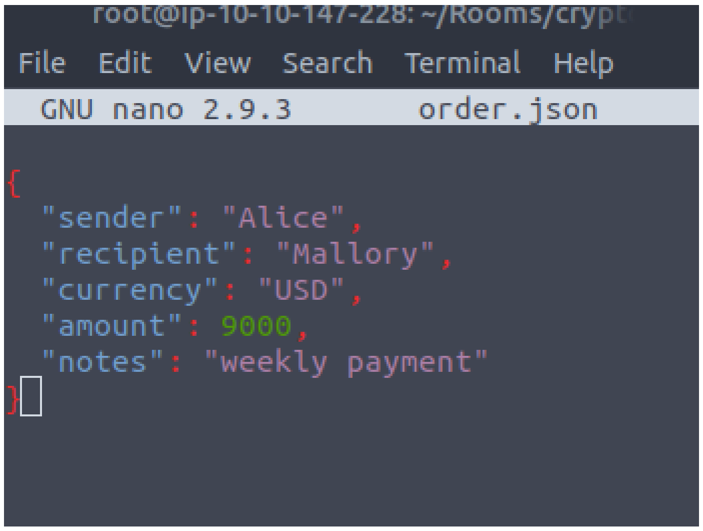
What is the SHA256 checksum of the file `order.json`?

Run the following command; `sha256sum order.json`

Answer: 2c34b68669427d15f76a1c06ab941e3e6038dacdfb9209455c87519a3ef2c660

Open the file `order.json` and change the amount from `1000` to `9000` What is the new SHA256 checksum?

Go into the file and change the amount to 9000:


root@ip-10-10-147-228: ~/Rooms/crypt
File Edit View Search Terminal Help

```
  GNU nano 2.9.3          order.json

{
  "sender": "Alice",
  "recipient": "Mallory",
  "currency": "USD",
  "amount": 9000,
  "notes": "weekly payment"
}
```

Now run the same command again as the previous question

Answer: 11faeec5edc2a2bad82ab116bbe4df0f4bc6edd96adac7150bb4e6364a238466

Using SHA256 and the key `3RfDFz82`, what is the HMAC of `order.txt`?

We use OpenSSL to get the answer; `openssl dgst -sha256 -hmac 3RfDFz82 order.txt`

Answer: c7e4de386a09ef970300243a70a444ee2a4ca62413aeaeb7097d43d2c5fac89f

PKI and SSL/TLS

PKI is a public key infrastructure is a set of roles, policies, hardware, software and procedures needed to create, manage, distribute, use, store and revoke digital certificates and manage public-key encryption.

For a certificate to get signed by a certificate authority, we need to:

1. Generate Certificate Signing Request (CSR): You create a certificate and send your public key to be signed by a third party.
2. Send your CSR to a Certificate Authority (CA): The purpose is for the CA to sign your certificate. The alternative and usually insecure solution would be to self-sign your certificate.

For this to work, the recipient should recognize and trust the CA that signed the certificate. And as we would expect, our browser trusts DigiCert Inc as a signing authority; otherwise, it would have issued a security warning instead of proceeding to the requested website.

SSL supports older algorithms with known security vulnerabilities. TLS uses advanced encryption algorithms. An SSL handshake is complex and slow. A TLS handshake has fewer steps and a faster connection.

Once the client, i.e., the browser, receives a signed certificate it trusts, the SSL/TLS handshake takes place. The purpose would be to agree on the ciphers and the secret key.

For example, the following command will generate a self-signed certificate.

`openssl req -x509 -newkey -nodes rsa:4096 -keyout key.pem -out cert.pem -sha256 -days 365`

The `-x509` indicates that we want to generate a self-signed certificate instead of a certificate request. The `-sha256` specifies the use of the SHA-256 digest. It will be valid for one year as we added `-days 365`.

To answer the below two questions we use `openssl x509 -in cert.pem -text`

What is the size of the public key in bits?
4096

Till which year is this certificate valid?
2039

Authenticating With Passwords

With PKI and SSL/TLS, we can communicate with any server and provide our login credentials while ensuring that no one can read our passwords as they move across the network. This is an example of protecting data in transit.

The improved approach would be to save the username and a hashed version of the password in a database. This way, a data breach will expose the hashed versions of the passwords. Since a hash function is irreversible, the attacker needs to keep trying different passwords to find the one that would result in the same hash.

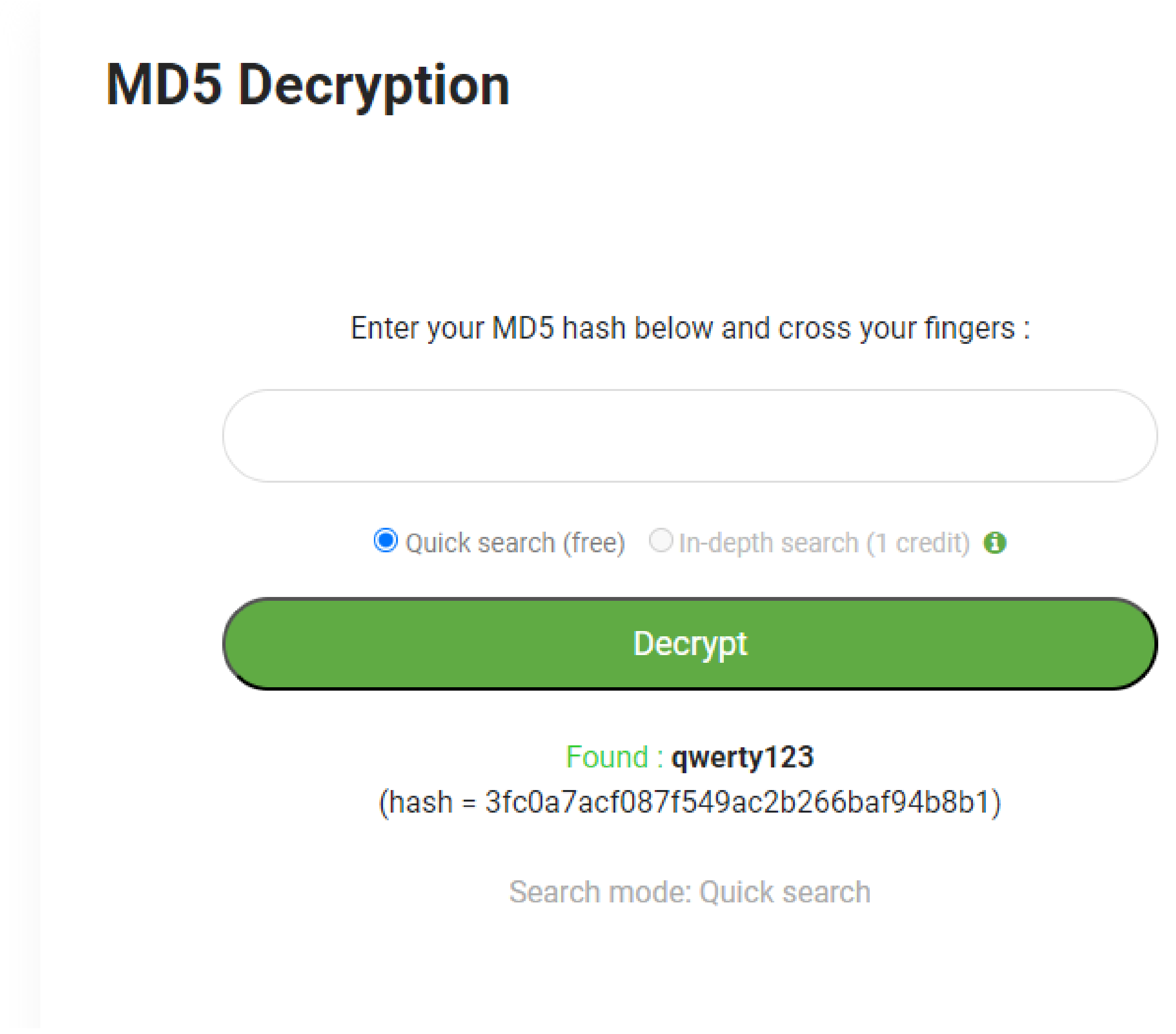
The previous approach looks secure; however, the availability of rainbow tables has made this approach insecure. A **rainbow table** contains a list of passwords along with their hash value. Hence, the attacker only needs to look up the hash to recover the password.

Another approach would be to use `hash(hash(password) + salt)`. Note that we used a relatively small salt along with the MD5 hash function. We should switch to a (more) secure hash function and a large salt for better security if this were an actual setup.

Another improvement we can make before saving the password is to use a key derivation function such as PBKDF2 (Password-Based Key Derivation Function 2). PBKDF2 takes the password and the salt and submits it through a certain number of iterations, usually hundreds of thousands.

You were auditing a system when you discovered that the MD5 hash of the admin password is `3fc0a7acf087f549ac2b266baf94b8b1`. What is the original password?

We search for an MD5 cracker online, put the hash in to crack it and we have the password


MD5 Decryption

Enter your MD5 hash below and cross your fingers :

Quick search (free) [✓] In-depth search (1 credit)

Found : **qwerty123**
(hash = 3fc0a7acf087f549ac2b266baf94b8b1)

Search mode: Quick search

Answer: qwerty123

Cryptography and Data - Example

In this task, we would like to explore what happens when we log into a website over HTTPS.

1. Client requests server's SSL/TLS certificate
2. Server sends SSL/TLS certificate to the client
3. Client confirms that the certificate is valid

Cryptography's role starts with checking the certificate. For a certificate to be considered valid, it means it is signed. Signing means that a hash of the certificate is encrypted with the private key of a trusted third party; the encrypted hash is appended to the certificate.

If the third party is trusted, the client will use the third party's public key to decrypt the encrypted hash and compare it with the certificate's hash. However, if the third party is not recognized, the connection will not proceed automatically.

Once the client confirms that the certificate is valid, an SSL/TLS handshake is started. This handshake allows the client and the server to agree on the secret key and the symmetric encryption algorithm, among other things. From this point onward, all the related session communication will be encrypted using symmetric encryption.

The final step would be to provide login credentials. The client uses the encrypted SSL/TLS session to send them to the server. The server receives the username and password and needs to confirm that they match.

Following security guidelines, we expect the server to save a hashed version of the password after appending a random salt to it. This way, if the database were breached, the passwords would be challenging to recover.