CRYPTOGRAPHY

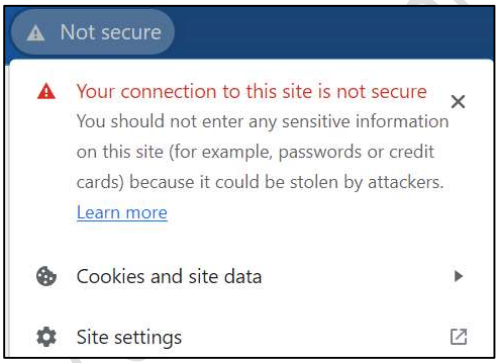
Once upon a time in a quiet village, there was a small shop eager to expand its business. To help achieve this goal, the shop owner decided to create a simple form on their website. This form allowed customers to provide their name, phone number, email address, and suggestions to boost sales. Excited about this new venture, the shop owner made the form accessible on the internet.

However, there was a significant problem: the shop owner didn't understand the difference between HTTP and HTTPS, which are crucial for securing websites.

HTTP (Hypertext Transfer Protocol) and HTTPS (Hypertext Transfer Protocol Secure) are like the locks on your front door. HTTP has no lock at all, while HTTPS is a strong and secure lock. When you see "HTTPS" in your web browser's address bar, it means the website is safe for sharing sensitive information like your name, phone number, and email.

But the shop owner, lacking this knowledge, unknowingly created their form using HTTP, which is like having no lock on their virtual door.

The shop owner enthusiastically shared the URL of their webpage in a public place, inviting everyone to fill out the form. Little did they know that a hacker was lurking nearby. This hacker noticed the absence of a secure connection (HTTPS) on the webpage and saw an opportunity to steal the information people submitted through the form.



Whenever someone filled out the form and clicked "Submit," their data would be sent to the website without any protection. The hacker, with just a bit of knowledge and the right tools, could easily intercept and access all the data being sent to the website. Names, phone numbers, email addresses, and even valuable sales suggestions—all of it would be exposed to the hacker's prying eyes.

After some time, a kind-hearted individual informed the shop owner about the security vulnerability of their website. They explained that the shop's website was using an insecure connection (HTTP), allowing the hacker to access all the data submitted by clients. This revelation left the shop owner startled, and they were eager to learn more about the difference between HTTP and HTTPS.

The good Samaritan started explaining the concept of HTTP and HTTPS in simple terms. They said, "HTTP is like sending your data on a postcard through the mail. Anyone who intercepts it can read what's inside, just like that hacker did with your website. On the other hand, HTTPS is like sending your data in a sealed, tamper-proof envelope. It ensures that your information remains private and secure during transmission."

The good guy further advised the shop owner on how to add HTTPS to their webpages. They recommended obtaining an SSL/TLS certificate, which is like a digital ID card for websites, proving they are secure. This certificate is essential for enabling HTTPS. The good guy explained that once the SSL/TLS certificate is installed, the website will display "HTTPS" in the browser's address bar, reassuring visitors that their data is safe.

With the newfound knowledge about online security and the importance of HTTPS, the shop owner quickly took action. They secured their website, making it a safer place for customers to share their valuable

information. This not only protected their clients but also enhanced the shop's reputation and encouraged more people to visit and contribute their suggestions for boosting sales.

In this cybersecurity journey, the shop owner learned a valuable lesson about online safety, thanks to the guidance of the good-hearted individual who showed them the way to protect their digital storefront and customer data.

Before we delve into the implementation of HTTPS on a webpage, let's familiarize ourselves with some fundamental concepts.

Encryption: Imagine you're sending a secret message to your friend, but you don't want anyone else to read it if they intercept it. Encryption is like putting your message inside a locked box. Only your friend, who has the key to unlock the box, can read the message. In the digital world, encryption scrambles your information into a code that can only be deciphered with the right key, making it secure from prying eyes.

Encryption can be implemented in two ways:

i) Symmetric Key
ii) Asymmetric Key

Symmetric Key: Think of a symmetric key as a special kind of key that both you and your friend can use to lock and unlock the box with your secret message. It's like having the same key for your diary's lock that only you and your friend have. You both know the secret key, and you can use it to encrypt and decrypt your messages. It's called "symmetric" because it's the same key for both locking and unlocking, like a mirror image. This makes it efficient for quick and secure communication between you and your friend.

One of the algorithms for symmetric key is AES.

In Symmetric Encryption, we have different kinds of Cipher modes available like **ECB, CBC, CFB, OFB, CTR.**

Suppose there is a requirement to encrypt some data. How can we do it? For this we have one command available. "**openssl"** this command is main command, and it contains sub command.

```
[root@ip-172-31-12-48 ~]# openssl version
OpenSSL 3.0.8 7 Feb 2023 (Library: OpenSSL 3.0.8 7 Feb 2023)
[root@ip-172-31-12-48 ~]#
```

When we use help option in openssl command then it shows 3 Types:

```
[root@ip-172-31-12-48 ~]# openssl --help
help:

Standard commands
asn1parse         ca                ciphers           cmp
cms               crl               crl2pkcs7         dgst
dhparam           dsa               dsaparam          ec
ecparam           enc               engine            errstr
fipsinstall       gendsa            genpkey           genrsa
help              info              kdf               list
mac               nseq              ocsp              passwd
pkcs12            pkcs7             pkcs8             pkey
pkeyparam         pkeyutl           prime             rand
rehash            req               rsa               rsautl
s_client          s_server          s_time            sess_id
smime             speed             spkac             srp
storeutl          ts                verify            version
x509
```

```
Message Digest commands (see the `dgst' command for more details)
blake2b512        blake2s256        md2               md4
md5               rmd160            sha1              sha224
sha256            sha3-224          sha3-256          sha3-384
sha3-512          sha384            sha512            sha512-224
sha512-256        shake128          shake256          sm3

Cipher commands (see the `enc' command for more details)
aes-128-cbc       aes-128-ecb       aes-192-cbc       aes-192-ecb
aes-256-cbc       aes-256-ecb       aria-128-cbc      aria-128-cfb
aria-128-cfb1     aria-128-cfb8     aria-128-ctr      aria-128-ecb
aria-128-ofb      aria-192-cbc      aria-192-cfb      aria-192-cfb1
aria-192-cfb8     aria-192-ctr      aria-192-ecb      aria-192-ofb
aria-256-cbc      aria-256-cfb      aria-256-cfb1     aria-256-cfb8
aria-256-ctr      aria-256-ecb      aria-256-ofb      base64
bf                bf-cbc            bf-cfb            bf-ecb
bf-ofb            camellia-128-cbc  camellia-128-ecb  camellia-192-cbc
camellia-192-ecb  camellia-256-cbc  camellia-256-ecb  cast
cast-cbc          cast5-cbc         cast5-cfb         cast5-ecb
cast5-ofb         des               des-cbc           des-cfb
des-ecb           des-ede           des-ede-cbc       des-ede-cfb
des-ede-ofb       des-ede3          des-ede3-cbc      des-ede3-cfb
des-ede3-ofb      des-ofb           des3              desx
idea              idea-cbc          idea-cfb          idea-ecb
idea-ofb          rc2               rc2-40-cbc        rc2-64-cbc
rc2-cbc           rc2-cfb           rc2-ecb           rc2-ofb
rc4               rc4-40            rc5               rc5-cbc
rc5-cfb           rc5-ecb           rc5-ofb           seed
seed-cbc          seed-cfb          seed-ecb          seed-ofb
zlib
```

In the world of cybersecurity, imagine a company that wants to protect its sensitive data by turning it into a secret code using a special encryption method called "OpenSSL AES" with a key size of 256 bits. They also want to save this coded data in a separate file. Now, a developer has a solution for this.

The developer uses a command-line tool called "openssl" with the following command:

openssl enc -aes256 -e -in plain.txt -out secure.txt

Here's what each part does:

- **openssl enc** is used for encryption.

- **-aes256** specifies the AES algorithm with a 256-bit key size.

- **-e** is used to indicate that we want to encrypt the data.

- **-in** tells the tool to take the input from a file called "plain.txt."

- **-out** specifies the name of the output file where the encrypted data will be saved, in this case, "secure.txt."

After running this command, you'll be prompted to enter a master password. This master password plays a critical role. When you run the command, it generates a random "master key." This master key has the power to both encode and decode the data. To keep this master key safe and secure, you need to enter the master password. It's like having a special key to lock and unlock a secret vault that holds the code for your data. So, the master password is your way of safeguarding this crucial key.

Let’s see the practical for this.

```
[root@ip-172-31-12-48 Cryptography]# cat > plain.txt
I am Plain Text
[root@ip-172-31-12-48 Cryptography]# cat plain.txt
I am Plain Text
[root@ip-172-31-12-48 Cryptography]# openssl enc -aes256 -e -in plain.txt -out secure.txt
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[root@ip-172-31-12-48 Cryptography]# ls
plain.txt  secure.txt
[root@ip-172-31-12-48 Cryptography]# cat secure.txt
Salted__[illegible]root@ip-172-31-12-48 Cryptography]#
```

First, I made a file called "plain.txt" and put some information in it. Then, I turned that file into a secret code using encryption. When I tried to open the file in the usual way, all I saw was a bunch of jumbled-up numbers and letters, which is encrypted data.

Once the data was safely stored in its secret code, I deleted the original "plain.txt" file. Now, if we ever need to see the original information again, we can use the same encryption method to decode it and get back the original data.

```
[root@ip-172-31-12-48 Cryptography]# rm plain.txt
rm: remove regular file 'plain.txt'? y
[root@ip-172-31-12-48 Cryptography]# ls
secure.txt
[root@ip-172-31-12-48 Cryptography]# openssl enc -aes256 -d -in secure.txt
enter AES-256-CBC decryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
I am Plain Text
[root@ip-172-31-12-48 Cryptography]#
```

openssl enc -aes256 -d -in secure.txt

- **openssl enc** is the tool we're using for encryption.
- **-aes256** specifies the AES encryption method with a 256-bit key for decryption.
- **-d** is used to decrypt the data.
- **-in secure.txt** tells the tool to take the encrypted input from a file named "secure.txt."

Now, to unlock and read the data, we need to provide the Master Password. It's like having a secret code to open a locked box and access the valuable information inside.

In our security approach, we rely on a Master Password to protect our data. However, it's important to note that if someone learns the Master Password, they can gain access to our data. It's worth mentioning that the primary purpose of the Master Password is to safeguard the Master Key. So, if we opt to secure our data solely using the Master Key, then there's no need for the Master Password in that scenario.

Let's see about this. I created one file with the name of General-Data.txt and put some content in the file.

```
[root@ip-172-31-12-48 Cryptography]# ls
secure.txt
[root@ip-172-31-12-48 Cryptography]# cat > General-Data.txt
I am General Data and Here I am not Using Master Password to Secure this Data
[root@ip-172-31-12-48 Cryptography]# ls
General-Data.txt  secure.txt
[root@ip-172-31-12-48 Cryptography]# cat General-Data.txt
I am General Data and Here I am not Using Master Password to Secure this Data
[root@ip-172-31-12-48 Cryptography]#
```

For Encrypting this File, I run the Command:

openssl enc -aes256 -e -in General-Data.txt -out Secure-Data.txt -k 123456789 -iv aaaaaaaa

```
[root@ip-172-31-12-48 Cryptography]# openssl enc -aes256 -e -in General-Data.txt -out Secure-Data.txt -k 123456789 -iv aaaaaaaa
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
hex string is too short, padding with zero bytes to length
[root@ip-172-31-12-48 Cryptography]# ls
General-Data.txt  Secure-Data.txt  secure.txt
[root@ip-172-31-12-48 Cryptography]# cat Secure-Data.txt
Salted__ [illegible] #z/6Mt)m] [illegible] [root@ip-172-31-12-48 Cryptography]#
```

- **-k 123456789:** This is the encryption key (password) you've chosen for the encryption. In this case, the key is "123456789."

- **-iv aaaaaaaa**: This is the initialization vector (IV) used for encryption. The IV is like an additional layer of security to make the encryption more robust.

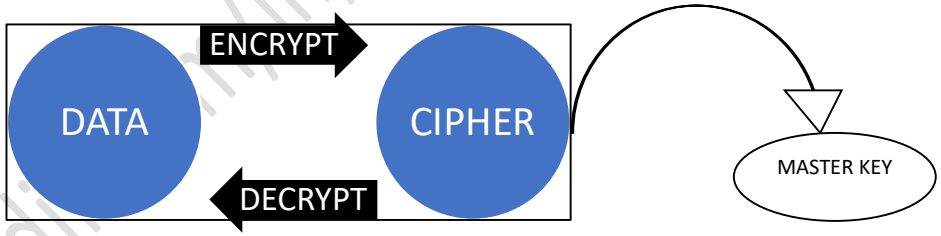
So, when you run this command, it will encrypt the contents of "General-Data.txt" using the AES-256 algorithm with the specified key and IV and store the encrypted data in "Secure-Data.txt."

Here You can see that, it won't ask for the master Password and Store the data in encrypted form.

Now If we have a requirement to see the original data without using any Master Password we can see that with the help of Master Key.

```
[root@ip-172-31-12-48 Cryptography]# openssl enc -aes256 -d -in Secure-Data.txt -k 123456789 -iv aaaaaaaa
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
hex string is too short, padding with zero bytes to length
I am General Data and Here I am not Using Master Password to Secure this Data
[root@ip-172-31-12-48 Cryptography]#
```

When I **encrypt** this data then this data is converted into Cipher Text and for Viewing the Cipher Text in normal Form, we need that master Key.



Let's see that What Master Key contains, for this run the below command and this command require 1 Master Password, so give any password you want here I am giving **test** as Master Password.

openssl aes-256-cbc -P

```
[root@ip-172-31-12-48 Cryptography]# openssl aes-256-cbc -P
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
salt=2AB5B94F617B64DA
key=F1F453E5B9D135F54F91A8DFED6C469F77654BDC925DC4143B48270CECF09384
iv =F7678AA22B99AFA9172FFDF631A228D5
[root@ip-172-31-12-48 Cryptography]#
```

Here This key contains Salt values, Key values, IV values and if we run the same command and give the same value as **test.** Then next time the values will be changed.

```
[root@ip-172-31-12-48 Cryptography]# openssl aes-256-cbc -P
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
salt=B3794E47112A9F45
key=458666DE8A14127C1C71B1806C3770D7C64A18B0BC7328E88A8B6AD22799100E
iv =9FBCA23AE9BC0AD7BDC2E136CEE91D18
[root@ip-172-31-12-48 Cryptography]#
```

Here I am using Key size is 256. So, if any hacker wants to break this key using the current Operating System, then it takes 1000 of years to break it.

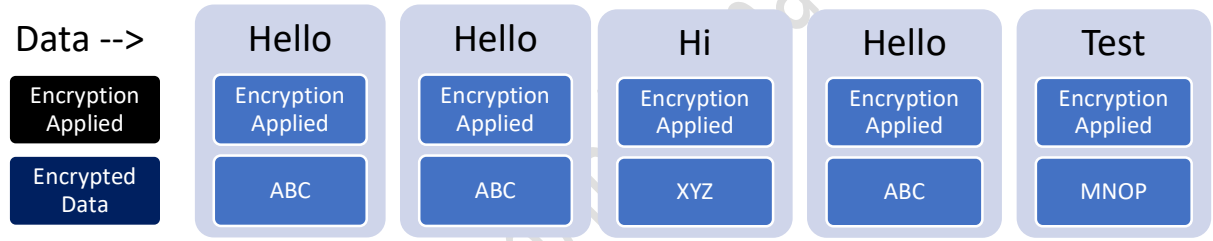
In a Symmetric algorithm, when we have data in plain text and the file size is 111 bytes, the encryption process occurs in blocks. If the block size is set to 64 bytes, then encryption will be carried out in two blocks: one of 64 bytes and another of 47 bytes.

Block Size may be different From Algorithm to Algorithm.

DES = 64 bit.

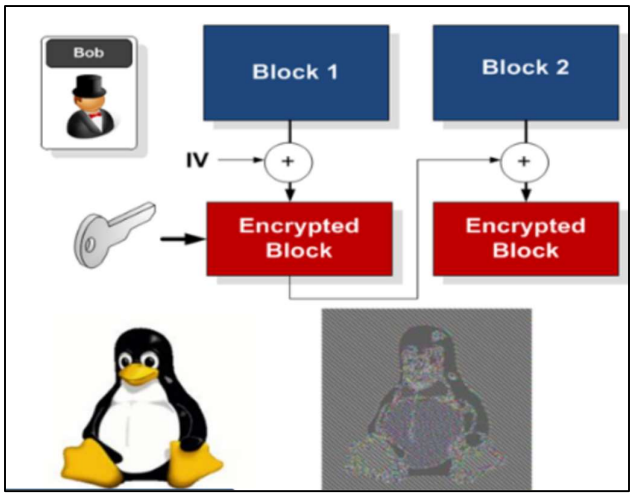
AES = 128 bit.

Suppose In file I have put Some Data and the data is stored in some block



Here every block is individual so we can do Parallel computing, so encryption will be done faster. This is known as Electronic Code Book (ECB).

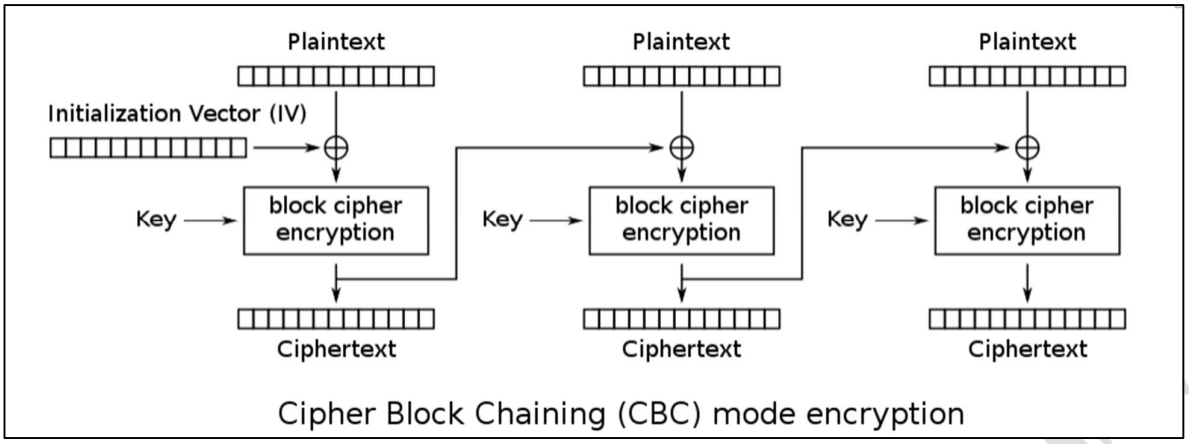
Electronic Code Book (ECB).



ECB is a basic encryption mode used in cryptography. It divides the data into fixed-size blocks and encrypts each block individually using the same encryption key. However, because identical blocks of data always produce the same ciphertext, it lacks diffusion, which means that patterns in the plaintext may remain visible in the ciphertext. This can make it less secure, especially when encrypting large datasets with repeated patterns, like images or videos.

The Electronic Codebook (ECB) encryption mode has a notable weakness: when similar blocks of data are encrypted, they produce identical ciphertext. If we need to encrypt image or video data, this predictability in the ciphertext becomes a vulnerability. Hackers can exploit this to gain insights into the encrypted data, making it easier for them to discern the nature of the information. This problem is solved by Cipher Block Chaining (CBC).

Cipher Block Chaining (CBC)



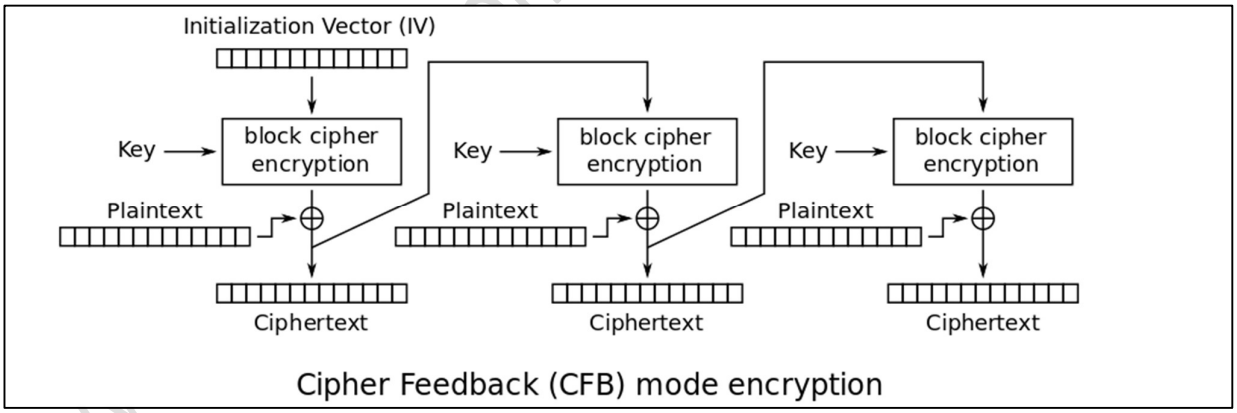
When we're encrypting plain text using Cipher Block Chaining (CBC), we add something called an Initialization Vector (IV), which is essentially random data.

Here's what you need to know about CBC:

- It's slower because it encrypts one block of data at a time, and we can't start on the next block until the current one is encrypted. Think of it like a series of dominoes falling one after the other.
- It's not recommended for big data because the slow, sequential encryption process can be a bottleneck for large datasets.
- Initially, we need to provide a random value as the Initialization Vector.
- The good news is that decryption in CBC can be done in parallel, meaning multiple blocks can be decrypted simultaneously, which speeds up the process compared to encryption.

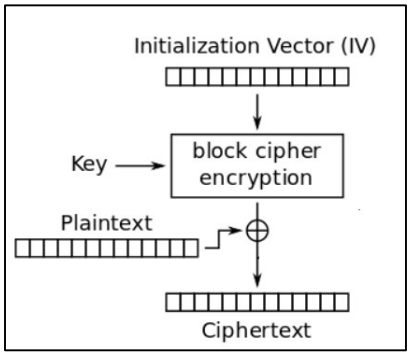
If we don’t give the Initialization Vector then it throws an error.

Cipher Feedback (CFB) Mode

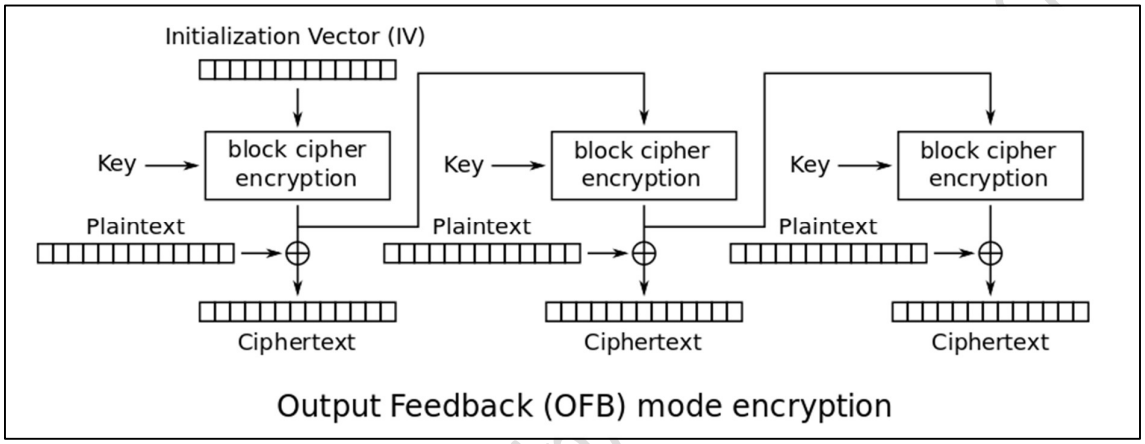


Suppose in CFB Mode, Conversion of 1 Block of size 64 bytes take 1 second and based on this example if we have 1000000 of blocks then that much time is consumed.

- Based on the image, we only need to input data into a precalculated section, and it gives us the encrypted output, known as Cipher Text.
- When dealing with large volumes of data, this process is exceptionally fast.
- It doesn't support parallel encryption, but it can handle parallel decryption.
- This falls into the category of stream ciphers.



Output Feedback (OFB) Mode



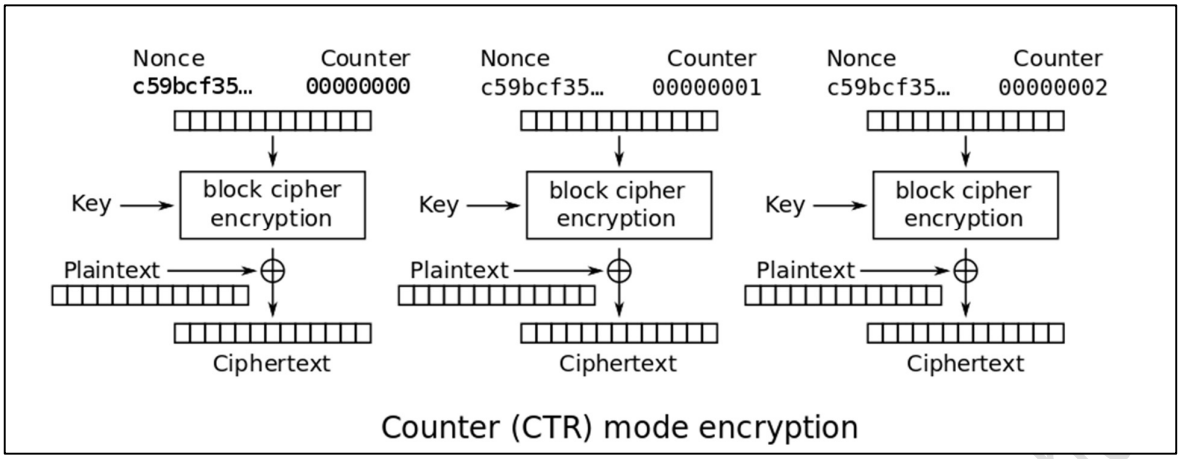
Think of OFB as a way to encrypt data in a kind of "feedback loop."

- We start with an initial value, like a secret code called the "Initialization Vector" (IV).
- We use this IV to create a random-looking stream of data, which we call the "keystream."
- We then combine this keystream with our plain text to produce the encrypted text.
- The unique part is that we keep this process going. Instead of using the previous encrypted text, we continue to use the keystream to encrypt the next block of data. It's like a series of secret codes that keep changing with each block.
- This "feedback loop" ensures that even if someone figures out one part of the code, they can't easily crack the rest because it keeps evolving.

In a nutshell, OFB is like continuously changing secret codes to encrypt your data, making it a secure way to protect information.

Counter (CTR) Mode

In Counter (CTR) Mode, we take a random number (similar to a nonce) and combine it with a counter. Then, we use this combined value to perform encryption with a cipher. After encryption, we add the resulting cipher output to the plaintext to get the final encrypted text. One notable feature of CTR Mode is that it allows for both parallel encryption and parallel decryption, making it efficient for handling data.



Counter (CTR) mode encryption

- **Random Number and Counter:** In CTR Mode, we start with a random number (think of it like a special secret) and mix it with a counter. This combined value is used for encryption.
- **Encryption Process:** After combining the random number and counter, we use this mixture to do the encryption. It's like putting your message in a secret code.
- **Adding to Plain Text:** Once encryption is done, we take the result and add it to the original plain text. This gives us the final encrypted text.
- **Parallel Encryption & Decryption:** CTR Mode is pretty clever because it allows us to encrypt and decrypt multiple things at the same time, which can be super speedy and efficient. It's like having many workers doing their tasks simultaneously, getting things done faster.

In the world of cybersecurity, Encryption and Authentication serve different purposes. **Authentication is about proving your identity,** while **Encryption is about keeping data secret.**

Symmetric Key: Think of this like a single secret key that both you and your friend use to lock and unlock a box. It's quick and efficient for authentication.

Asymmetric Key: This one is a bit more complex. Under Asymmetric Key, we have two keys:

- **Private Key:** This is like your own special secret key that you keep to yourself.

- **Public Key:** This key is like a special lock that you give to others. They can use it to send you secret messages that only you can unlock.

Now, let's see how authentication works with a Symmetric Key:

Imagine 'A' wants to send a message to 'B,' but there's a sneaky hacker watching their internet activity. This hacker tries to trick 'B' by pretending to be them, a sneaky tactic known as "IP Spoofing." They change their IP address to look like 'B.'

But here's the catch: 'A' and 'B' share a secret symmetric key for encryption and decryption. This key is like a special secret handshake that only 'A' and 'B' know.

So, even though the hacker can intercept the message, they don't have that secret key. Without it, they can't unlock and read the information inside the message. They also can't make changes to it without the secret key.

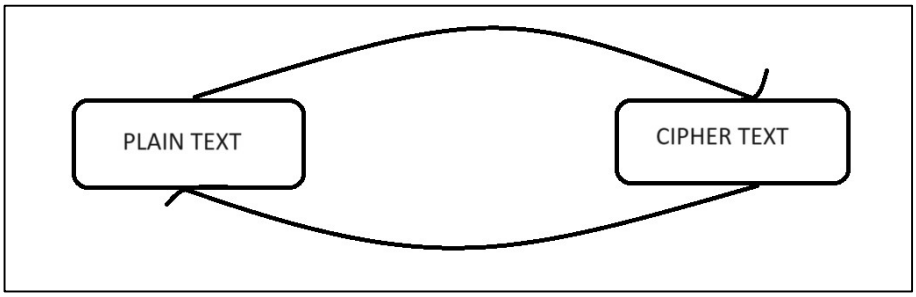
In the end, the hacker's attempt to access the message fails because they lack the secret key needed for authentication. So, in the world of cybersecurity, it's not just about encryption; it's also about having the right keys to prove who you are and keep your information safe.

```
[root@ip-172-31-45-173 Cryptography]# cat plain.txt
This is plain Text in unencrypted form.
[root@ip-172-31-45-173 Cryptography]# openssl enc -aes256 -e -in plain.txt -k 123456789 -out Secure.txt
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
[root@ip-172-31-45-173 Cryptography]# ls
Secure.txt  plain.txt
[root@ip-172-31-45-173 Cryptography]# cat Secure.txt
Salted__[root@ip-172-31-45-173 Cryptography]# rm plain.txt
rm: remove regular file 'plain.txt'? y
[root@ip-172-31-45-173 Cryptography]# openssl enc -aes256 -d -in Secure.txt -k 123456789
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
This is plain Text in unencrypted form.
```

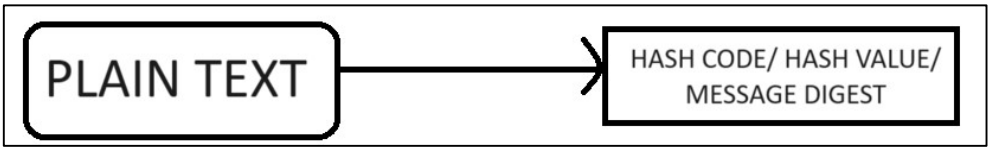
Here in Symmetric Algorithm, we are using a single key for encryption and decryption and, we can use this for authentication purposes. But here we have one problem if someone gets our key then it is easier to decrypt the original data. To solve this problem, we have the solution of **Hashing**.

HASH

Encryption works in two directions. It can transform plain data into ciphertext, and conversely, it can convert ciphertext back into plain data.



However, when we use Hashing, it operates in a one-way manner. It transforms plain text into a new data format, which we call a **Hash value** or **Hash code**.



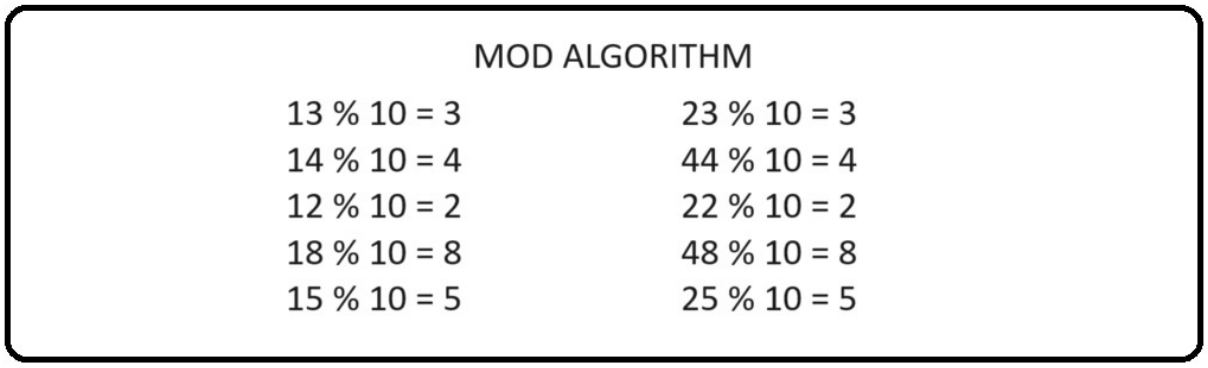
How Hashing Work?

Imagine a bank that collects your Name and Aadhar Number, which are super important because they have all your personal info. They want to protect this info, so they use something called a "Hashing" method.

Here's how hashing works: When someone like Akash gives his name, the bank's special formula counts the number of words in his name and gives a result, like 5. But this 5 doesn't reveal Akash's name. It's just a code.

But here's the tricky part. If someone else, like Rajeev, also gets a 5, that's a problem. It's like having two people with the same secret code. This is called a "collision issue."

To fix this, they try another method called "Mod Algorithm," but it has its own issues with values sometimes being the same.



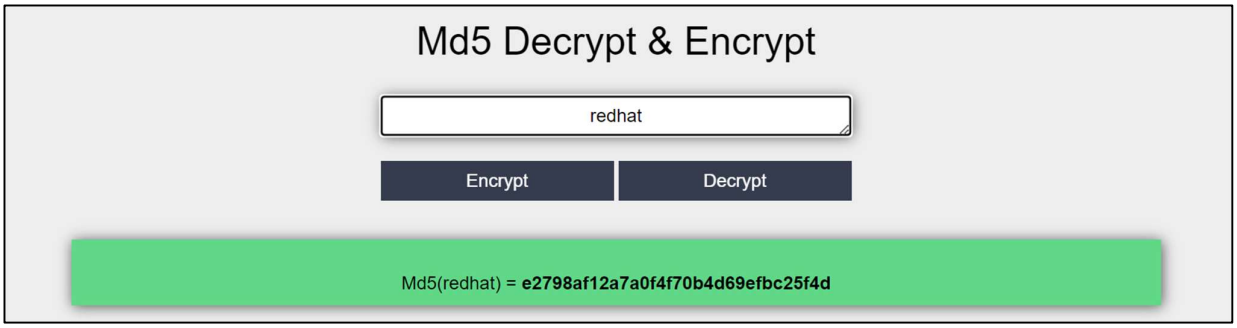
Then there's the famous "MD5 Message Digest" method, but it's not foolproof. In 2013, some smart folks found a way to break it $2^{\wedge 18}$ times, which is not great for security. So, they moved to a more reliable method called "SHA algorithm." It's like getting a better lock for your secret box.

Remember, MD5 and SHA are all about hashing, not encrypting. They're like secret codes to protect your info, but they're not the same as encryption.

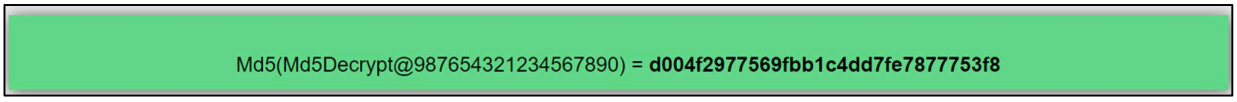
At times, hackers claim they can figure out the original info from a hash code. To show this, they've made something called a "Rainbow table." This table has tons of possible hash codes stored in it. So, when someone gives a hash code, they secretly use search tricks to find the matching hash in their table and reveal the original info.

Here's an important thing to note: If you try to make a hash code for "test" on my computer, it might turn out as "1234567890." But if you do the same on your computer, it won't change; it will still be "1234567890." Hashing is consistent, so it gives the same result no matter where you do it.

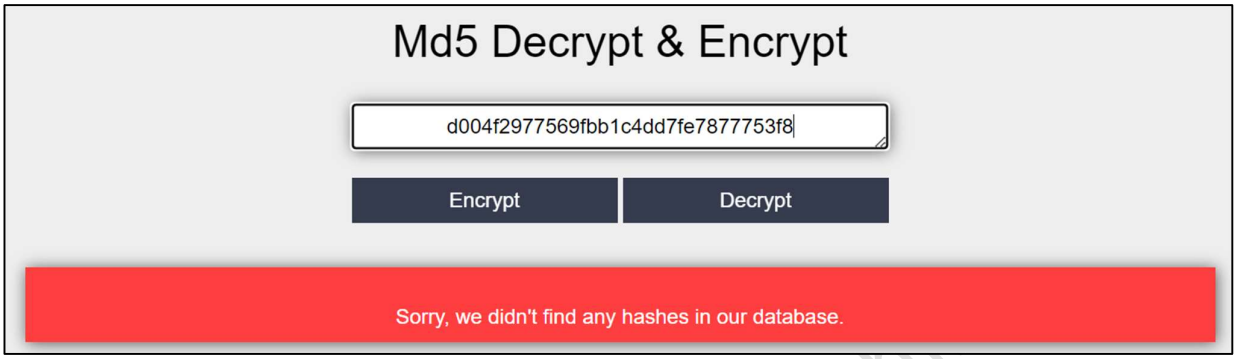
Here for showing this demo I used this Website: https://md5decrypt.net/en/



It gives the value of redhat in the Hash format but if I try to give some other value like Md5Decrypt@987654321234567890 For this I am getting the Hash Value as:



But If I Copy the Same Hash Value and try to decrypt. It won't be able to find it this is not present in their database. That is the reason You should Use some very strong password.



In MD5 and other cryptographic contexts, "salt" refers to random data that is generated and added to a password or data before it is hashed. The purpose of using salt is to enhance security, particularly against attacks like precomputed rainbow table attacks.

Hash also provides the solution for **AUTHENTICATION** and **DATA INTEGRITY.**

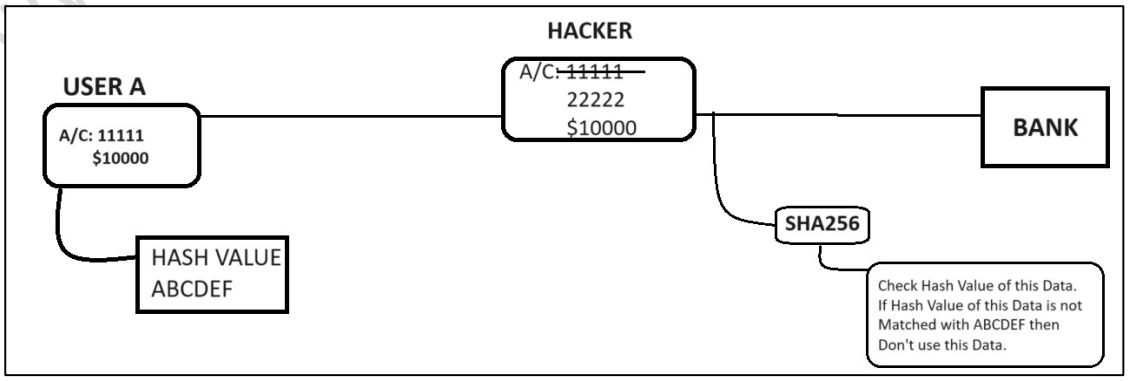
Authentication: authentication typically refers to the process of verifying the identity of a user or entity by comparing a provided hash value with a stored or expected hash value.

Data Integrity: Data integrity refers to the accuracy and reliability of data over its entire lifecycle. It ensures that data remains unchanged and uncorrupted during storage, transmission, or processing. In essence, data integrity guarantees that data has not been tampered with, altered, or corrupted accidentally or intentionally.

Checksums are commonly used in the context of data integrity to verify the integrity of data. A checksum is a value that is calculated from the content of data using a specific mathematical algorithm. This value is then compared to a previously calculated checksum or a known reference checksum to check if the data has remained intact.

If the checksums do not match, it suggests that the data may have been corrupted or tampered with in some way. This is a red flag for potential data integrity issues, and further investigation may be necessary.

Checksums are commonly used in various scenarios, including data transmission over networks, file storage, and error detection in data processing to ensure data remains reliable and unaltered. They are a fundamental tool in maintaining data integrity and can help identify and mitigate potential data corruption or tampering.



Once upon a time, there was a person named User A who wanted to open a bank account. User A went to the bank, shared their account details, and the bank stored this information in a special code called a hash.

Later, when User A decided to transfer some money to Account Number 11111, something tricky happened. A sneaky hacker tried to interfere, performing what's known as a "Man in the Middle" attack. The hacker changed the recipient's account number to their own. But here's the twist: the hacker didn't realize that User A's data was stored in hash format.

When the hacker made changes, the hash value also changed. When the bank received this tampered data, they realized something fishy was going on. They informed User A that there had been an attempted attack, and the money transfer was halted.

You see, when User A has some data, let's say it's represented by the hash value "ABCDEF," and the bank receives the same hash value, it means the data is intact and hasn't been tampered with. This is called data integrity, and it's checked using a checksum.

But there's a challenge. If User A wants to transfer money to an account not registered with the bank, they might think of sending bank details via network and the hash code via SMS. However, our sneaky hacker is watching. They capture both sets of information and replace it with their own bank data. This time, the bank doesn't know User A's actual hash value, so they transfer the money to the hacker's account, thinking it's User A.

This situation teaches us that relying solely on data integrity is not enough because documents and hash values can be altered by hackers. To solve this problem, we use two concepts together:

i) Message Digest
ii) Authentication.

When you combine these two things, it becomes a "**MAC**" or **Message Authentication Code**. Now, User A needs to use a secret key provided by the bank. When they want to transfer money, they include their details, the shared key, the account number, and the transfer amount, all in hash format. The beauty is, the hacker doesn't know the key value or the real data. If they change either the key or the data, the hash value will change too.

MAC is a powerful concept that helps maintain both authentication and integrity. Some famous algorithms used for this purpose include HMAC and GCM. It's like having a secret handshake between User A and the bank, making sure no sneaky hackers can trick their way in.

Earlier, we learned that in Symmetric Key Encryption, we use just one key for both encrypting and decrypting data. That's convenient, but if a hacker gets hold of this key, they can easily access the user's data.

However, when we use an Asymmetric Key Algorithm, things work differently. Here, we have two keys: a private key and a public key.

- The **private key** is like a secret key that we keep very safe. We use it to encrypt the data.

- The **public key**, on the other hand, is used to decrypt the data and make it readable again.

So, with these two keys, even if someone knows the public key, they can't easily decrypt the data because the real secret is the private key. It's like having two locks on a treasure chest, and only one of them can open it.

When it comes to creating keys for security, here's a fundamental rule to remember:

1. **Private Key First:** Always start by creating the private key. This is like having your own super-secret key.
2. **Public Key from Private Key:** From that private key, you can generate a public key. Think of it as a special code that pairs with your private key.

Here's the important part: When you create a new private key, it's like getting a fresh, brand-new secret. But when you generate the corresponding public key from that private key, it stays the same every time you do it.

So, to keep it simple, private keys are unique secrets, and their associated public keys are like locked doors that always match that one specific private key.

For creating the Private key, we have command available: **openssl genrsa 1024**

```
Himanshu Kumar@AICPL-L128 MINGW64 ~/Desktop/ADO/Cryptograpgy
$ openssl genrsa 1024
-----BEGIN PRIVATE KEY-----
MIICdwIBADANBgkqhkiG9w0BAQEFAASCAmEwggJdAgEAAoGBAMTFMMnfyxDKVboI
YkPukehjsU+Y5ygoPL7sRsZa9hjM37LqExxaSfDCqyWvknm+CLOuqbV47NAjFqrR
JIB1GupHoG9tA9v7myWu1ThV0J6eHcdzv/gwQOOD3OLlxlv3LsAZMFuKXSlV1vcI
gl+sdcP+8WGHis7lg1BAxZW+R4ZxAgMBAAECgYBuWSuYOsH1o5LZ5jO9ZwA6dBYO
zm/r7EzMwVKbkI1ssRBOlMis4NGQjItQchrR3UmiArS3yZ38x4DBwNIYnBPADTKD
q8DhUP3mioEa5HlVLnJgO2HzumQoHydn9R8Szxt55styn7jOww+YK5duCEtLXHB+
T3cfLxvo6blOnRFeAQJBAOOtj+kFyYb5FnqsFhXtQ2frfYmFu2+f5sOBJPDqEnOk
+HXMxJrWV+pdQujwZzrYA/vUpRPfqAY82byA/KOOPZECQQDdP15FOSb7y6OkFV7B
DaG+vFMcaXukDPDU1wk99JNvK4/imQZ+GxEOilJMuQrJImipJBYeMWKFu/EIsSF/
OcrhAkEAiefHEO3+E1Ms2oEBn6Zm/bsV/waRObr4+9Jtnb8qE8Mg+7lcHKlX8w0q
Sg3TeUD+qLrUgoUmMSEKgbNfU4kJAQJAfWKNNWPXYPmear4lrfjgwmyOGVr6VBV1
YtnfBfqWl3kzmPKqSEtklj3zuEFiCGzOrW6jDkdkk2/YgNmJpOWEYQJBAJoxHhs2
zO9x+3jOynHXN2FSX/NzSw3AGccDFFOMQ6IKlsg5+BfzwBLa4V3c4wAyjQvwuYSv
EVHpcHvgNIVS/cE=
-----END PRIVATE KEY-----
```

The command "openssl genrsa 1024" is used to generate an RSA (Rivest–Shamir–Adleman) private key with a length of 1024 bits using the OpenSSL tool. RSA is a widely used public-key cryptosystem for secure data transmission and digital signatures.

When I rerun this command then it will give new key value.

```
$ openssl genrsa 1024
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAKKnwDFeedtpURp1
yltqbYr8HrT4CTKhJQ4iI3G55i1ZsVkWsoSt1RbFCMDU2dwwvO/pBYoP7LbjZI8c
6CWztzXtH5xL6OEM6MHtAu4wM+xYScQUugUR4wVOFshhixegOGqYnhLYDliuyBc/
OGBybnruhlP1KhuIa15ZO1oAvakFAgMBAAECgYAbRf+PB6VtKfqkci3NKEpX7LCF
bbW1HJonBYhl8GLtJcX8BzdFOhQR7lXqZ4ZkSnPTOoHQRZ1OiVB6QsGWWMOWZE5M
/5zybGlynCvMMX5zTNt8NRcZmjQaohmnxlifptFc1rEOYLVzHzjdaPmp3bk2f6Vb
dZqm5gDb+PZs8hO2HQJBAM1Cr9t67Nuo1rvKbXVkisWuhOCTNE+P1S7CcFZF7xI8
2f/A64I1DWVIkQMRIDgFxJ/SnA495q1HgCI6jJX8Lz8CQQDK3OqfxK8ida94W3Yx
cRVpKvVjdWxkisDdTlYiOxJAArOQLGeOS37Bcx27lLMFnk+lKCw4UP8ktu+3daCn
VVq7AkEAy6wuX7S8LsJ8QN7+egwH/1vKkiS/NWtdeAaL/a9rKEelcTf2+L63yhaV
fduSgsSGdKhBHYiiyrI2HMDIG1OfdQJAcho8Jc09LMbBAZOAMWMHxfWRvHWJmCur
OO8ah3puU26xjQBM/E4SZ8DWEnIbF6fp6F1mMe7s87uprakNydDqjQJAZpdkIriB
PKUwGkMPq+E4GtoHviVJN2X1IaCfvDg5uq+rEmhiU9XHja+9Oh7bra8711w59mNP
1dQE3ndo9P77lw==
-----END PRIVATE KEY-----
```

Let's store this Private key in one file and then I am going to show how we can create public key from that private key.

```
Himanshu Kumar@AICPL-L128 MINGW64 ~/Desktop/ADO/Cryptograpgy
$ ls
plain.txt

Himanshu Kumar@AICPL-L128 MINGW64 ~/Desktop/ADO/Cryptograpgy
$ openssl genrsa 1024 > MyPrivate.key

Himanshu Kumar@AICPL-L128 MINGW64 ~/Desktop/ADO/Cryptograpgy
$ ls
MyPrivate.key  plain.txt
```

In this command I have store my Key in MyPrivate.key file and let's see how to create public key using file MyPrivate.key

```
Himanshu Kumar@AICPL-L128 MINGW64 ~/Desktop/ADO/Cryptograpgy
$ openssl rsa -in MyPrivate.key -pubout
writing RSA key
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDMxIIhbbJjWRqVEZhaHNHcPYDP
TMWqN/hMsOHdqIHt4ZSP+mivl6oEpqnoYSHyQ50XJd68JH6NgynoML8rr+x+Daad
xEwh96Fgd+SUBzgig5XYIvzpcGR2eCoy0JZTCgt4KtdHUmz2chF1lxkkoJGPJBbh
ijRlL/hHUB4INJP25wIDAQAB
-----END PUBLIC KEY-----

Himanshu Kumar@AICPL-L128 MINGW64 ~/Desktop/ADO/Cryptograpgy
$ openssl rsa -in MyPrivate.key -pubout > MyPublic.pub
writing RSA key

Himanshu Kumar@AICPL-L128 MINGW64 ~/Desktop/ADO/Cryptograpgy
$ ls
MyPrivate.key  MyPublic.pub  plain.txt
```

The command "openssl rsa -in MyPrivate.key -pubout" is used with OpenSSL to generate a public key from an existing private key stored in a file named "MyPrivate.key."

Here's what each part of the command does:

- **openssl rsa:** This is the OpenSSL command for working with RSA keys.
- **-in MyPrivate.key:** This part specifies the input file, which is the private key file named "MyPrivate.key." The command will read the private key from this file.
- **-pubout:** This option tells OpenSSL to output the corresponding public key. After executing this command, OpenSSL will generate the public key that corresponds to the private key in "MyPrivate.key"

I store this Public key in a file with the extension of **pub.**

NOTE: It is not recommended to share your private key to anyone. Because from public key we can not generate the private key but from private key we can easily get the public key.

We can encrypt our data using public key. Let's see:

Here in the below image, I have created one file with the name of plain.txt and then I have stored the data as redhat in plain text format. Then I encrypt this data with this command.

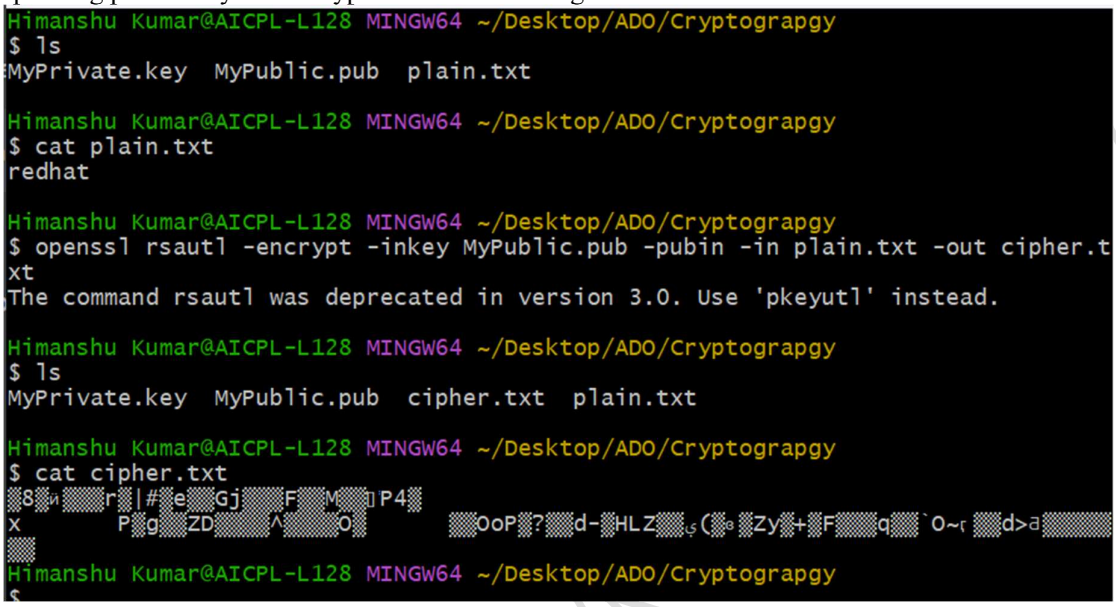
```
openssl rsautl -encrypt -inkey MyPublic.pub -pubin -in plain.txt -out cipher.txt
```

The command is used with OpenSSL to perform RSA encryption on a file named "plain.txt" using a public key stored in a file named "MyPublic.pub." The encrypted result is saved in a file called "cipher.txt."

- `**openssl rsautl**`: This is the OpenSSL command for performing RSA utility operations.
- `**-encrypt**`: This option specifies that you want to perform encryption using RSA.
- `**-inkey** MyPublic.pub`: Here, you specify the public key file ("MyPublic.pub") that will be used for encryption.

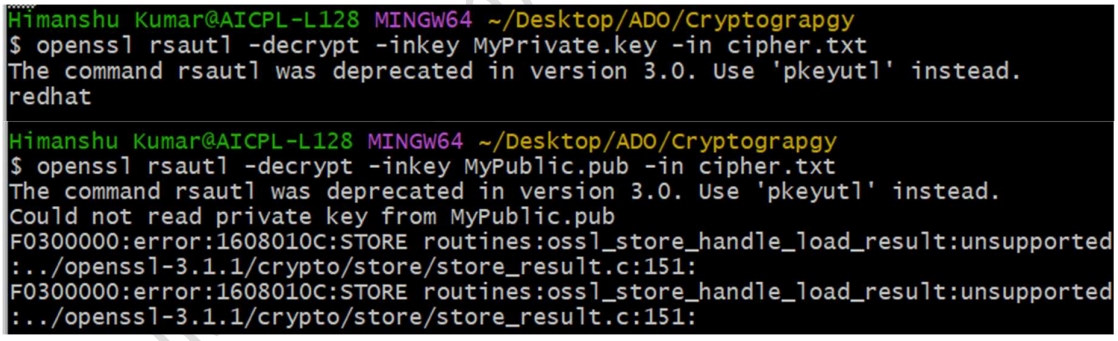
- `**-pubin**`: This option indicates that the input key file is a public key. In this case, it's used to confirm that "MyPublic.pub" is indeed a public key.
- `**-in plain.txt**`: This specifies the input file containing the data you want to encrypt, in this case, "plain.txt."
- `**-out cipher.txt**`: This is where you specify the output file ("cipher.txt") where the encrypted result will be saved.

When you run this command, OpenSSL will encrypt the contents of "plain.txt" using the public key from "MyPublic.pub" and save the encrypted data in "cipher.txt." This process ensures that only someone with the corresponding private key can decrypt and access the original data.

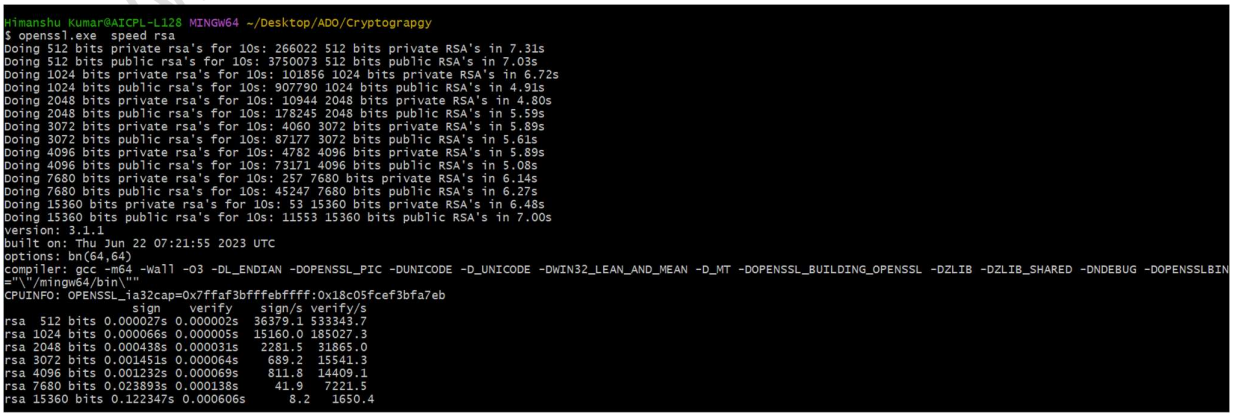


And For Decryption we need to only use the Private key then we can be able to decrypt it.

```
openssl rsautl -decrypt -inkey MyPrivate.key -in cipher.txt
```



If we try to use public key for decryption it will give the error.



```
Himanshu Kumar@AICPL-L128 MINGW64 ~/Desktop/ADO/Cryptograpgy
$ openssl.exe  speed aes-128-cbc
Doing aes-128-cbc for 3s on 16 size blocks: 241672519 aes-128-cbc's in 1.91s
Doing aes-128-cbc for 3s on 64 size blocks: 83041670 aes-128-cbc's in 2.22s
Doing aes-128-cbc for 3s on 256 size blocks: 21740232 aes-128-cbc's in 2.22s
Doing aes-128-cbc for 3s on 1024 size blocks: 5530159 aes-128-cbc's in 2.05s
Doing aes-128-cbc for 3s on 8192 size blocks: 688510 aes-128-cbc's in 1.97s
Doing aes-128-cbc for 3s on 16384 size blocks: 328539 aes-128-cbc's in 2.05s
version: 3.1.1
built on: Thu Jun 22 07:21:55 2023 UTC
options: bn(64,64)
compiler: gcc -m64 -Wall -O3 -DL_ENDIAN -DOPENSSL_PIC -DUNICODE -D_UNICODE -DWIN32_LEAN_AND_MEAN -D_MT -DOPENSSL_BUILDING_OPENSSL -DZLIB -DZLIB_SHARED -DNDEBUG -DOPENSSLBIN
="\"/mingw64/bin\""
CPUINFO: OPENSSL_ia32cap=0x7ffaf3bfffebffff:0x18c05fcef3bfa7eb
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes  16384 bytes
aes-128-cbc    2028464.42k  2395342.82k  2508394.09k  2766599.24k  2864901.04k  2629756.57k
```

We can also test how fast an algorithm works. The results I showed in the picture above are from my computer. If your computer has a more powerful CPU, you might notice a difference in the time it takes.

Problem in Asymmetric Algorithm

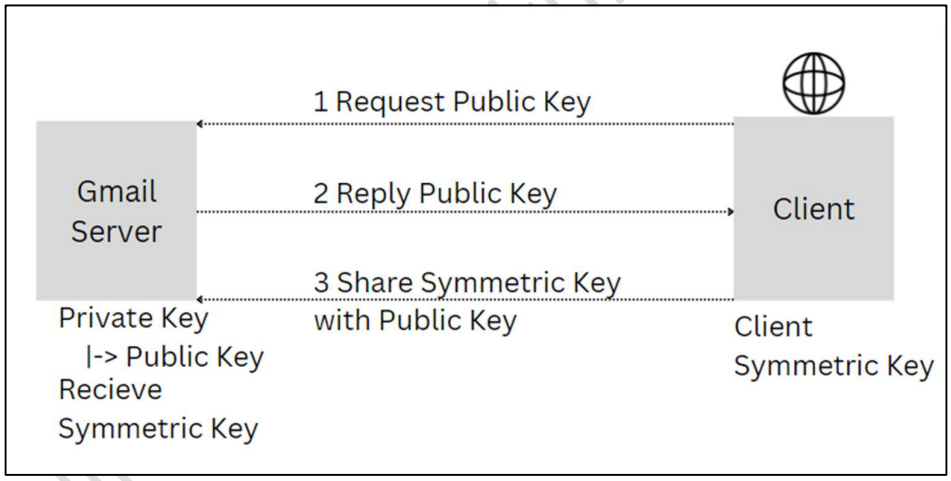
In the world of cybersecurity, there's a problem with something called the Asymmetric algorithm. It's not very fast, and here's why:

Imagine you're sending a big file, let's say 1 MB in size, to someone. Without using Asymmetric encryption, it takes just 1 second to reach them. But if you decide to use the Asymmetric algorithm to encrypt that same 1 MB file, it suddenly takes a whopping 1 minute to get to its destination. That's pretty slow, especially when we're used to things happening in a snap.

So, in terms of performance, Asymmetric encryption can be a bit sluggish compared to symmetric method.

For this we need to take best from Symmetric and Asymmetric algorithm, and we have one term for this Hybrid encryption.

HYBRID ENCRYPTION



We all know that Symmetric Encryption is super speedy for keeping data safe and unlocking it. But there's a problem: if we just hand over the secret key to the server, a sneaky hacker might snatch it while doing a "Man in the Middle" attack.

Now, imagine this: Instead of handing over the secret key, the client starts the connection. They ask the server for something called a "public key." But guess what? If there's a hacker lurking around, they'll grab that public key too.

But here's where it gets interesting. To unlock the data, you need something called a "private key." The hacker didn't get their hands on that one. So, when the client wants to send data, they lock it up with the server's public key. But to open that lock and read the data, you need the private key.

Now, the hacker has the locked data, but they can't open it because they're missing the key. So, even if they have the data, it stays safe and sound.

Let’s deep dive into the Private key. What Private Key Contains?

When we take a closer look at a private key, we find that it's made up of two prime numbers. Additionally, it includes two exponent values along with one coefficient. These numbers work together in a special way, kind of like a secret recipe.

When the private key performs some complex calculations using these numbers, it produces another value, which we call the public key. This public key is like the outcome of those secret calculations and is used for specific security tasks.

```
Himanshu Kumar@AICPL-L128 MINGW64 ~/Desktop/ADO/Cryptograpgy
$ openssl rsa -in MyPrivate.key -text -noout
Private-Key: (1024 bit, 2 primes)
modulus:
    00:cc:c4:82:21:6d:b2:63:59:1a:95:11:98:5a:1c:
    d1:dc:3d:80:cf:4c:c5:aa:37:f8:4c:b0:e1:dd:a8:
    81:ed:e1:94:8f:fa:68:af:97:aa:04:a6:a9:e8:61:
    21:f2:43:9d:17:25:de:bc:24:7e:8d:83:29:e8:30:
    bf:2b:af:ec:7e:0d:a6:9d:c4:4c:21:f7:a1:60:77:
    e4:94:07:38:22:83:95:d8:22:fc:e9:70:64:76:78:
    2a:32:d0:96:53:0a:0b:78:2a:d7:47:52:6c:f6:72:
    11:75:97:19:24:a0:91:8f:24:16:e1:8a:34:65:2f:
    f8:47:50:1e:08:34:93:f6:e7
publicExponent: 65537 (0x10001)
privateExponent:
    26:a3:e2:66:8a:7b:c2:31:9b:49:8c:03:72:e5:c3:
    58:4b:26:d6:77:87:49:62:bc:71:44:63:aa:df:54:
    7d:5e:37:19:a0:48:46:f2:58:bb:fd:f3:4c:0d:a0:
    49:46:27:0b:aa:32:c8:d0:1e:2d:78:15:dd:61:ed:
    81:41:f6:5f:50:ab:e6:0b:a5:b9:04:0d:bf:7d:f9:
    a7:9c:f6:17:11:3c:46:6b:42:35:0e:42:8e:ed:5f:
    c5:e8:1a:1b:41:fc:94:26:d0:80:1a:cf:d6:be:46:
    1b:03:9c:32:82:4a:3f:dd:1d:e7:83:99:96:30:f1:
    c7:c9:d4:3c:76:25:26:09
prime1:
    00:e7:61:10:55:c0:43:2e:fa:e7:08:1c:e0:0f:79:
    29:fe:07:8b:f9:cd:a8:5d:7e:62:b1:96:8d:17:ed:
    02:68:da:67:21:64:b5:3a:60:0c:4b:e1:6b:12:6f:
    29:e4:ef:54:5a:a4:b4:4e:71:83:ee:14:d5:5e:e2:
    2b:04:15:e7:93
prime2:
    00:e2:8e:86:d6:7c:b4:35:3f:aa:66:01:ac:16:c2:
    d7:55:1c:34:72:b2:ec:07:0b:a4:f2:ac:ab:0b:3c:
    02:9b:c5:86:a0:7b:d8:97:b0:79:fe:2d:5d:63:20:
    0d:1a:cc:3a:2d:d3:89:a1:01:0c:0d:f3:bd:e1:01:
    1c:56:86:df:dd
exponent1:
    00:d3:87:d6:92:3e:da:6c:f9:ee:fb:68:c2:a3:02:
    c7:94:39:3c:0c:1d:ab:9e:05:ec:9e:a4:9f:cb:4e:
    e0:14:12:0f:94:e4:e8:16:3d:37:20:ac:2d:e2:45:
    84:cc:8a:cc:d8:ad:99:35:d9:eb:9e:5e:a7:8f:3e:
    3f:cb:55:67:47
exponent2:
    00:b2:f8:dc:d3:fb:d7:70:69:46:f7:26:43:08:08:
    0b:8b:94:a8:6a:23:a6:09:3a:cd:a7:57:78:17:7b:
    e4:a3:a1:ba:e4:74:71:7a:03:53:9b:d3:c2:7e:58:
    07:82:2f:eb:95:e0:12:4d:3a:42:87:93:11:60:fa:
    3b:a4:af:0f:91
coefficient:
    00:e2:6b:ca:9b:39:1b:3c:ff:c7:99:e2:05:d8:7b:
    67:cd:b4:8f:1a:01:e5:39:83:a9:83:76:ce:8f:a4:
    08:df:9d:fd:5a:2d:1b:3a:9b:09:33:29:6b:79:1d:
    4a:c8:bb:ab:3e:0c:1b:09:bc:e3:33:5d:18:7d:db:
    1e:bf:18:46:35
```

Command: openssl rsa -in MyPrivate.key -text -noout

THANK YOU FOR READING