**Name : Dhananjay Salunke**

**UID : 2020401073**

**Course : CSS**

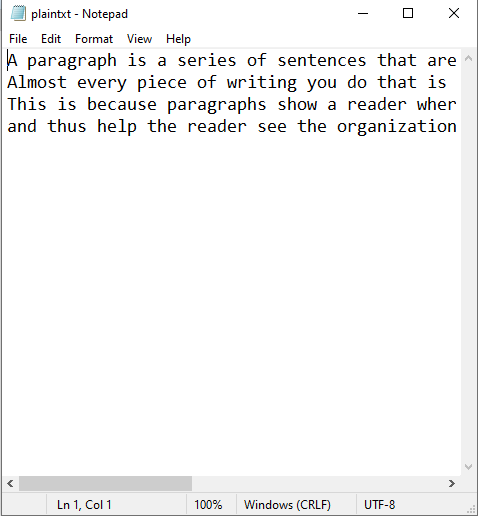
**Experiment No. 3**

**AIM :** To get familiar with the concepts in secret-key encryption also gain first-hand experience on encryption algorithms, encryption modes, paddings, and initial vector (IV). After this lab should be able to use tools and write programs to encrypt/decrypt messages.

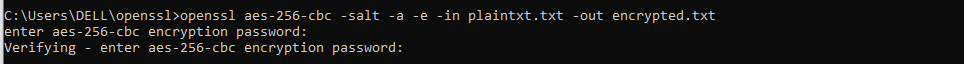
**TASK 1 :**

1. **using the cipher type – aes**

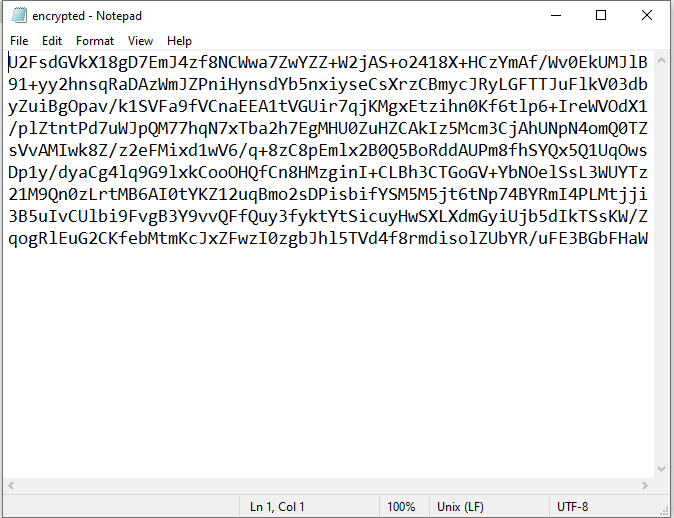
**Plain Text :**

****

**A.1) -aes-256-cbc**

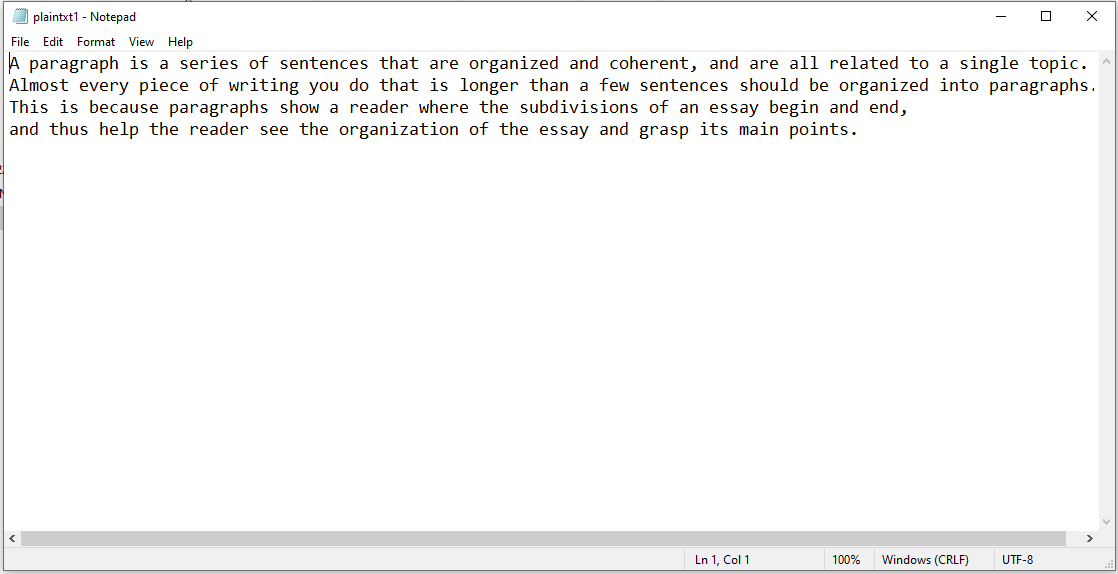
****

**Output:**

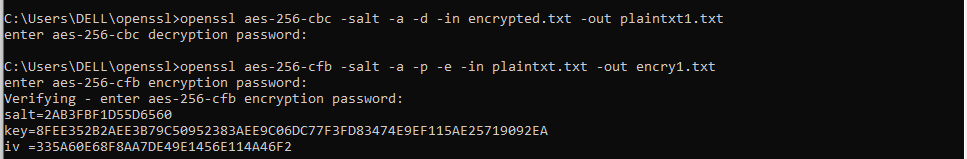
****

**Decryption**

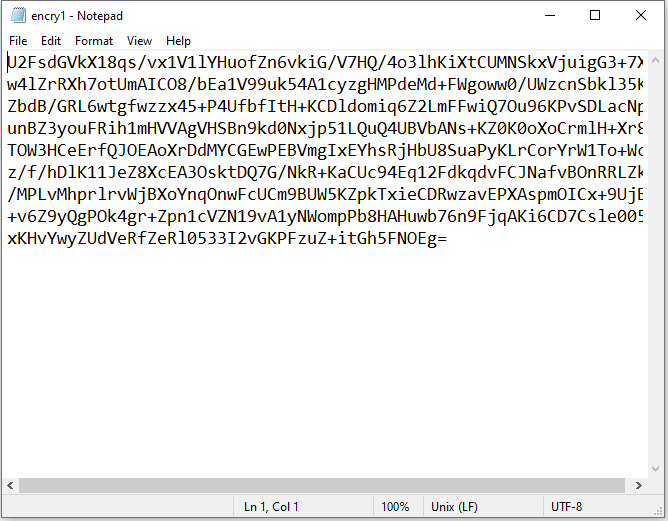
****

****

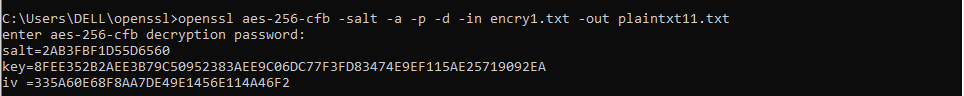
**A.2) -aes-256-cbf**

****

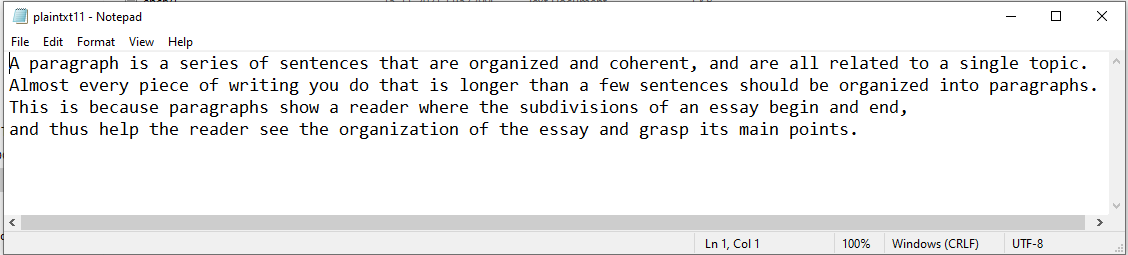
**Output:**

****

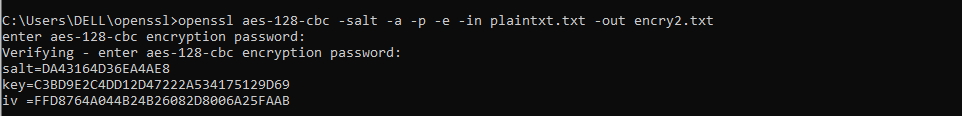
**Decryption**

****

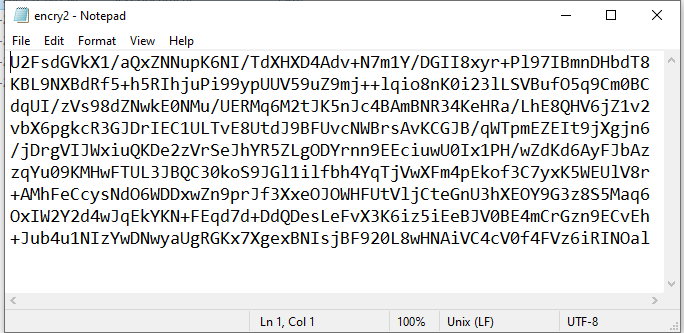
**Output:**

****

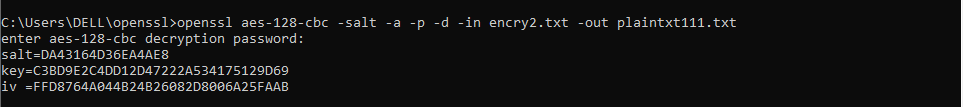
**A.3) -aes-128-cbc**

****

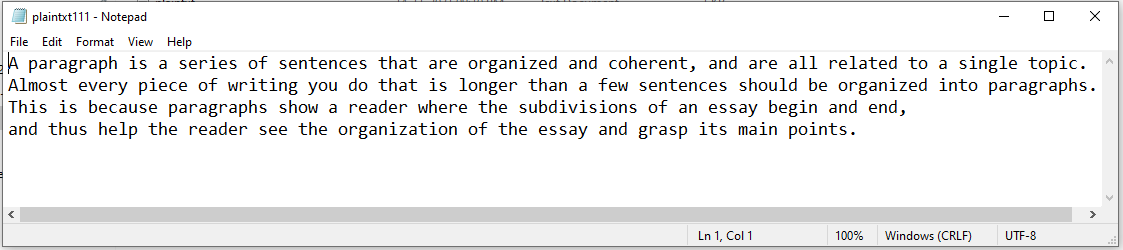
**Output:**

****

**Decryption**

****

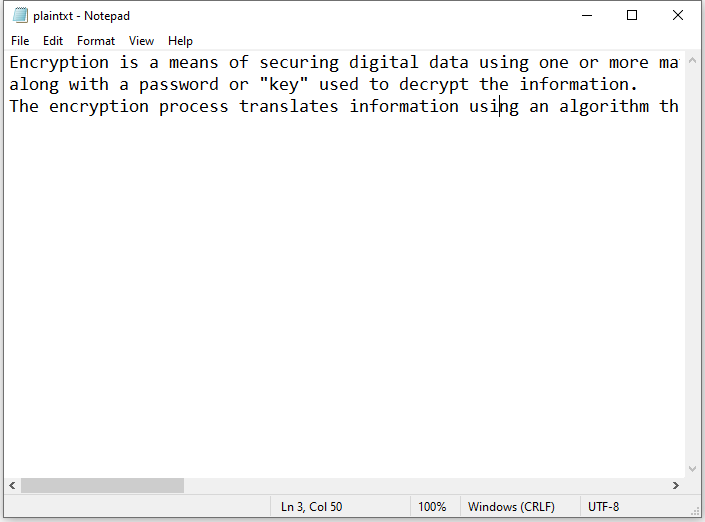
**Output:**

****

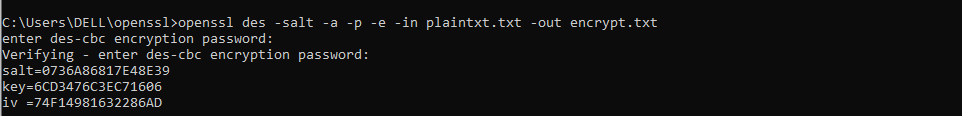
1. **Using the cipher type – des**

**B.1) -des**

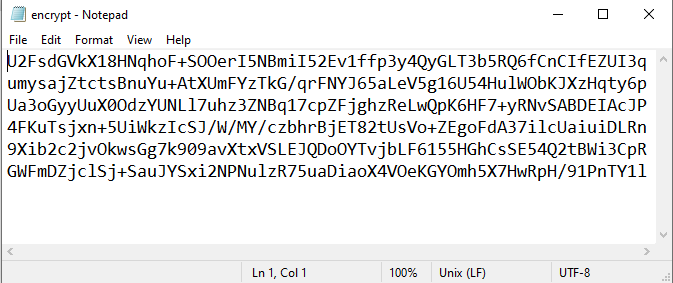
**Plaintext :**

****

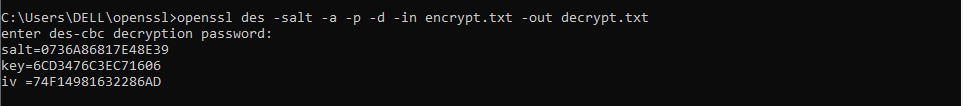
**Encryption :**

****

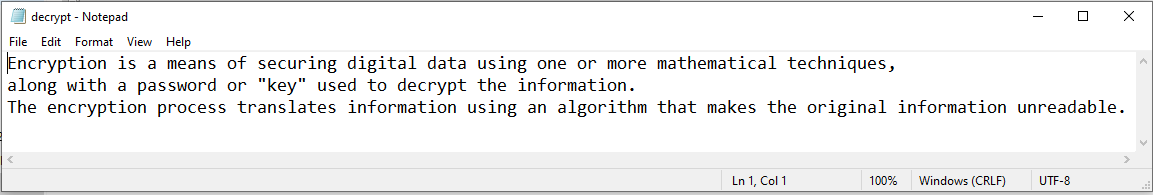
**Output :**

****

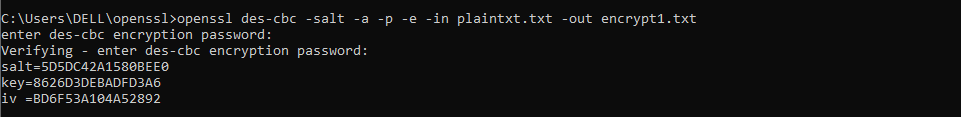
**Decryption:**

****

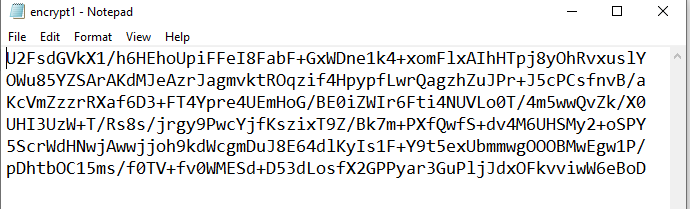
**Output:**

****

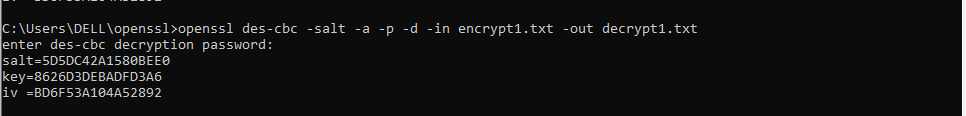
**B.2) -des-cbc**

****

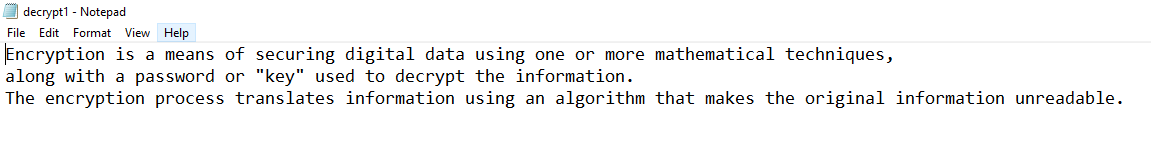
**Output:**

****

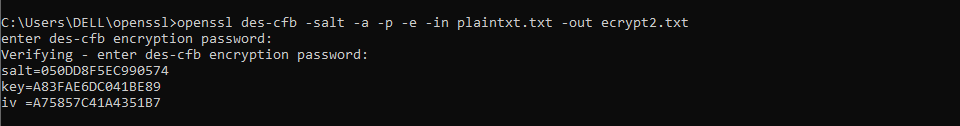
**Decryption:**

****

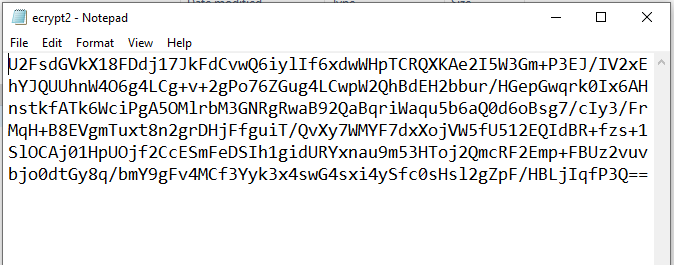
**Output:**

****

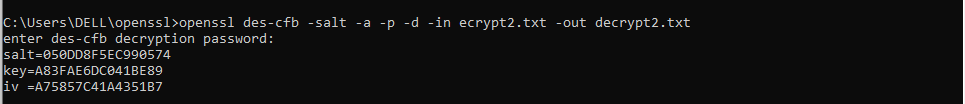
**B.3) -des-cfb**

****

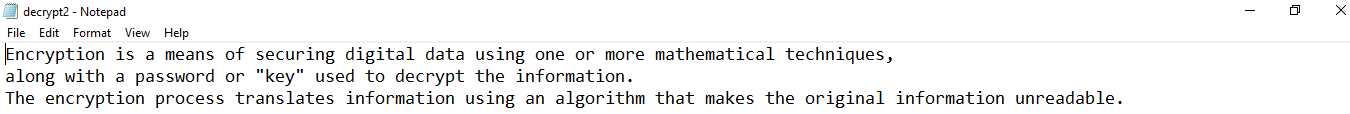
**Output:**

****

**Decryption:**

****

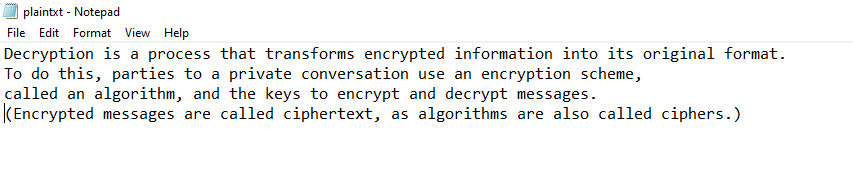
**Output:**

****

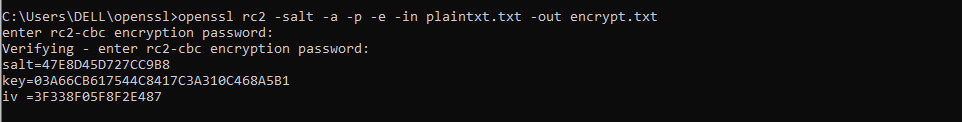
1. **Using the cipher type - rc2**

**C.1) rc2**

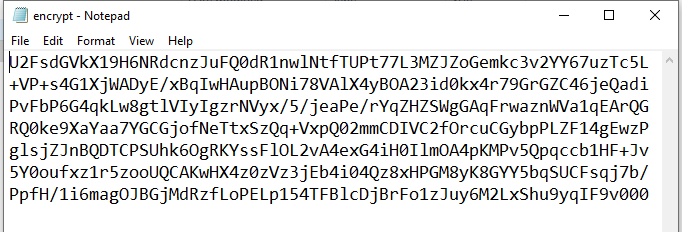
**Plaintext:**

****

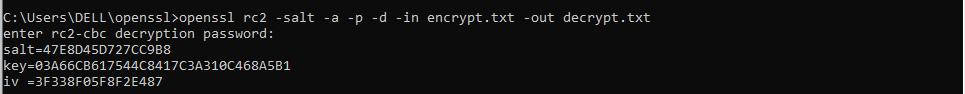
**Encryption:**

****

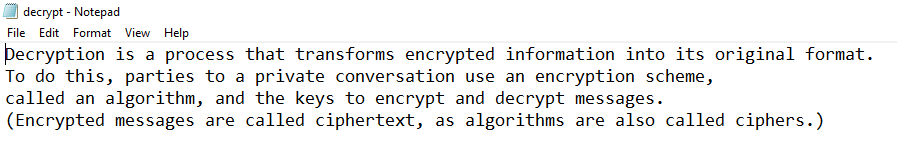
**Output:**

****

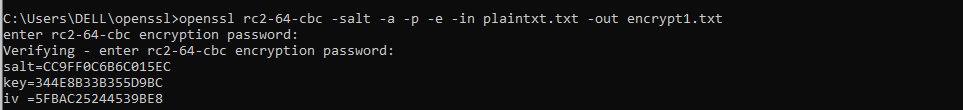
**Decryption:**

****

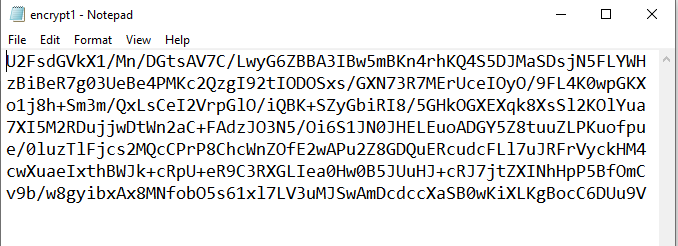
**Output:**

****

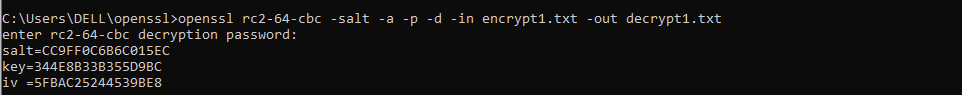
**C.2) rc2-64-cbc**

****

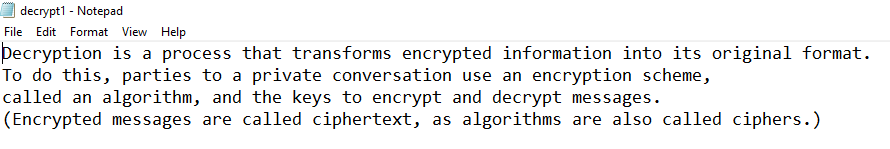
**Output:**

****

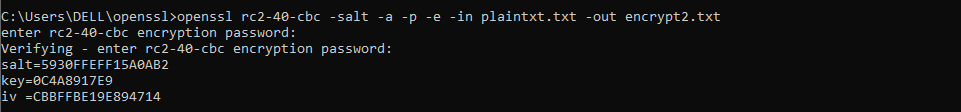
**Decryption:**

****

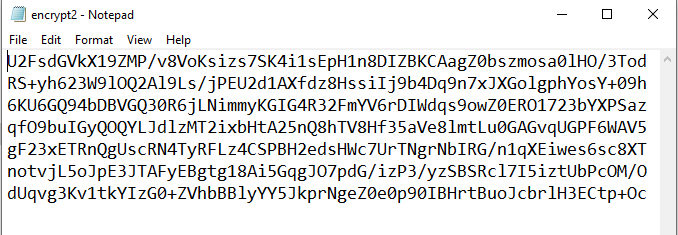
**Output:**

****

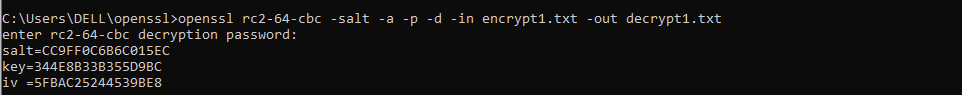
**C.3) rc2-40-cbc**

****

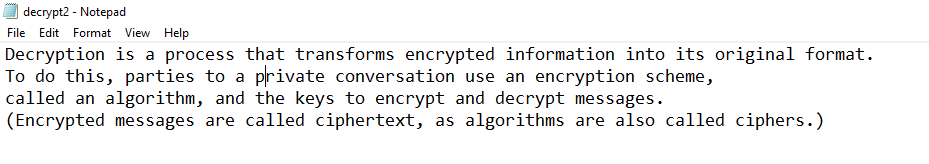
**Output:**

****

**Decryption :**

****

**Output :**

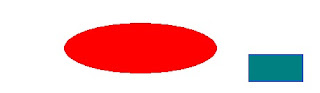
****

**Conclusion :**

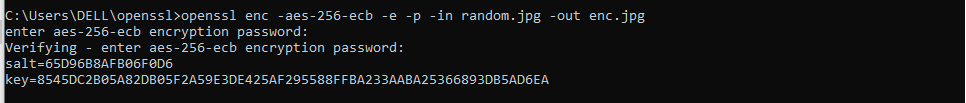
* In both aes and rc2 I observed that the key length can be changed but in DES it cannot be because it only supports a key length of 56 bits. AES supports the key lengths of 128, 192 and 256 bits. Similarly rc2 supports the key lengths of 40 and 64.
* I also observed that the encrypted text is much larger in size compared to the actual input text as encryption is providing redundancy to the plain text.
* And also using the -p parameter I was able to display the salt, key and the initial vector that were used as the input variables in the encryption.
* ECB being the most initial discovery is the most basic hence more vulnerable to attack compared to the other modes.
* AES, DES and RC2 are symmetric key algorithms using the same keys to encrypt and decrypt the data.

**TASK 2 :**

**Original image**

****

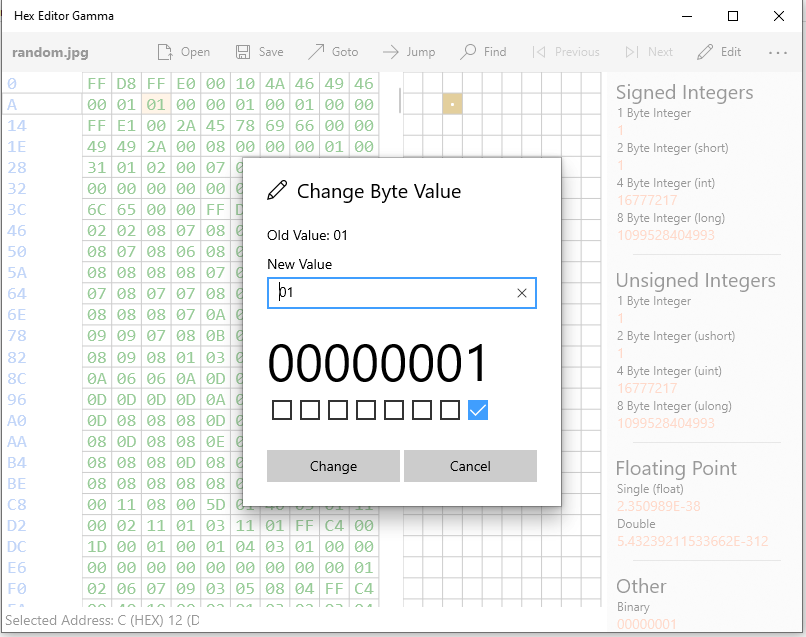
**A)Using the cipher type - aes-256-ecb**

****

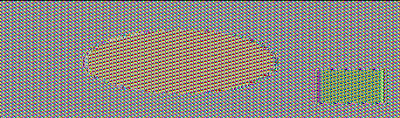
**New File Generated as name enc**

****

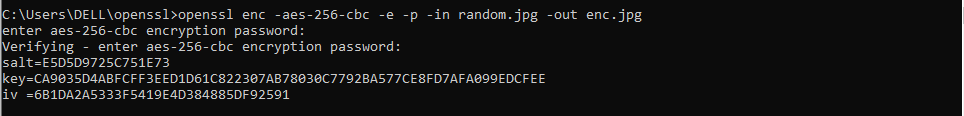
**Changing the 54 byte header of the ecb.bmp file and setting it the same as the original image :**

****

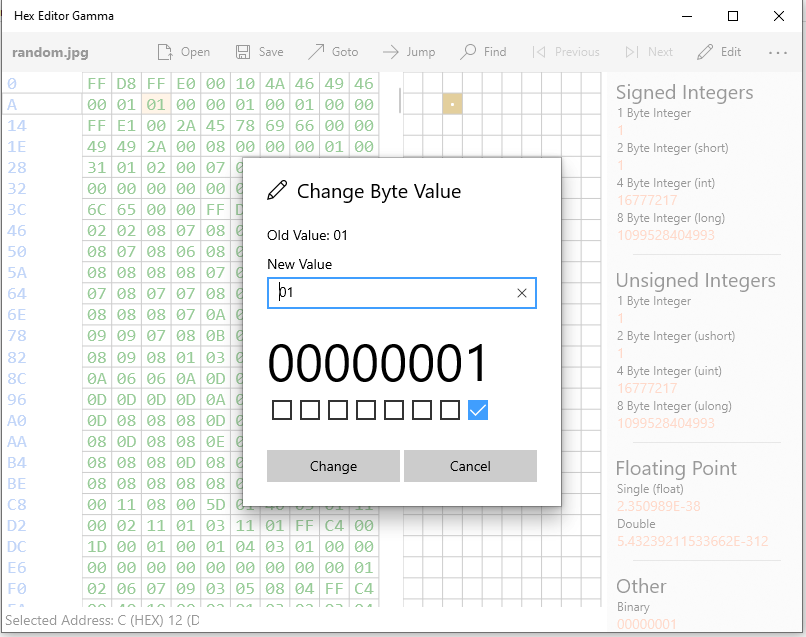
**Output:**

****

**B)Using the cipher type - aes-256-cbc**

****

**Changing the 54 byte header of the cbc.bmp file and setting it the same as the original image :**

****

**Output:**

****

**Conclusion :**

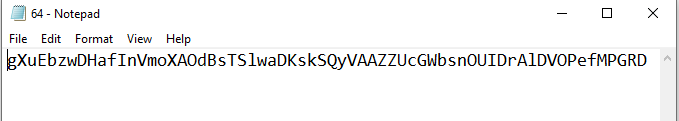
Here I observed that in the case of encrypting the file with the ECB mode the image obtained was not fully encrypted(the image boundaries are visibly clear) enabling the 3rd party to be easily able to decipher what the image could be about.

But in contrast the CBC mode encryption totally encrypted the image making it impossible to view and understand what the image was about.

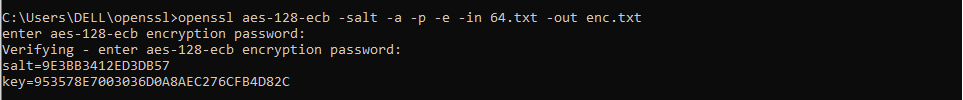
This difference can be easily justified by the fact that ECB (Electronic Codebook) is essentially the first generation of the AES. It is the most basic form of block cipher encryption whereas CBC (Cipher Block Chaining) is an advanced form of block cipher encryption. With CBC mode encryption, each ciphertext block is dependent on all plaintext blocks processed up to that point. This adds an extra level of complexity to the encrypted data hence harder to decrypt.

**Task 3:**

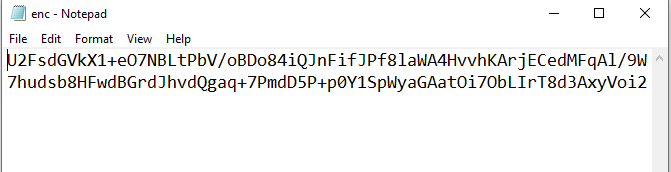
**64 Byte Text File :**

****

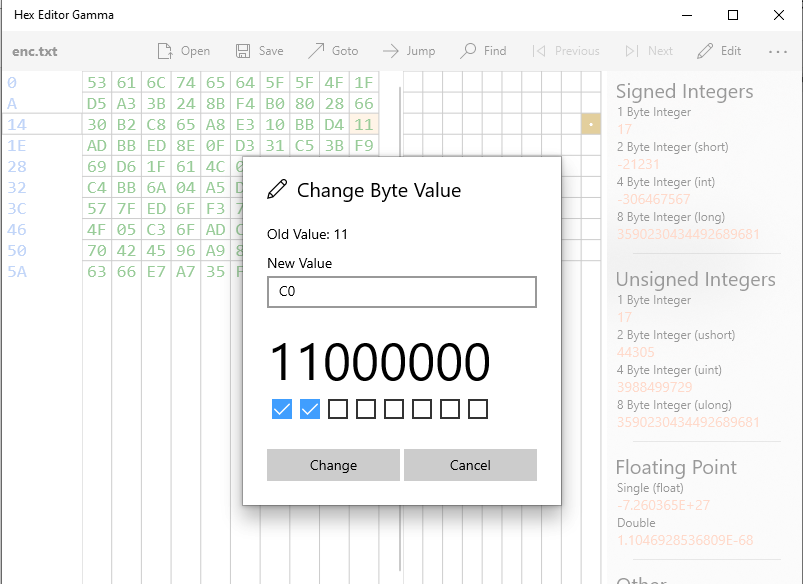
**Using the cipher type - aes-128-ecb to encrypt the text.txt**



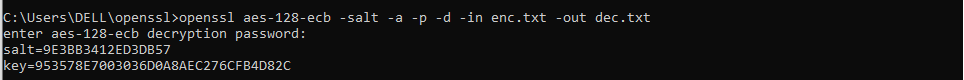
**Output:**

****

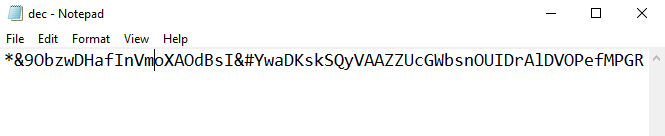
**Corrupting the 30th byte :**

****

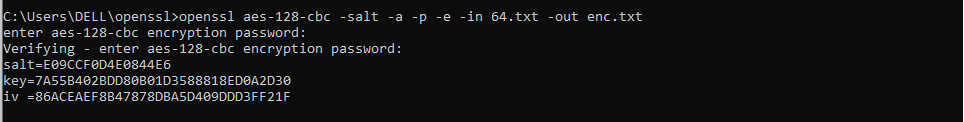
**Decryption:**

****

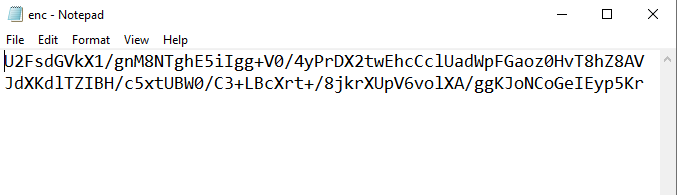
**Output:**

****

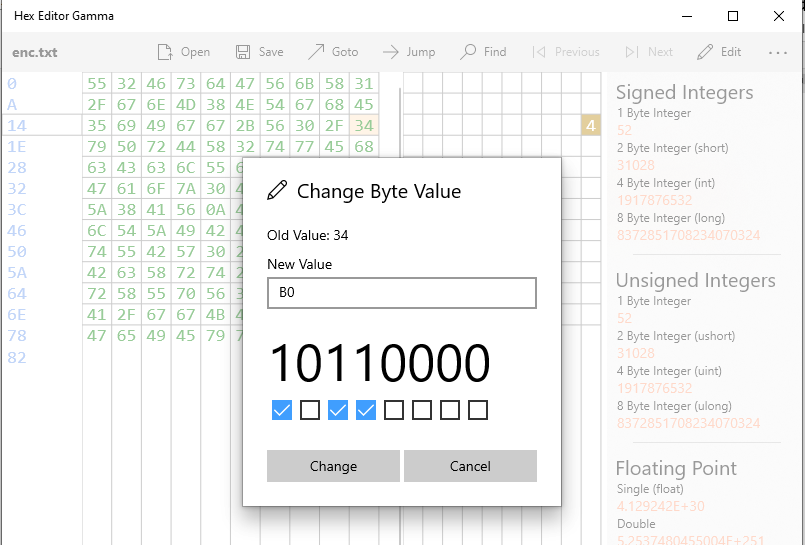
**Using the cipher type - aes-128-cbc to encrypt the text.txt**

****

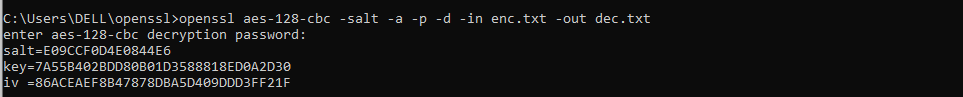
**Output:**

****

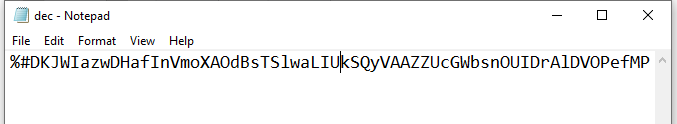
**Corrupting the 30th byte :**

****

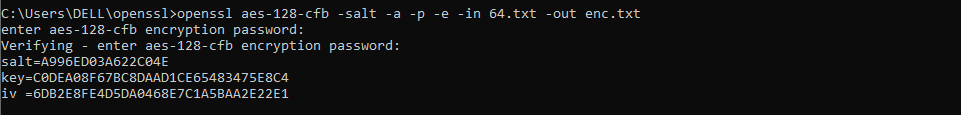
**Decryption:**

****

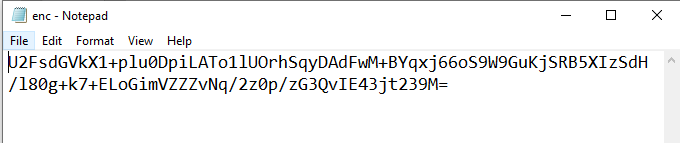
**Output:**

****

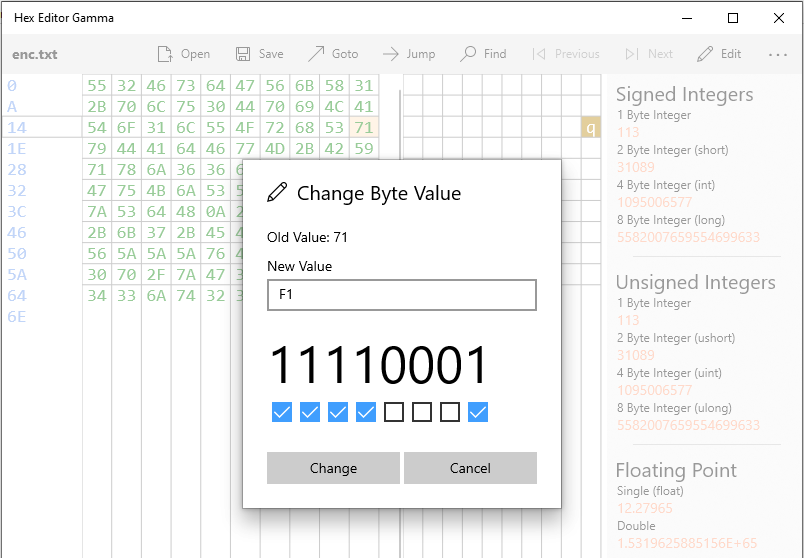
**Using the cipher type - aes-128-cfb to encrypt the text.txt**

****

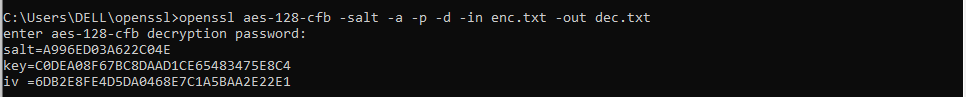
**Output:**

****

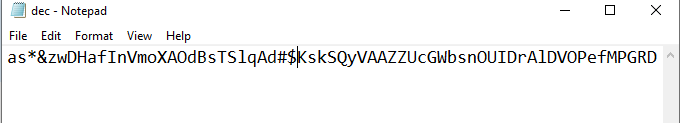
**Corrupting the 30th byte :**

****

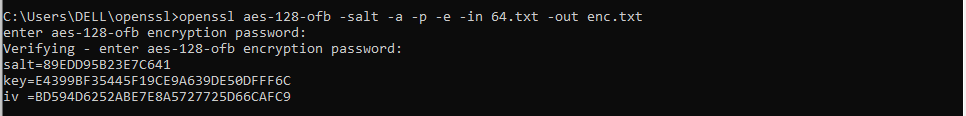
**Decryption:**

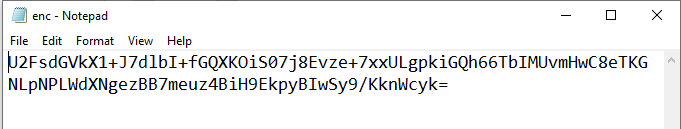
****

**Output:**

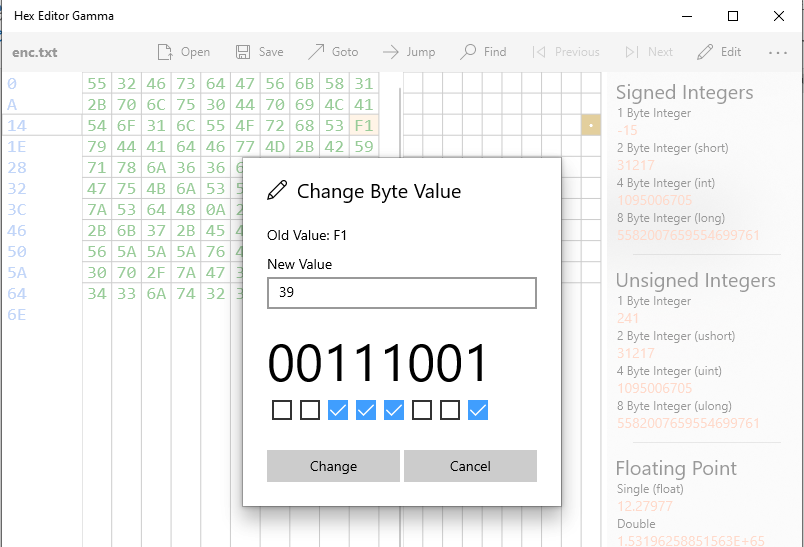
****

**Using the cipher type - aes-128-ofb to encrypt the text.txt**

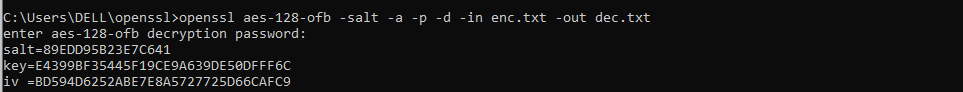
****

****

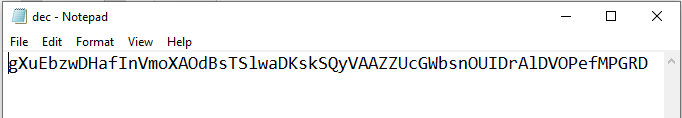
**Corrupting the 30th byte :**

****

**Decryption:**

****

**Output:**

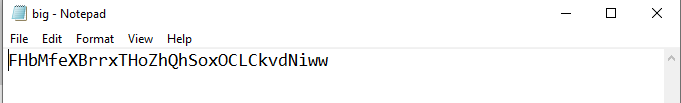
****

**Conclusion :**

* In the case of ecb mode encryption since we know that each plaintext block is encrypted separately similarly decrypted separately therefore only the block containing the corrupted byte gets corrupted there is no difference in the rest of the text.
* As we know in the cbc mode the chaining between input and output takes place , the block of plain text is XOR ed with the encrypted block of the previous pass and thus the chain continues. So I inferred and understood that if one bit of the actual plain block is corrupted then the entire chain will have corrupted bits leading to a totally corrupted text, but if say only one bit of the ciphertext is damaged only two received plaintext blocks will be damaged hence making it possible to recover the original data.
* The cfb mode is similar to the cbc mode but the only difference being that the ciphertext from the previous round needs to be encrypted and then added to the plaintext bits. Here the same encryption algorithm needs to be used for both encryption and decryption. I observed that after corrupting one ciphertext bit only the two consecutive plaintext blocks will be damaged.
* In case of ofb mode the keystream bits are created that are used for the encryption of subsequent data blocks and due to this the working of this mode is similar to a typical stream cipher. In this case I observed that if one bit of a plaintext or ciphertext message is damaged, only one corresponding ciphertext or respectively plaintext bit is damaged as well.

**Task 4:**

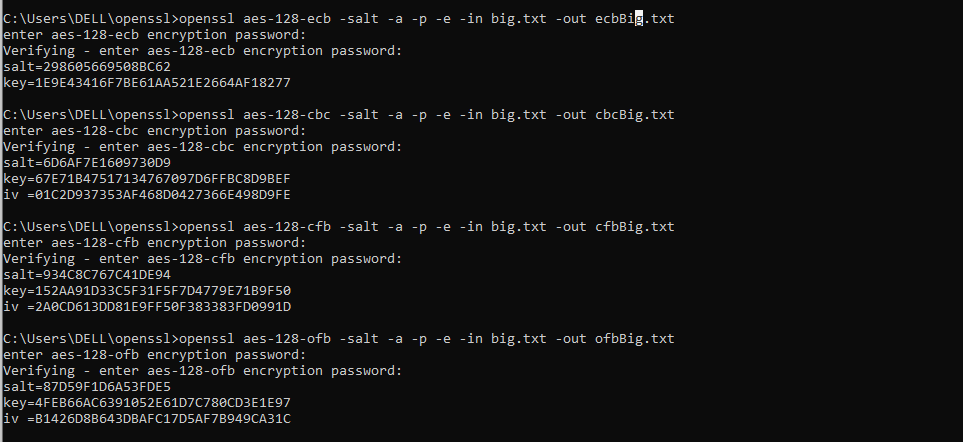
**Taking two 32 byte and 20 byte files**

****

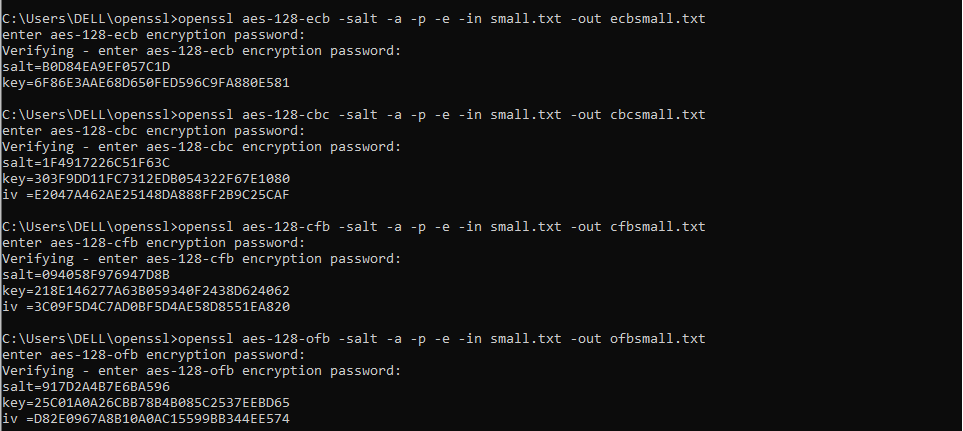
****

**Encrypting the above two text files using aes-256 in the 4 modes :**

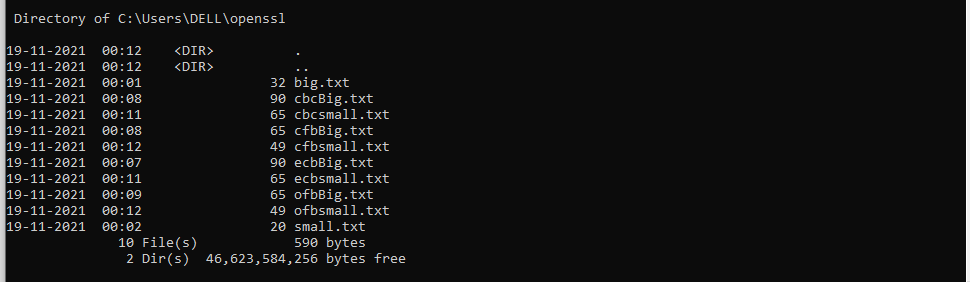
**ecb, cbc, ocb and cfb**

**Big.txt**

**Small.txt**

****

**After encrypting both the files through all the modes :**

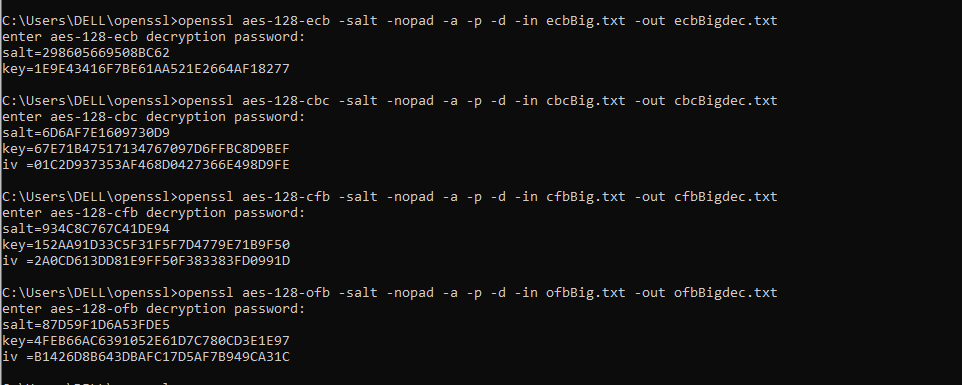
****

**Conclusion:**

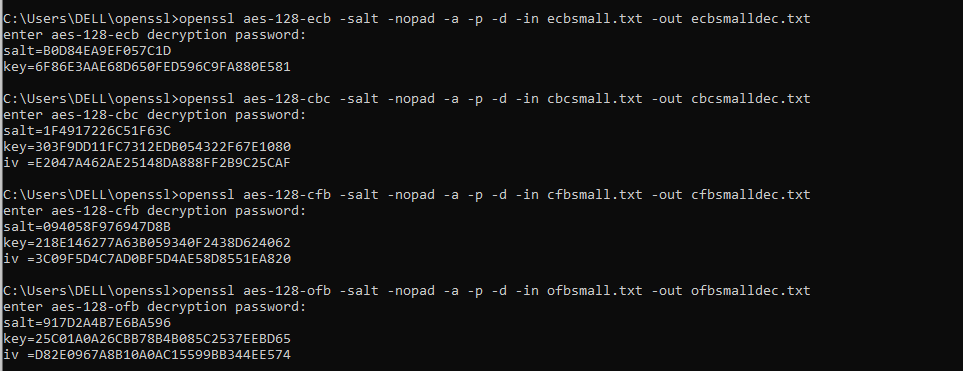
Here we observe that due to the bit size of AES being 256 bit the encrypted text obtained has a larger size for example the size has increased by 46 bytes in cbc mode for small.txt but after decryption the file becomes 20 bytes again as original.

Now we want to check the file size after decryption and check whether padding occurred or not as mentioned in the openssl manual. So we will now decrypt the file using the nopad parameter, It turns off the standard block padding. Normally, the padding is included by default during encryption, so after using the ‘no pad’ option, the padding in the decrypted file will be visible.

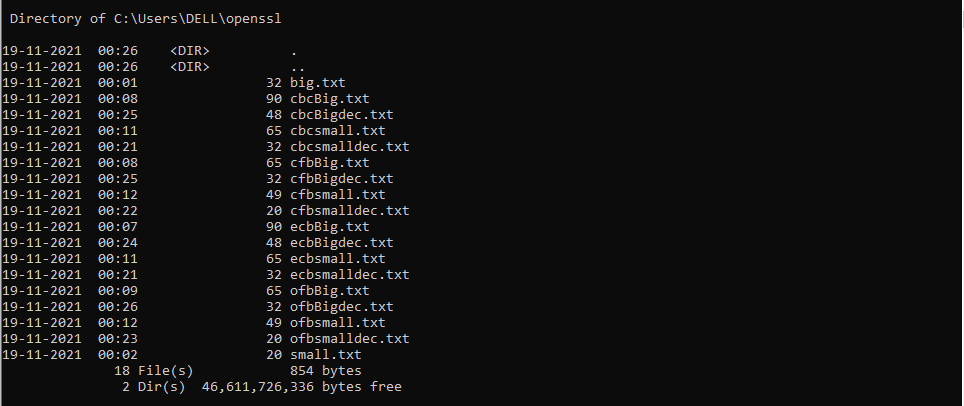
**Big.txt**

****

**Small.txt**

****

**After decrypting both the files through all the modes :**

****

**Conclusion :**

Here I observed that the ecb and cbc file size after decryption increases by 12 byte for the small.txt which was originally of 20 bytes and by 16 byte for the big.txt which was originally of 32 bytes, which means that padding was required during encryption and this can be explained by the fact that these are block ciphers and hence require that the length of the plaintext be an exact multiple of the block length so if that is not the case padding is used to fill in the blanks.

And in case of the cfb and ofb modes I observed that no padding was required since the file size after decryption did not increase and this behaviour can be explained by the fact that these are stream ciphers and since streaming modes of operation can encrypt and decrypt messages of any size, they do not require padding

**Task 5:**

**Code:**

**#task 5 code**

**from Crypto.Cipher import AES**

**from Crypto.Util.Padding import pad**

**plaintText = b"This is a top secret."**

**cipherText = "8d20e5056a8d24d0462ce74e4904c1b513e10d1df4a2ef2ad4540fae1ca0aaf9"**

**myFile = open('engwords.txt', 'r')**

**lines = myFile.readlines()**

**words = [str.strip(line) for line in lines]**

**arr = []**

**for word in words:**

**if len(word)<16:**

**word=word.lower()**

**key=word.encode()+b' '\*(16-len(word))**

**getCipher=AES.new(key, AES.MODE\_CBC, iv=bytes.fromhex('0'\*32))**

**ciphertext=getCipher.encrypt(pad(plaintText, AES.block\_size))**

**match="👎"**

**if bytes.hex(ciphertext)==cipherText:**

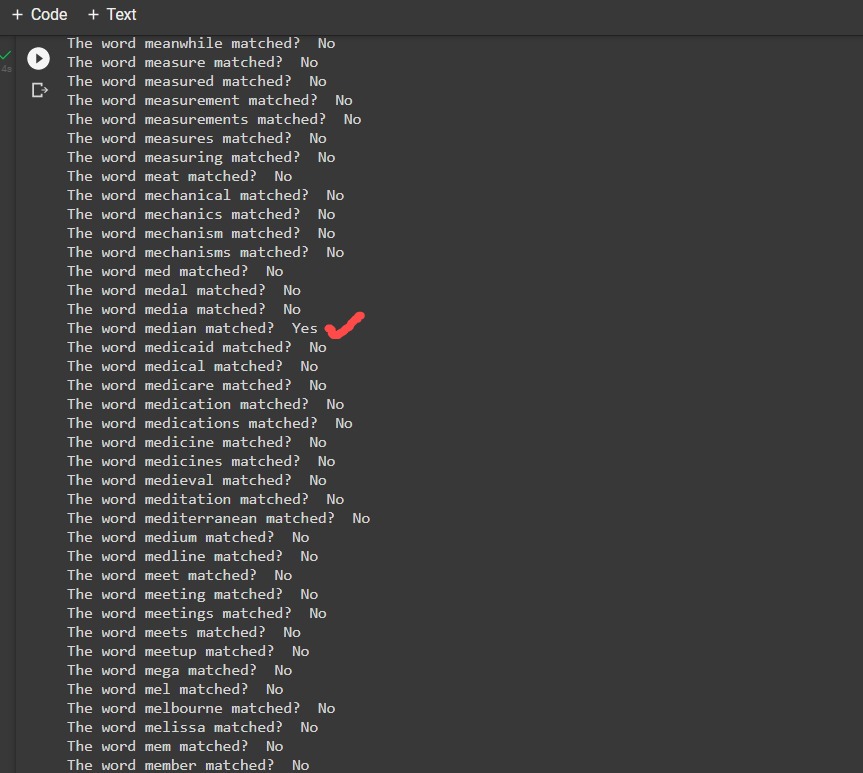
**match="👍"**

**arr.append(word)**

**print(word,match)**

**print("\n\nThe final key is :",arr)**

**Output:**

****

**CONCLUSION :**

Here I observed that using the pycryptodome library present in python and with given plain text, cipher text and the iv used I will be able to find the key by brute force approach.