<u>CSE 644 Internet Security Lab-9 (Secret Key Encryption)</u>

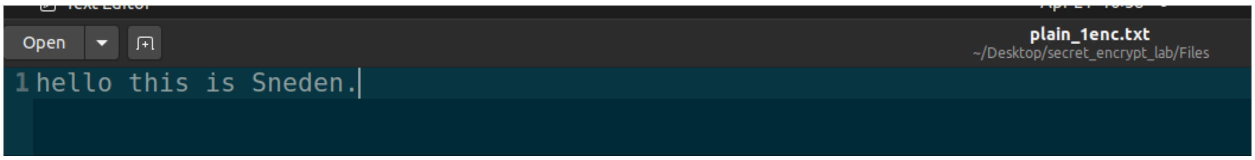
<u>Sneden Rebello</u>

<u>Task 2 :</u> Encryption using Different Ciphers and Modes

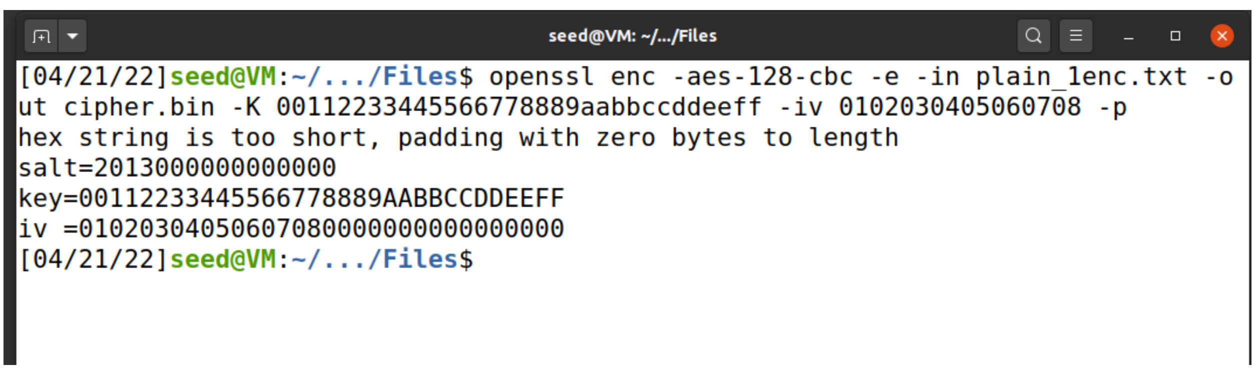
1. Using AES 128 cbc

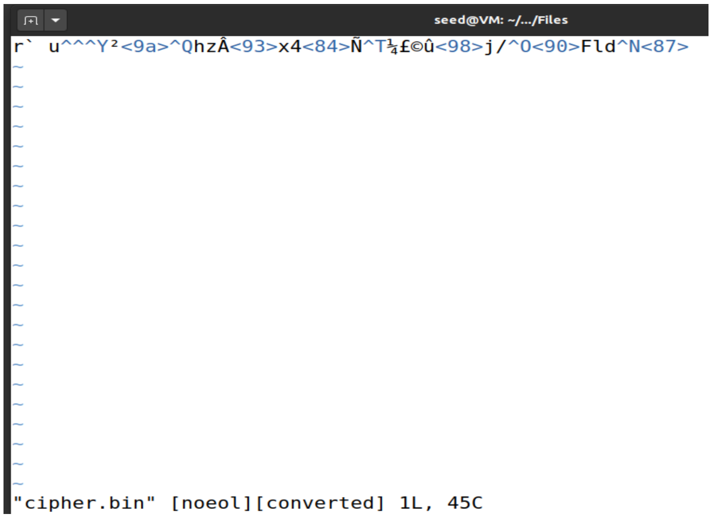
Encryption :

I created a file, ‘plain_1enc.txt’ and added the plain text as below.



Using the file ‘cipher.bin’ as the output file after encryption. The data looks like this. Using the IV and

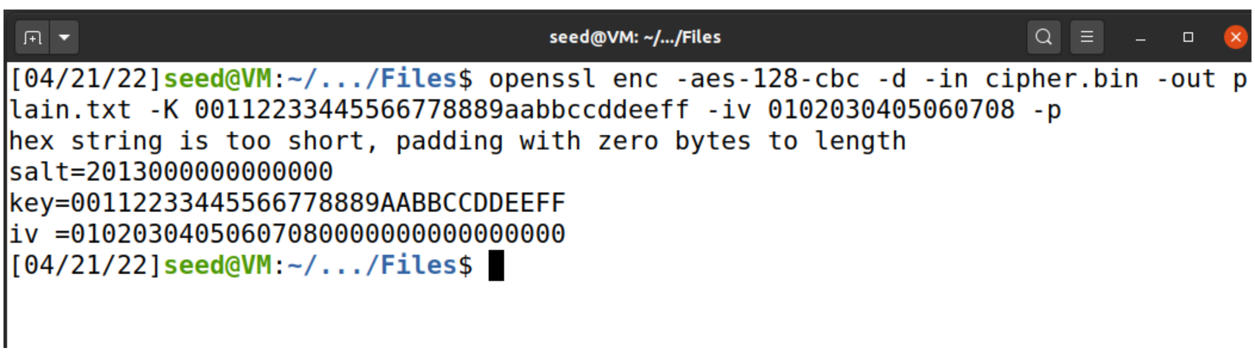


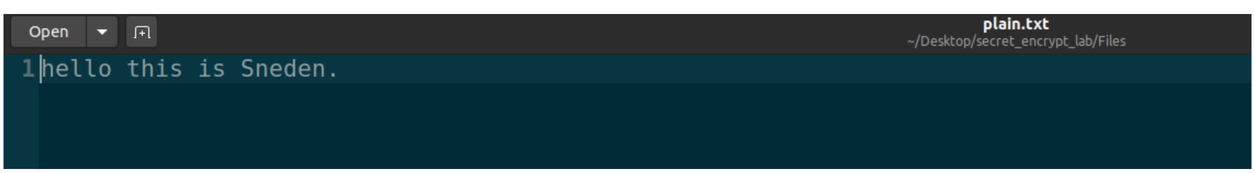


Decryption :

Use the openssl enc command with input as the ‘cipher.bin’ and output to the new ‘plain.txt’ file.

We get back the plain text entered in ‘plain_1enc’ file.





2. Using bf cbc

Encryption :

I created a file, 'plain_2enc.txt' and added the plain text as below.

```
plain_2enc.txt
~/Desktop/secret_encrypt_lab/Files
1 hello this is Sneden.
```

Using the file 'cipher.bin' as the output file after encryption. The data looks like this. Using the IV and Key.

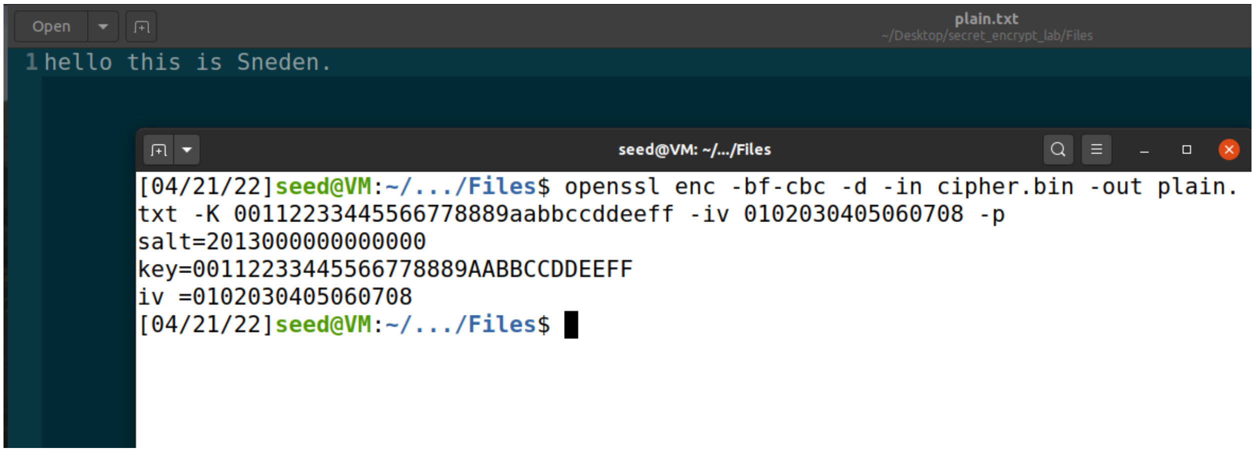
```
seed@VM: ~/.../Files
[04/21/22]seed@VM:~/.../Files$ openssl enc -bf-cbc -e -in plain_2enc.txt -out ci
pher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
salt=2013000000000000
key=00112233445566778889AABBCCDDEEFF
iv =0102030405060708
[04/21/22]seed@VM:~/.../Files$
```

```
seed@VM: ~/.../Files
  GNU nano 4.8                    cipher.bin
?������\�z��^F*�Y�U�DS�wV
```

Decryption :

Use the openssl enc command with input as the 'cipher.bin' and output to the new 'plain.txt' file.

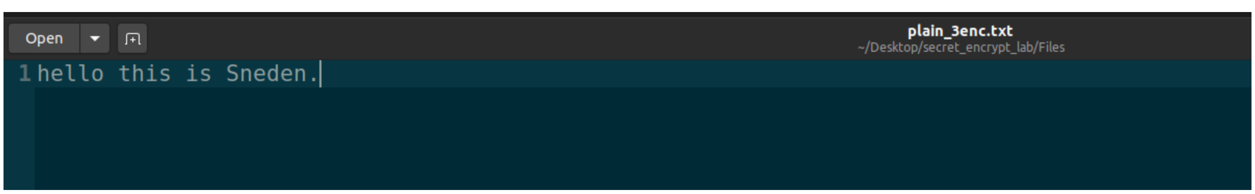
We get back the plain text entered in 'plain_2enc' file.



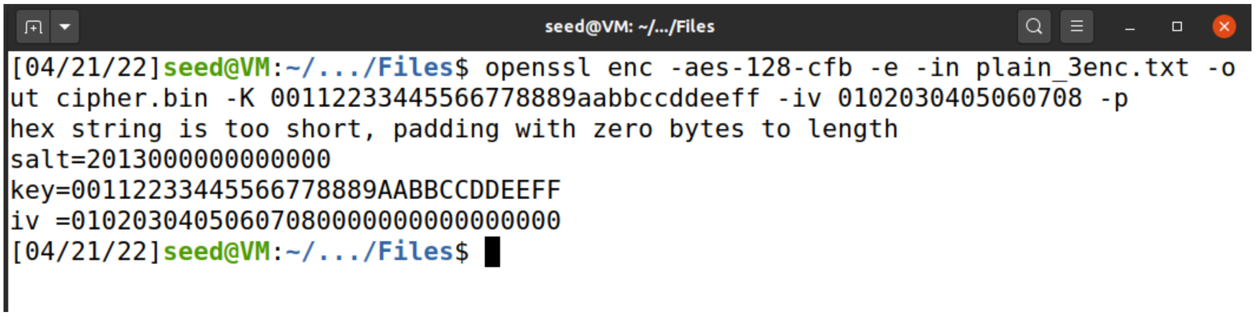
3. Using AES 128 cfb

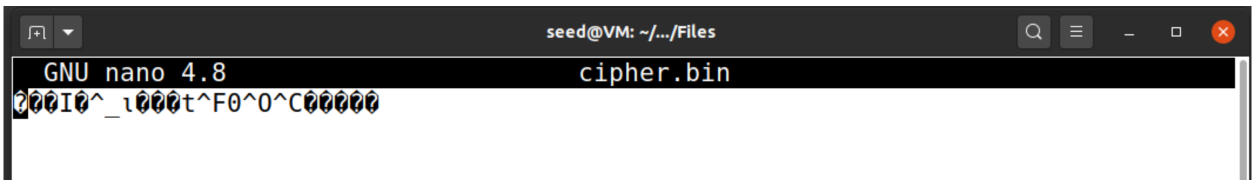
Encryption :

I created a file, ‘plain_3enc.txt’ and added the plain text as below.



Using the file ‘cipher.bin’ as the output file after encryption. The data looks like this. Using the IV and Key.

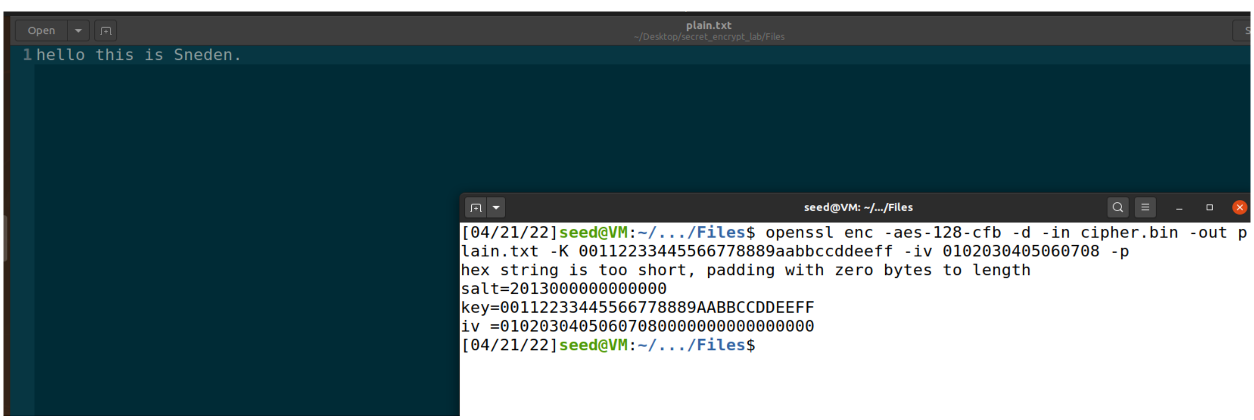




Decryption :

Use the openssl enc command with input as the ‘cipher.bin’ and output to the new ‘plain.txt’ file.

We get back the plain text entered in ‘plain_3enc’ file.

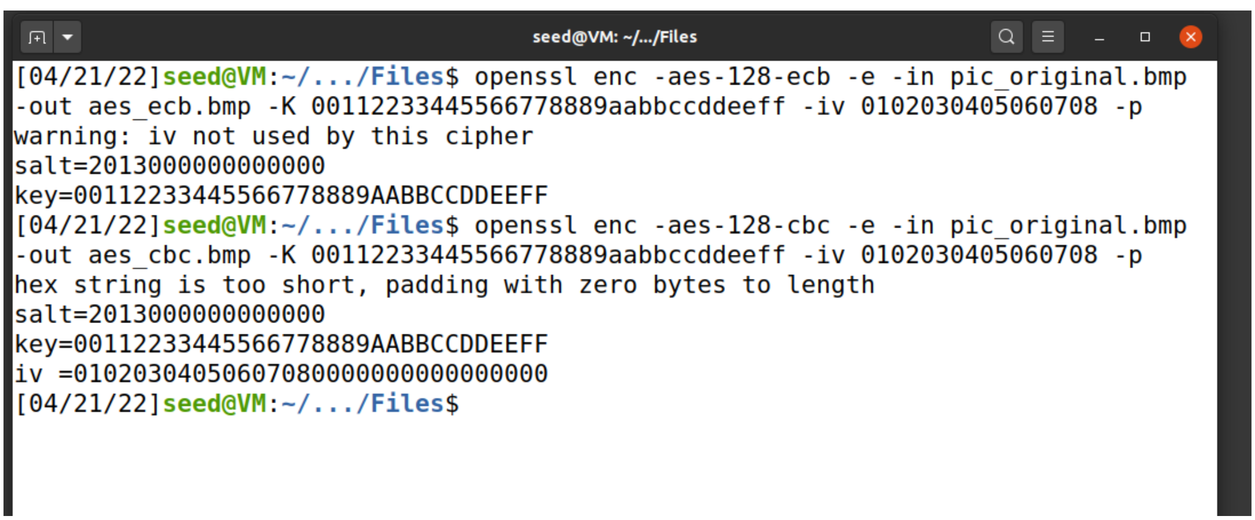


Task 3: Encryption Mode – ECB vs. CBC

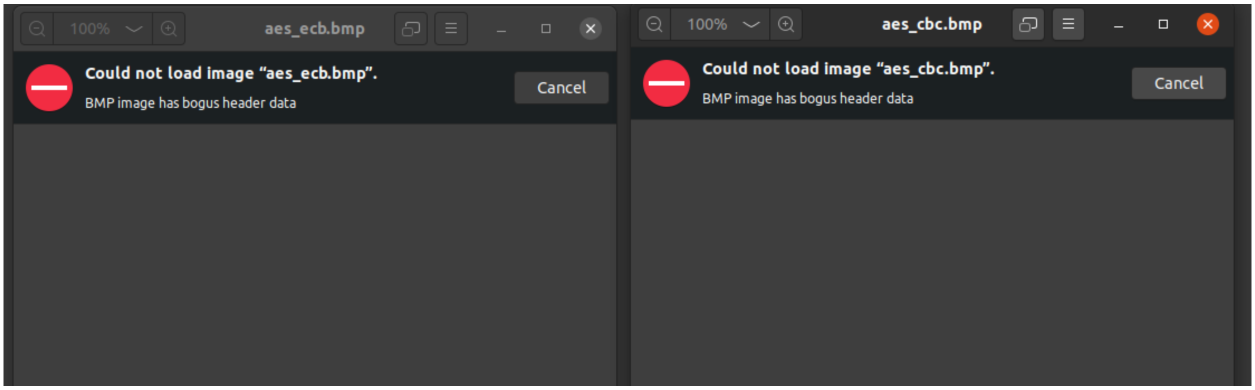
The original file ‘pic_original.bmp’ has been downloaded from the SEEDlabs webpage.

The ‘pic_original.bmp’ has been encrypted successfully using aes 128 ecb to filename ‘aes_ecb.bmp’. Similarly the ‘pic_original.bmp’ has been encrypted successfully using aes 128 cbc to filename ‘aes_cbc.bmp’.

Original Image (pic_original.bmp) :

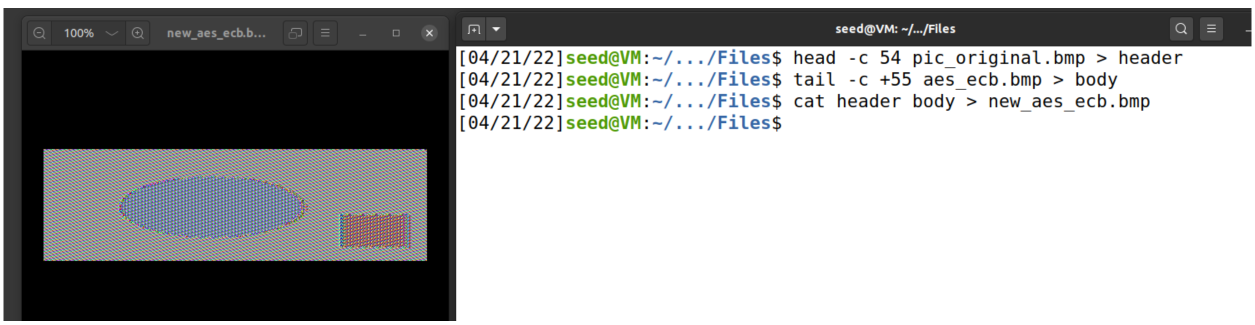


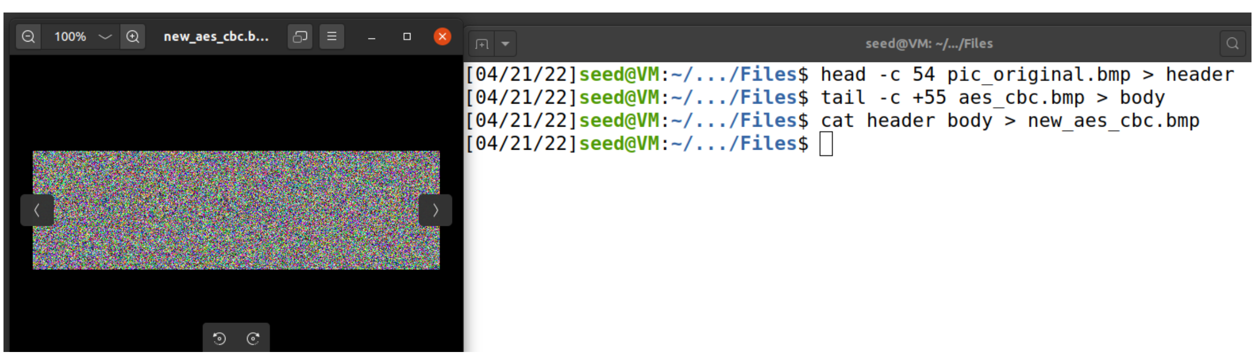
Now if we try to view the ‘.bmp’ files, we get the following error’s



Now we add the 54 byte header from 'pic_original.bmp' to the tails having the data in 'aes_ecb.bmp' and 'aes_cbc.bmp' to make them authentic '.bmp' files.

The .bmp files are now saved as, 'new_aes_ecb.bmp' and 'new_aes_cbc.bmp'





Running the same procedure for my picture ('my_pic.bmp') and encrypting it using aes 128 ecb (‘my_pic_aes_ecb.bmp’) and aes 128 cbc (‘my_pic_aes_cbc.bmp’)

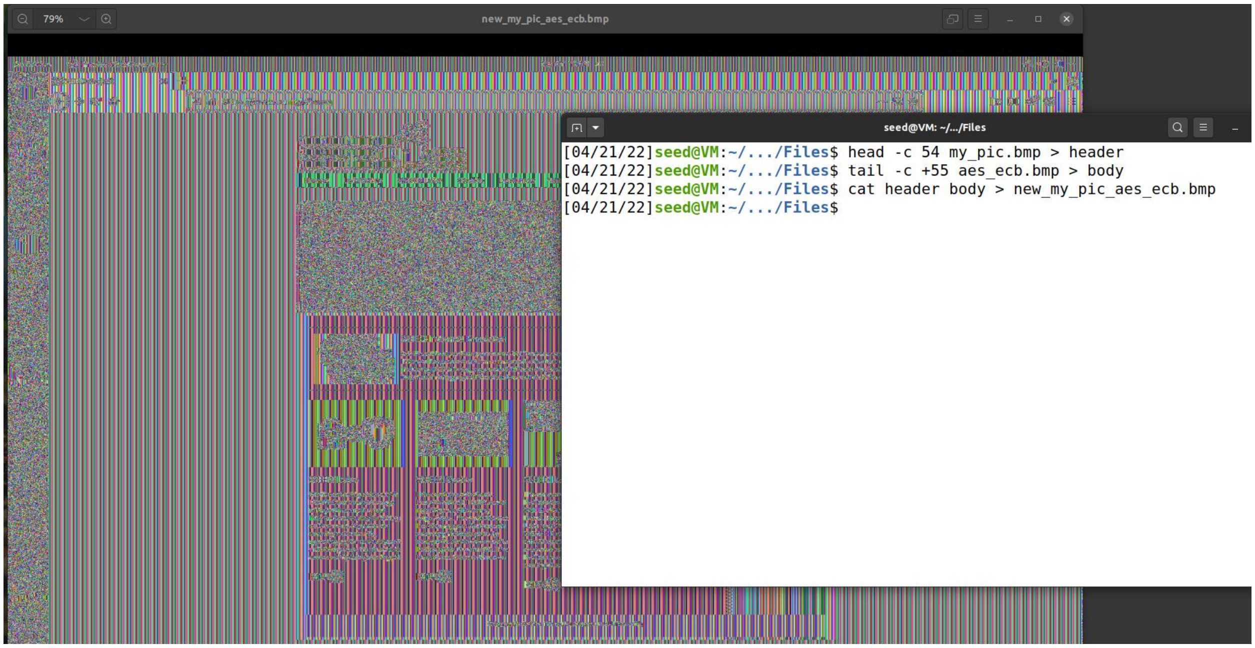
Original Picture (my_pic.bmp) :

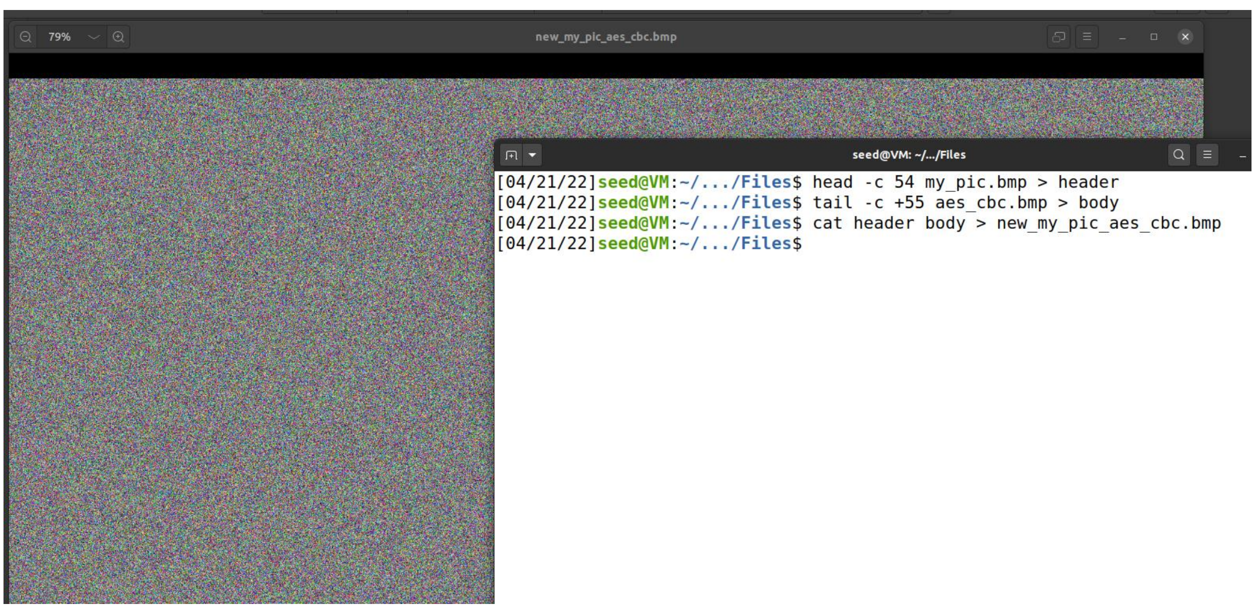

```
seed@VM: ~/.../Files
[04/21/22]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e -in my_pic.bmp -out a
es_ecb.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
warning: iv not used by this cipher
salt=2013000000000000
key=00112233445566778889AABBCCDDEEFF
[04/21/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in my_pic.bmp -out a
es_cbc.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -p
hex string is too short, padding with zero bytes to length
salt=2013000000000000
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[04/21/22]seed@VM:~/.../Files$
```

After this step, same as above the .bmp files need to be made authentic like the real one. Hence it shows an error while trying to open.

Now we add the 54 byte header from ‘my_pic.bmp’ to the tails having the data in ‘aes_ecb.bmp’ and ‘aes_cbc.bmp’ to make them authentic ‘.bmp’ files.

The .bmp files are now saved as, ‘new_my_pic_aes_ecb.bmp’ and ‘new_my_pic_aes_cbc.bmp’





2. Observations :

Comparing the size of original image to the encrypted via CBC and ECB images, we observe that the size of the encrypted image is very slightly increased from the original image size. But the height and width of the pixels have the same value for all.

Coming to the visible aspects, it's very noticeable that while using the ECB encryption over an image, the encrypted image can still contain visible information traits as the color of each individual pixel is not entirely encrypted, which is not very safe when it comes to protection of data.

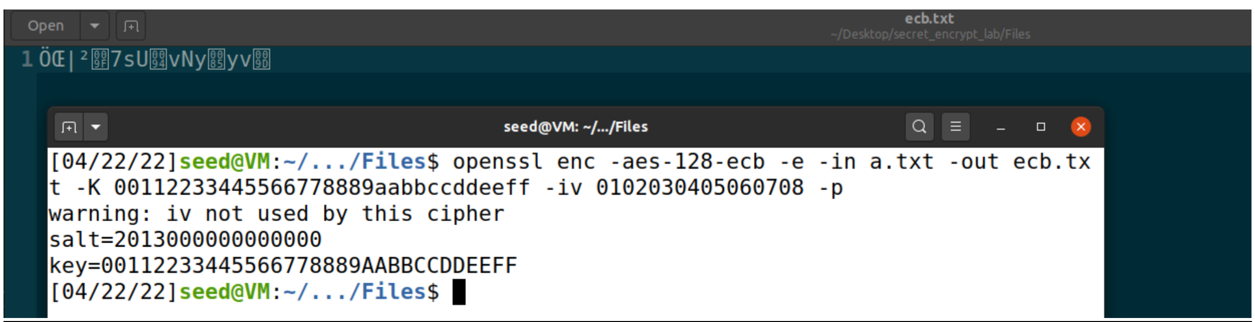
On the other hand, for the CBC encryption, there is absolutely no possible visible trait of the original image since the color of each individual pixel is encrypted hence providing a great encryption thereby increase the data protection.

In conclusion the CBC (Cipher Block Chaining) mode is more secure and provides a better data hiding capability than the ECB (Electronic Code Book) mode.

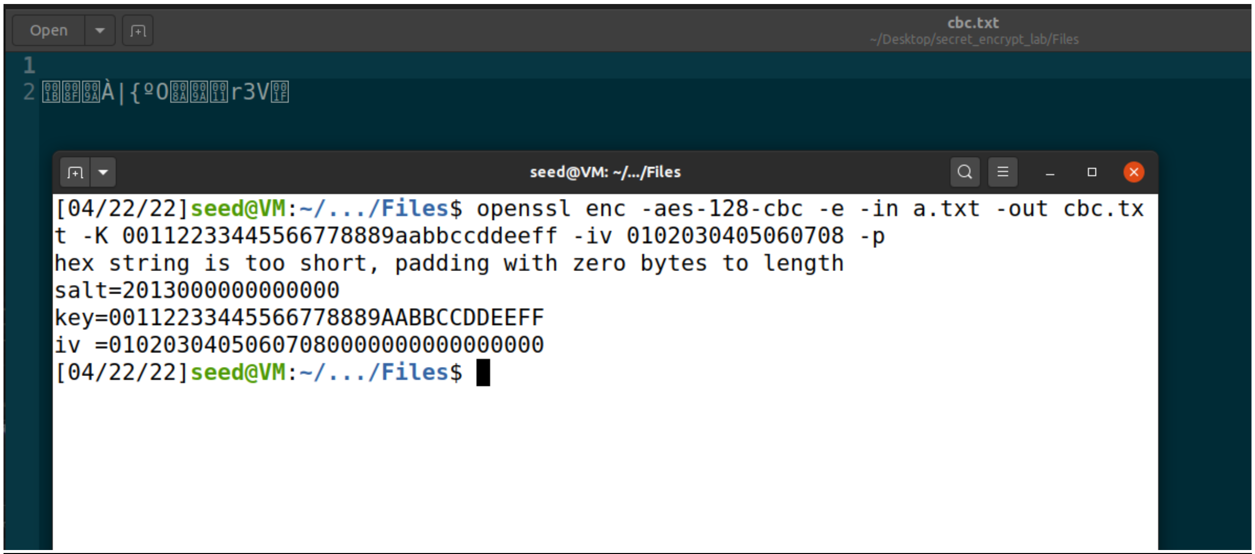
Task 4: Padding

1. The file ‘a.txt’ has been created, encrypted and stored in ‘ecb.txt’ using the ECB mode, ‘cbc.txt’ using the CBC mode, ‘ofb.txt’ using the OFB mode and ‘cfb.txt’ using the CFB mode.

ECB mode :



CBC mode :



OFB mode :

CFB mode :

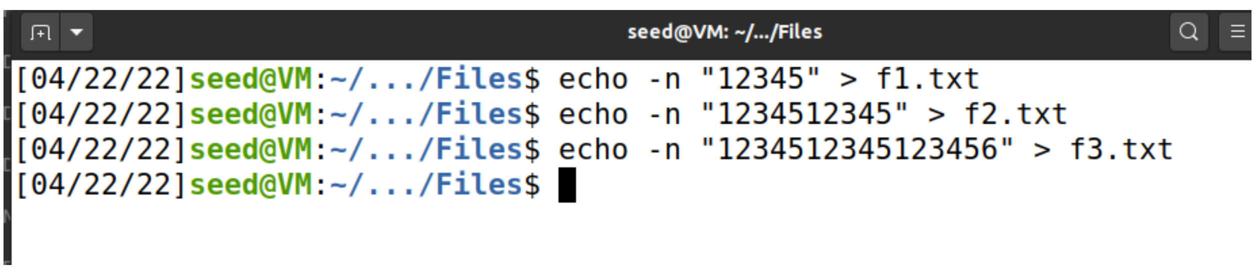
The ECB and CBC mode have paddings. The OFB and CFB modes do not have paddings.

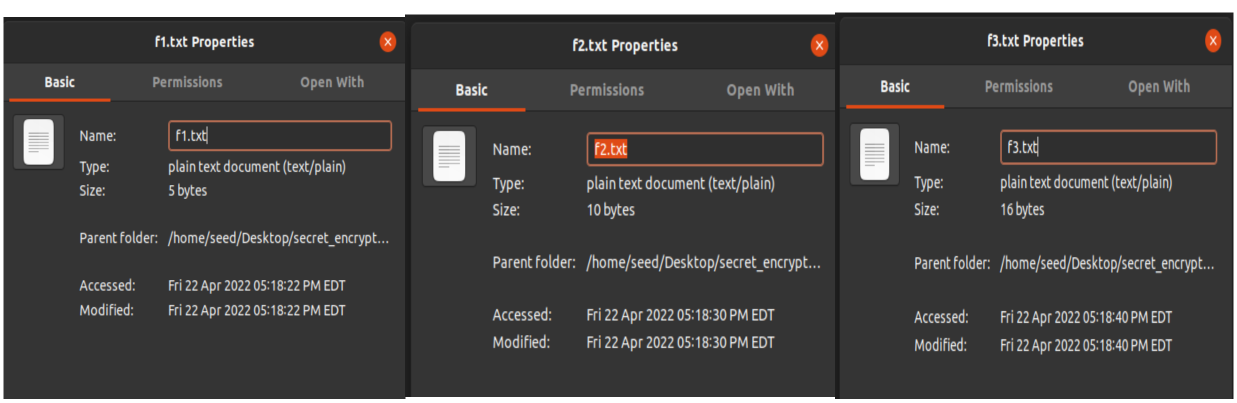
REASON :

If we encrypt a plaintext using Cipher-Feedback (CFB) mode then the encryption and decryption functions are the same, and does not require the plaintext to be padded that means, the ciphertext and the plaintext are of the same length.

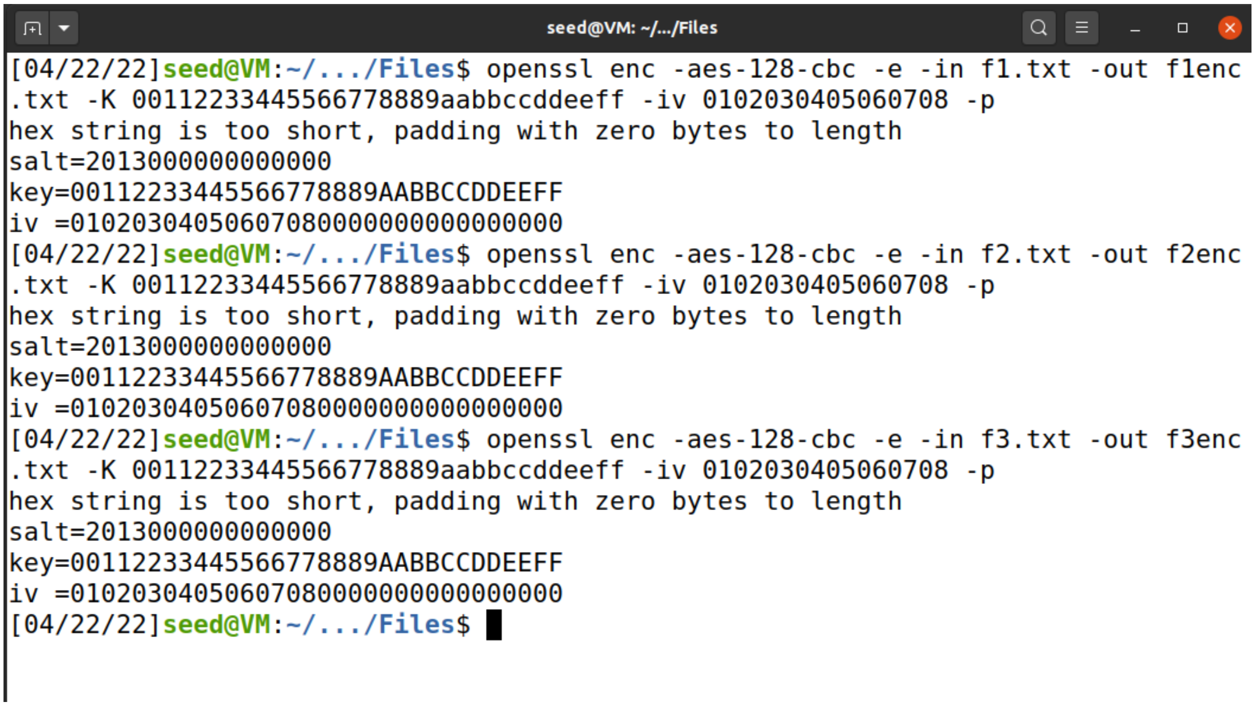
For Output-Feedback (OFB), similar like with CFB, the encryption and decryption methods are the same, and no padding is required.

2. Created 3 files ‘f1.txt’, ‘f2.txt’, ‘f3.txt’ of 5 bytes, 10 bytes and 16 bytes respectively via the echo -n “ ” > filename.txt command.

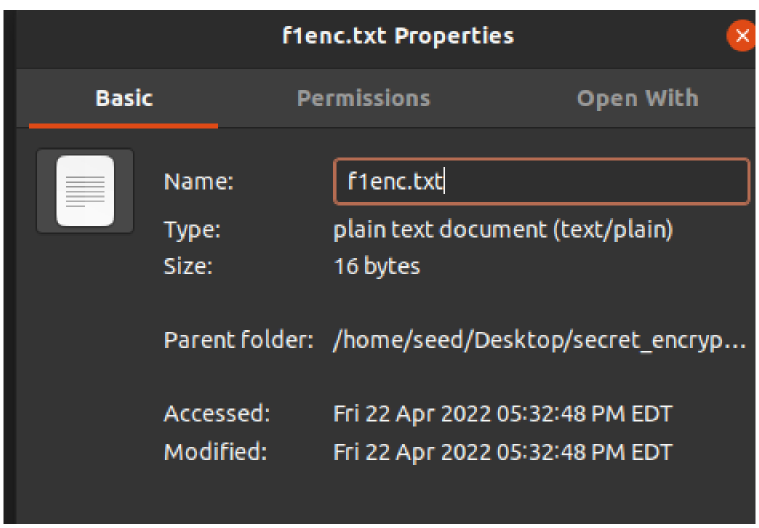




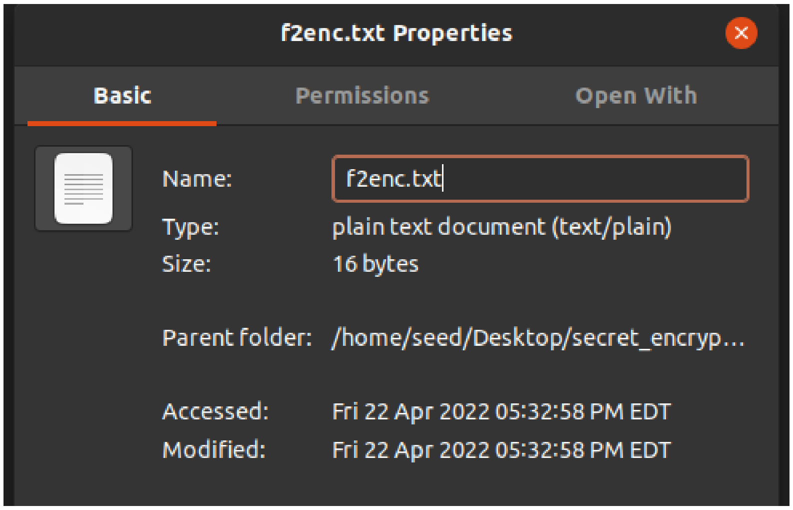
Used “openssl enc -aes-128-cbc -e" to encrypt these three files using 128-bit AES with CBC mode



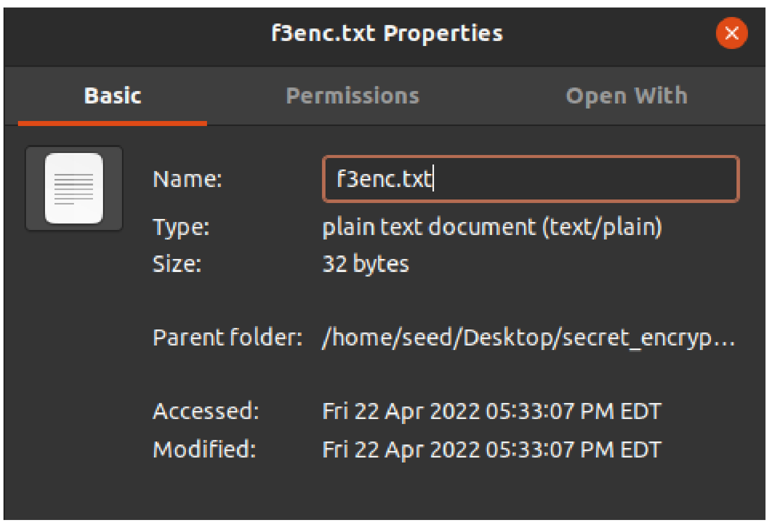
F1 encrypted (f1enc.txt) :



F2 encrypted (f2enc.txt) :



F3 encrypted (f3enc.txt) :



Conclusion on size :

As shown above, the size of 2 files ‘f1enc.txt’, ‘f2enc.txt’ are 16 bytes with unencrypted sizes as 5 bytes and 10 bytes respectively. Whereas the size of ‘f3enc.txt’ is 32 bytes, double the size of the unencrypted size (16 bytes). This is due to the padding which gets added to keep the plaintext as a multiple with the block length.

On decryption to find the padding,

Padding data for f1.txt :

```
seed@VM: ~/.../Files
[04/22/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -d -in f1enc.txt -out f1
dec.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -nopad -p
hex string is too short, padding with zero bytes to length
salt=2013000000000000
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[04/22/22]seed@VM:~/.../Files$ xxd f1dec.txt
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b  12345...........
[04/22/22]seed@VM:~/.../Files$
```

Padding data for f2.txt :

```
seed@VM: ~/.../Files
[04/22/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -d -in f2enc.txt -out f2
dec.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -nopad -p
hex string is too short, padding with zero bytes to length
salt=2013000000000000
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[04/22/22]seed@VM:~/.../Files$ xxd f2dec.txt
00000000: 3132 3334 3531 3233 3435 0606 0606 0606  1234512345......
[04/22/22]seed@VM:~/.../Files$
```

Padding data for f3.txt :

```
seed@VM: ~/.../Files
[04/22/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -d -in f3enc.txt -out f3
dec.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708 -nopad -p
hex string is too short, padding with zero bytes to length
salt=2013000000000000
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607080000000000000000
[04/22/22]seed@VM:~/.../Files$ xxd f3dec.txt
00000000: 3132 3334 3531 3233 3435 3132 3334 3536  1234512345123456
00000010: 1010 1010 1010 1010 1010 1010 1010 1010  ................
[04/22/22]seed@VM:~/.../Files$
```

Hence for the 16 byte file (block length = 16 bytes) using PKCS#5 padding, adds between 1 and n bytes, where n is the bytes of the cipher block length, In the ‘f3.txt’ case, 16 extra bytes added.

Task 6: Initial Vector (IV) and Common Mistakes

6.1. IV Experiment

The file ‘a.txt’ has been created

```
seed@VM: ~/.../Files
  GNU nano 4.8                              a.txt
plaintext
```

When same IV is used :

```
[04/22/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in a.txt -out sameiv1.txt -K 00112233445566778889aabbccddeeff
 -iv 01020304050607081234567891234567 -p
salt=2013000000000000
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607081234567891234567
[04/22/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in a.txt -out sameiv2.txt -K 00112233445566778889aabbccddeeff
 -iv 01020304050607081234567891234567 -p
salt=2013000000000000
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607081234567891234567
[04/22/22]seed@VM:~/.../Files$ diff sameiv1.txt sameiv2.txt
[04/22/22]seed@VM:~/.../Files$
```

I notice that when the same encryption with same IV gives and identical output.

When different IV is used :

```
seed@VM: ~/.../Files
[04/22/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in a.txt -out difiv1.txt -K 00112233445566778889aabbccddeeff
-iv 01020304050607081234567891234567 -p
salt=2013000000000000
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607081234567891234567
[04/22/22]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in a.txt -out difiv2.txt -K 00112233445566778889aabbccddeeff
-iv 01020304050607081234567897777777 -p
salt=2013000000000000
key=00112233445566778889AABBCCDDEEFF
iv =01020304050607081234567897777777
[04/22/22]seed@VM:~/.../Files$ diff difiv1.txt difiv2.txt
1c1
< �:�L"����2: ��
\ No newline at end of file
---
> �����y�5������
\ No newline at end of file
[04/22/22]seed@VM:~/.../Files$
```

I notice that when the same encryption on plaintext with different IV give different outputs.

The key point is that if you ever reuse an IV, you open yourself up to cryptographic attacks that are easier to execute than those when you use a different IV every time. So, for every sequence where you need to start encrypting again, you need a new, unique IV. Also, if the plaintext is repeating, using the same IV will result in information leakage and display some visible patterns that are readable.

6.2. Common Mistake: Use the Same IV

For OFB mode, If the key and IV are unchanged, knowing plaintext is easy.

Output stream can be obtained by XORing plaintext and ciphertext block by block. When sharing the same key and IV for OFB mode, the output streams are identical among encryptions.

In this case scenario, we know plaintext p1 and its OFB ciphertext c1, and another OFB ciphertext c2 with the same key and IV. But we do not know the plaintext p2 of c2.

Now to figure it:

First, get the output stream from the encryption of the first plaintext p1:

output_stream = p1 XOR c1

Then get p2 by:

p2 = output_stream XOR c2

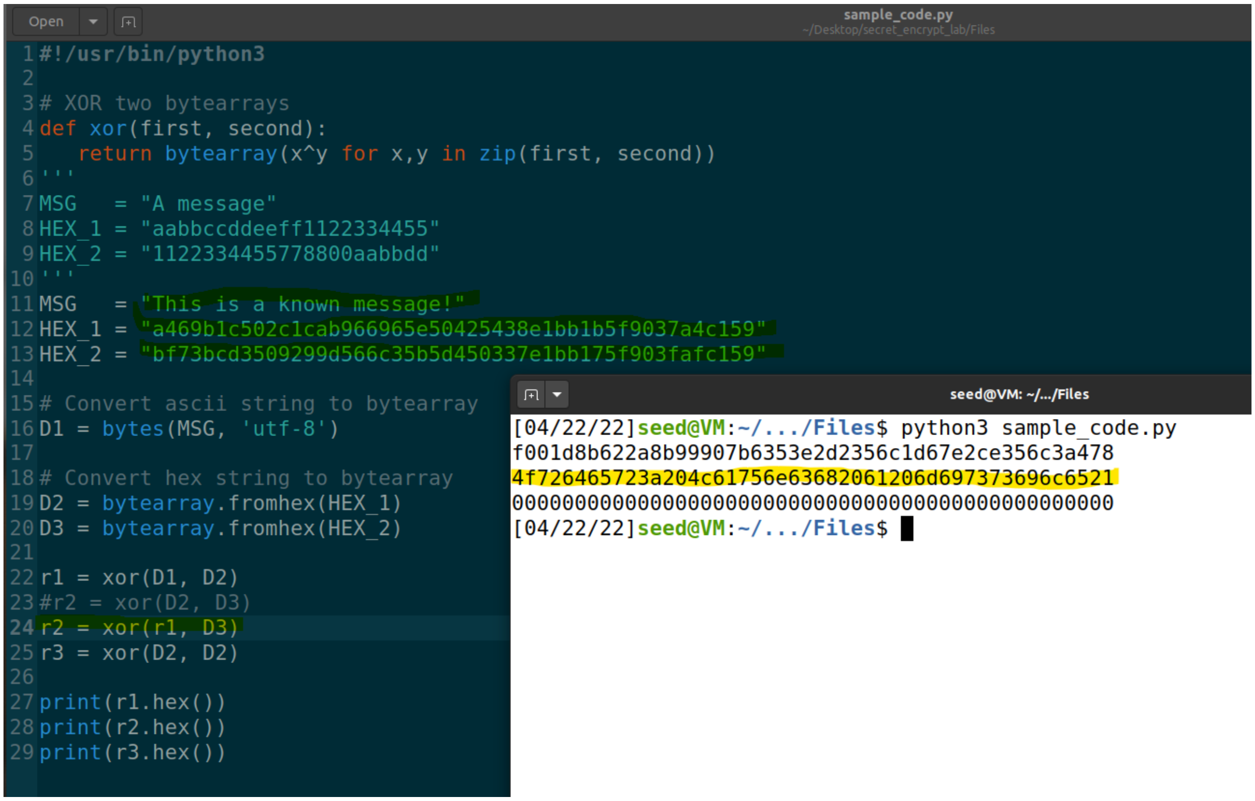
The example scenario is like :

Plaintext (P1): This is a known message!

Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

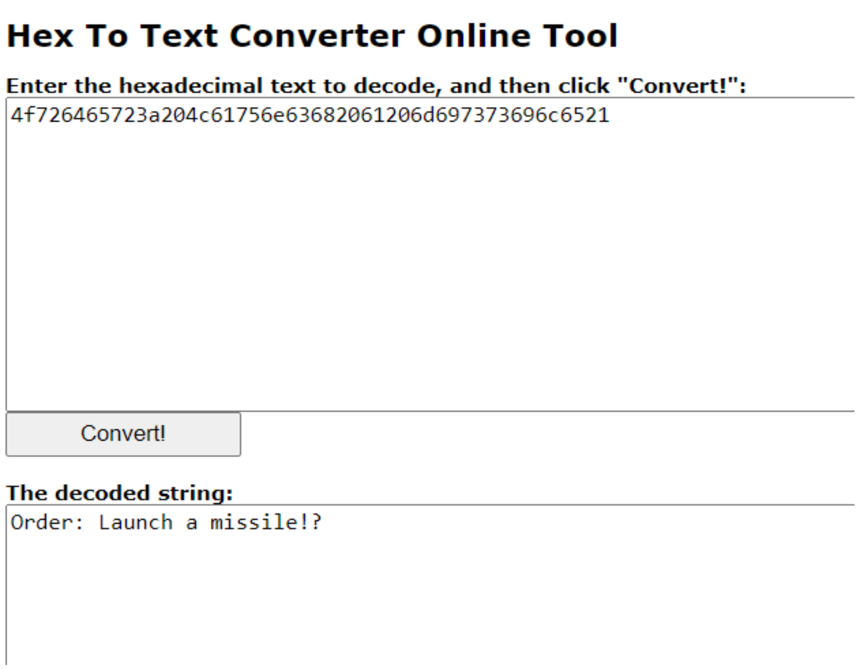
Plaintext (P2): (unknown to me)

Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159



Now if I take r2 which is in hex and convert it to string, I get :

P2 as "Order: Launch a missile!" as shown below.



For CFB mode, as its demonstration, it is the same situation for the initial block (i.e. can get plaintext by simple XOR). However, if the key remains secret, the following parts of ciphertext will not be revealed. Hence when using the CFB mode, the attacker can only know the first block of the message.

6.3. Common Mistake: Use a Predictable IV

I guess P1 is "Yes".

Now I construct,

P2 = "Yes" XOR IV XOR IV_NEXT

Where IV is the IV used to generate C1 and IV_NEXT is the predictable IV used to encrypt the next plaintext input.

In this case:

Encryption method: 128-bit AES with CBC mode.

Ciphertext C1 (known to both)

IV used on P1 (known to both)

Next IV (known to both)

Plaintext P1 (Unknown)

Open | sample_code.py ~/Desktop/secret_encrypt_lab/Files

```python
#!/usr/bin/python3

# XOR two bytearrays
def xor(first, second):
   return bytearray(x^y for x,y in zip(first, second))
yes = b"Yes" + bytes("\x0d"*13, 'utf-8')
no = b"No" + bytes("\x0d"*14, 'utf-8')

#MSG   = "Yes"
HEX_1 = "d5b025e8a45d18b9a566e45b857e8d06"
HEX_2 = "1426665ea55d18b9a566e45b857e8d06"

# Convert ascii string to bytearray
#D1 = bytes(MSG, 'utf-8')
D1 = yes
# Convert hex string to bytearray
D2 = bytearray.fromhex(HEX_1)
D3 = bytearray.fromhex(HEX_2)

r1 = xor(D1, D2)
#r2 = xor(D2, D3)
r2 = xor(r1, D3)
#r3 = xor(D2, D2)

#print(r1.hex())
print(r2.hex())
#print(r3.hex())
```

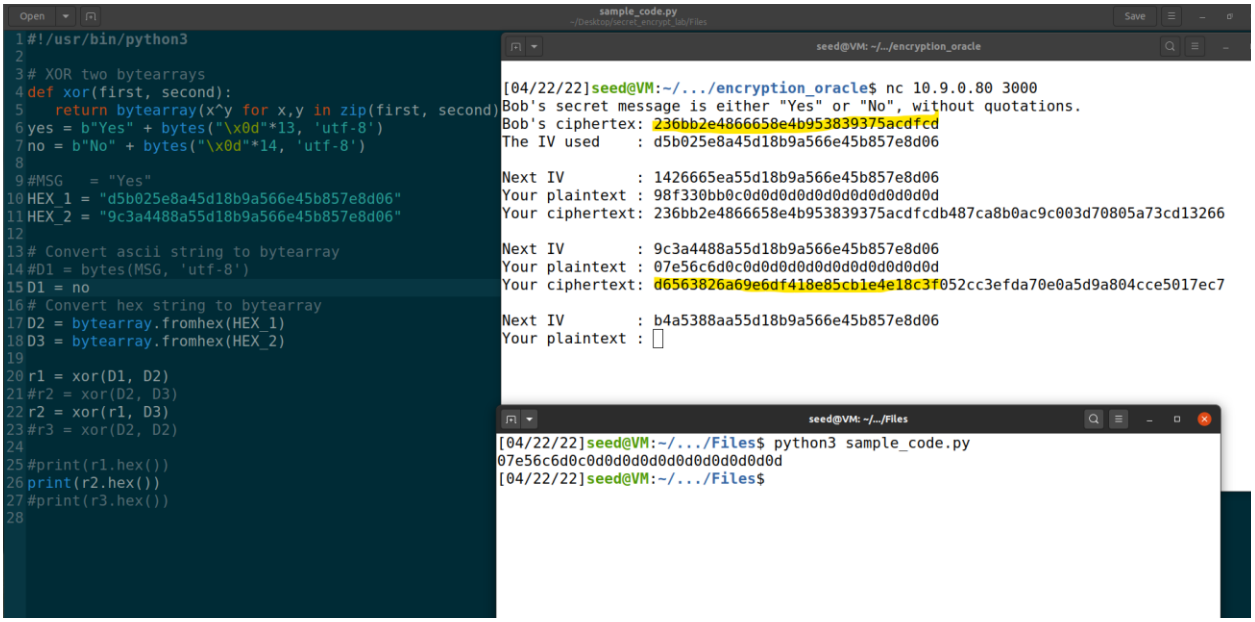
seed@VM: ~/.../Files

```
[04/22/22]seed@VM:~/.../Files$ python3 sample_code.py
98f330bb0c0d0d0d0d0d0d0d0d0d0d0d0d
[04/22/22]seed@VM:~/.../Files$
```

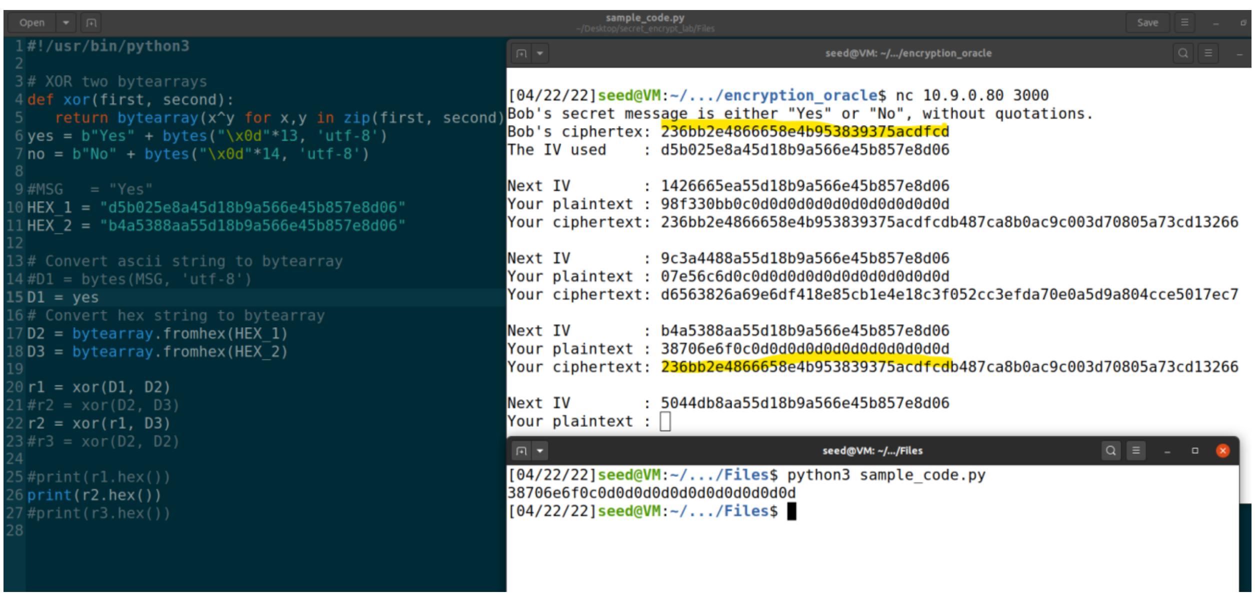
```
[04/22/22]seed@VM:~/.../encryption_oracle$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 236bb2e4866658e4b953839375acdfcd
The IV used    : d5b025e8a45d18b9a566e45b857e8d06

Next IV        : 1426665ea55d18b9a566e45b857e8d06
Your plaintext : 98f330bb0c0d0d0d0d0d0d0d0d0d0d0d
Your ciphertext: 236bb2e4866658e4b953839375acdfcdb487ca8b0ac9c003d7080
5a73cd13266
```

No after the XOR process, we get the plaintext. On putting the plaintext into the oracle I notice that the starting block of cipher text generated is similar to bobs ciphertext. Hence my prediction of Yes seems to be right. I confirm this by attempting again with No and then another Yes to check.

With a 'no' prediction on the new next IV :



And now I check again with a prediction of Yes.



This concludes that bobs secret plain text has “Yes” in the content.