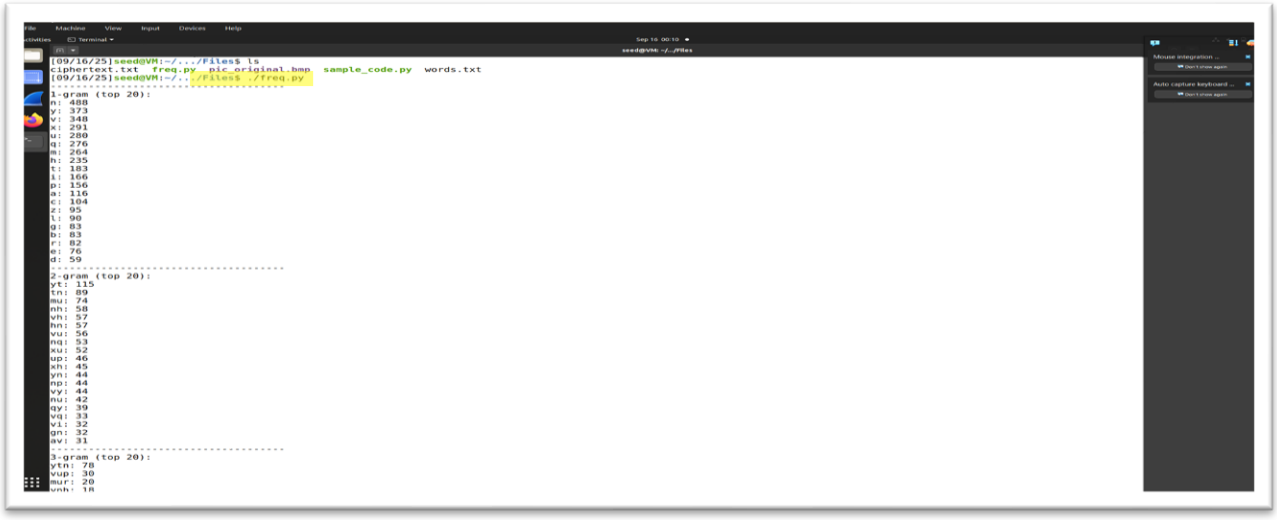
LAB- 01

By :- Faraz Ahmed

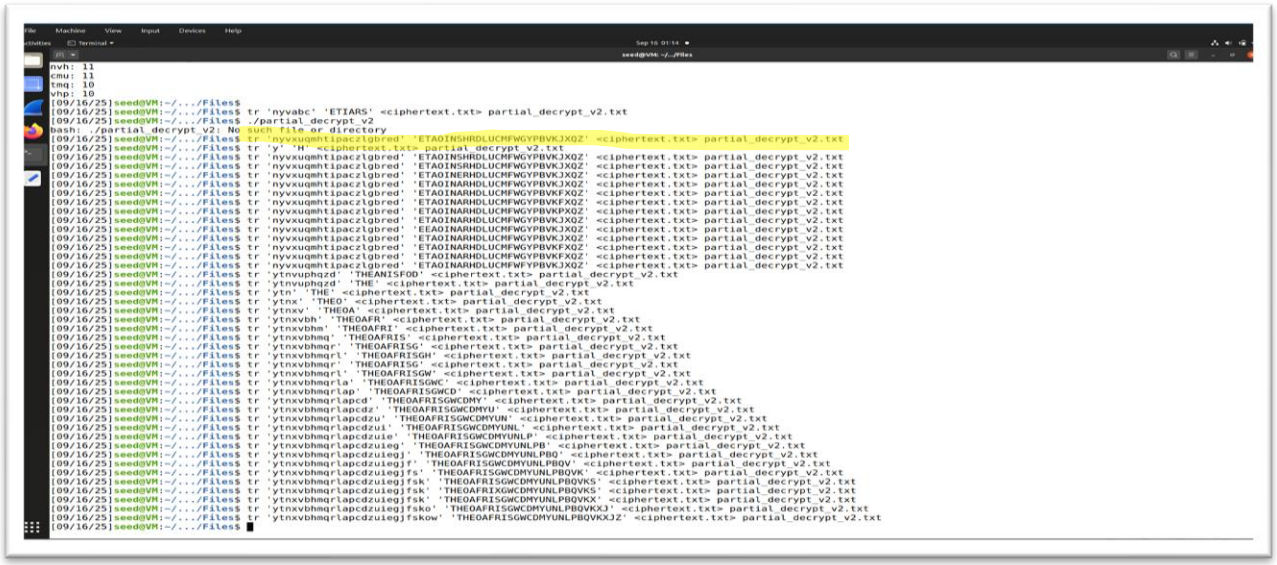
Task 1- Frequency Analysis

- First, we will download Labsetup.zip and unzip it to get Labsetup file.
- Then run the python file freq.py using "./freq.py" command as highlighted in screenshot 1. It is used to analyze the frequency of letters and n-grams in a ciphertext file and can count how often certain letter is present in that ciphertext. (as shown in Screenshot 1)



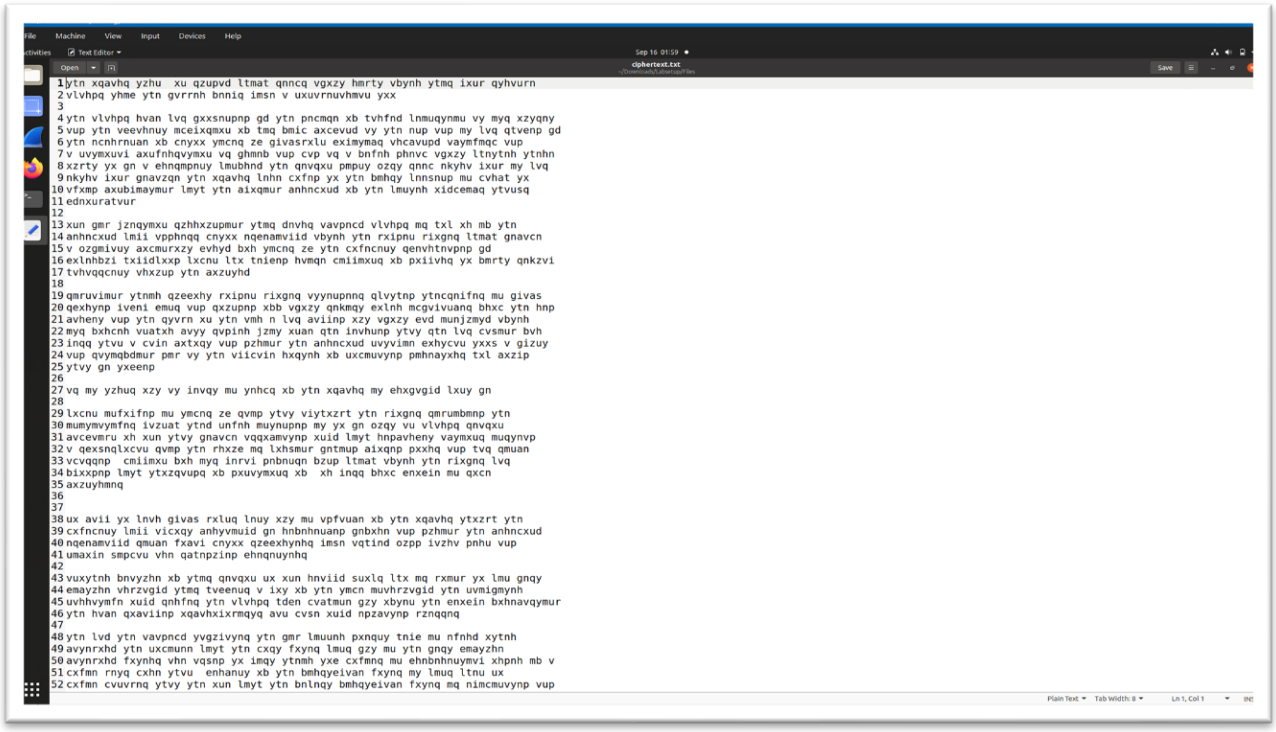
Screenshot 1: frequency of letters in freq.py file.

- Now by using "tr 'nyv...' 'ETI...' <ciphertext.txt> partial_decrypt_v2.txt" command as highlighted in Screenshot 2, we can decrypt the ciphertext by replacing different letters in ciphertext till we get a fully readable plaintext as seen in Screenshot 4.

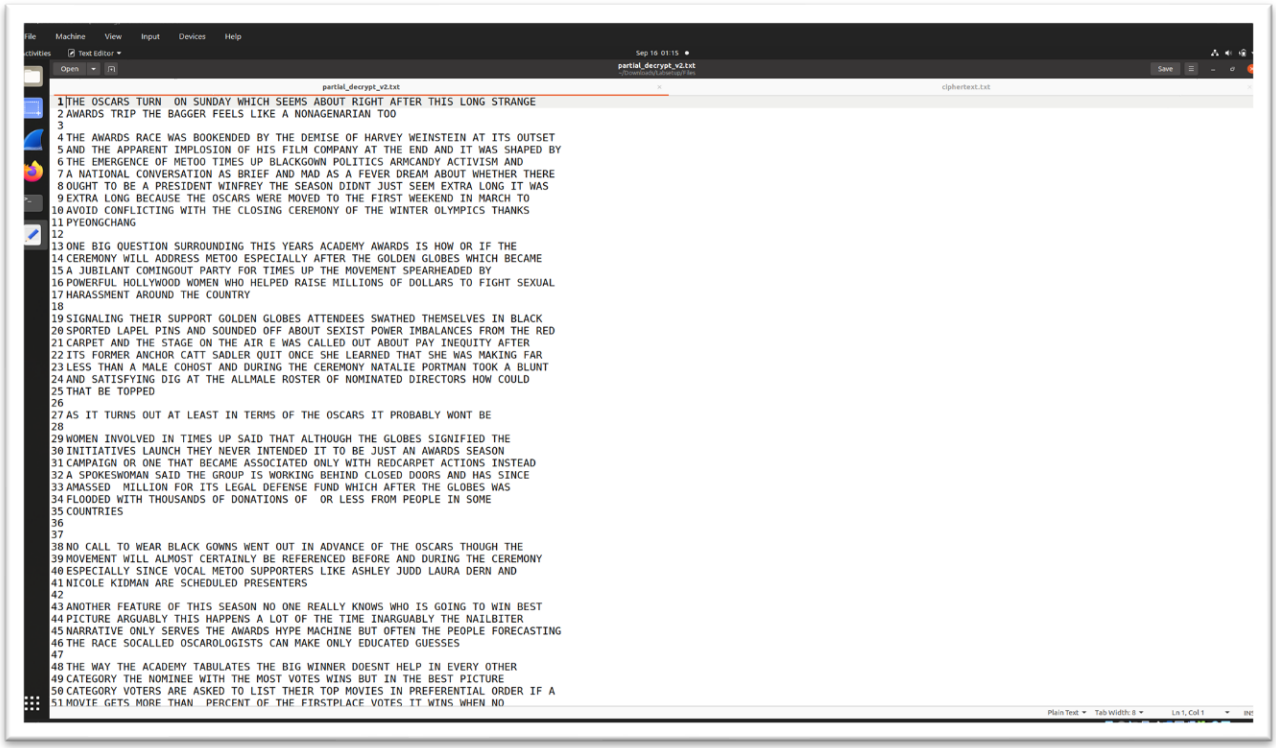


Screenshot 2: Entering command "tr 'nyv...' 'ETI...' <ciphertext.txt> partial_decrypt_v2.txt" to decrypt ciphertext.

- Screenshot 3 is about the original ciphertext which will be converted into fully readable plaintext as shown in Screenshot 4.



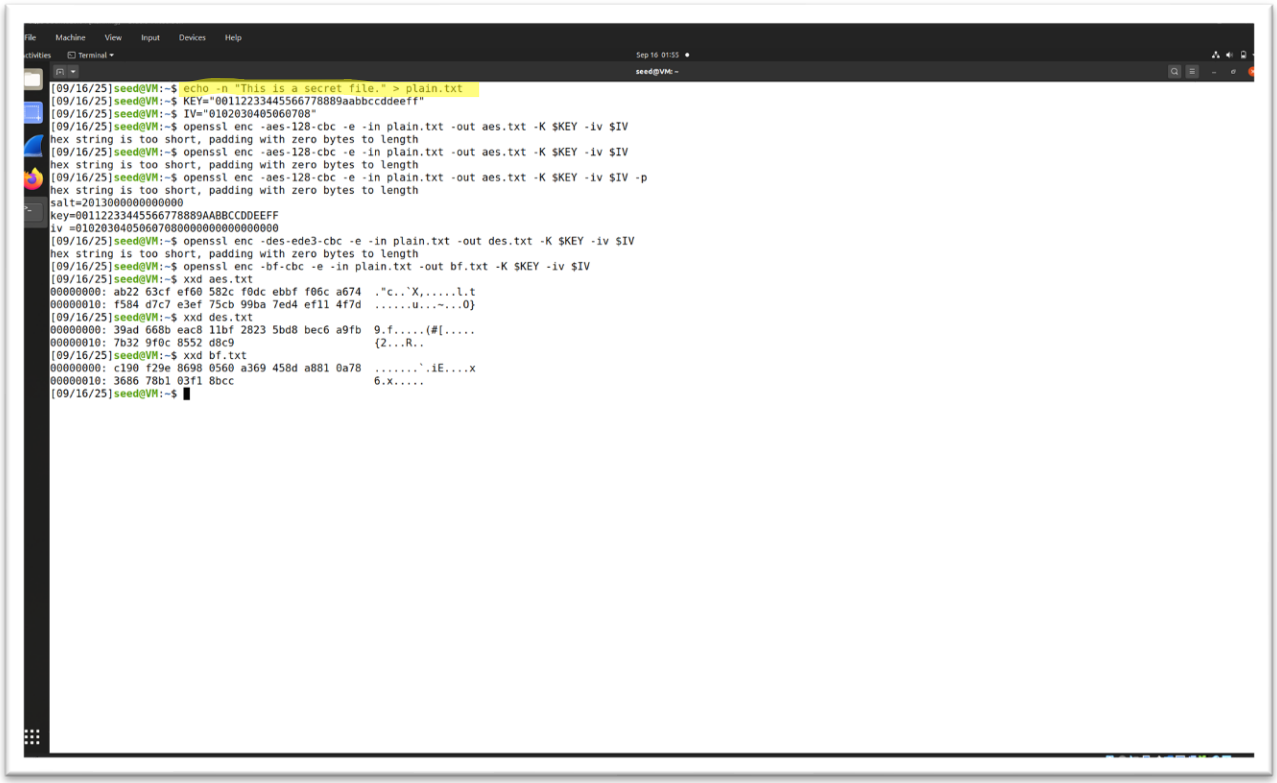
Screenshot 3: Ciphertext with all random letters which is unreadable for anyone.



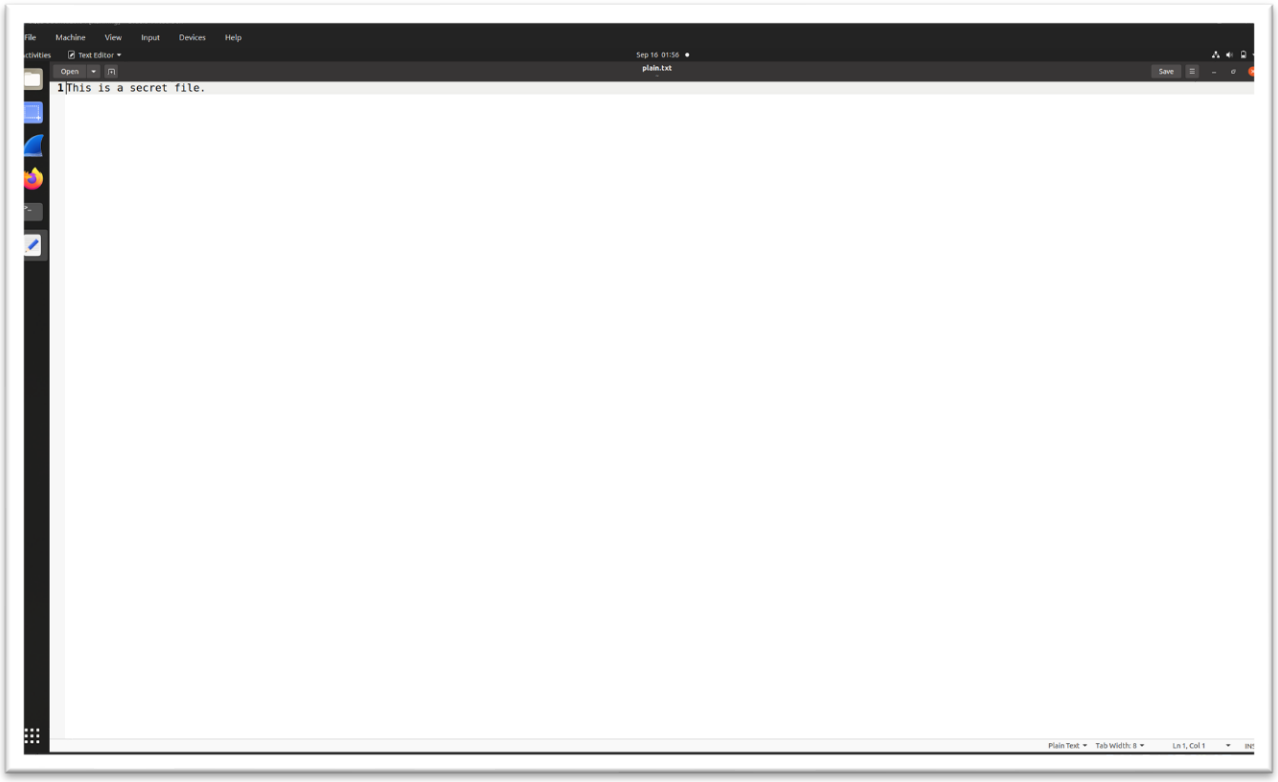
Screenshot 4: By using command mentioned above, we can convert ciphertext into readable plaintext.

Task 2- Encryption using Different Ciphers and Modes.

- So first we can create in new plain file with command “echo -n "This is a secret file." > plain.txt” as highlighted in Screenshot 5. And we can observe the message inside that plain file as observed in Screenshot 6. Then we enter 32 hex digits as key (128 bits) and 16 hex digits as IV (for CBC modes). Finally, we can encrypt that plain file using “openssl enc -aes-128-cbc -e -in plain.txt -out aes.txt -K $KEY -iv $IV” command (repeat same command for rest DES and Blowfish) and see the result by entering “xxd aes.txt” (same for rest DES and Blowfish) as observed in Screenshot 5. We can also observe the encrypted plaintext in Screenshot 7,8 and 9 for AES, DES and Blowfish respectively.

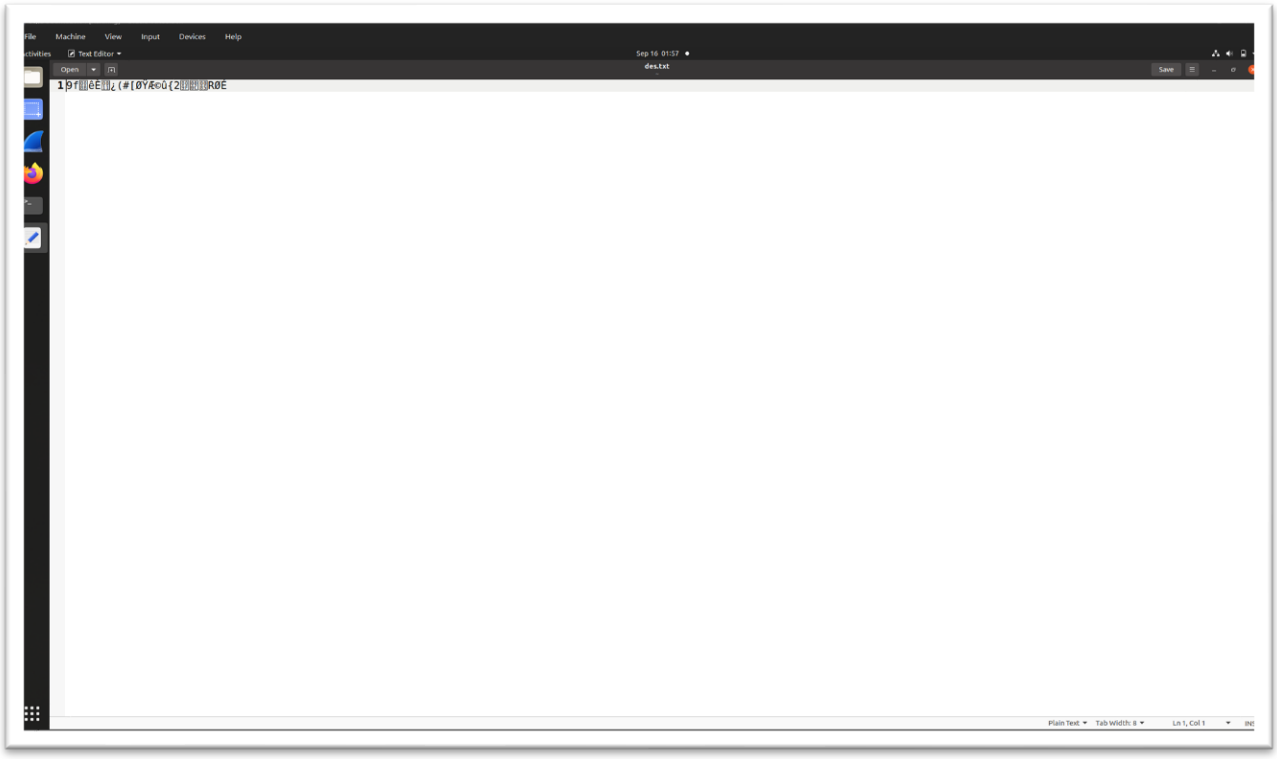


Screenshot 5: Different commands in order to encrypt plaintext into ciphertext using AES, DES and Blowfish.



Screenshot 6: Message inside plain.txt file which we created to encrypt.

Screenshot 7: Encrypted message inside aes.txt which was encrypted using AES.

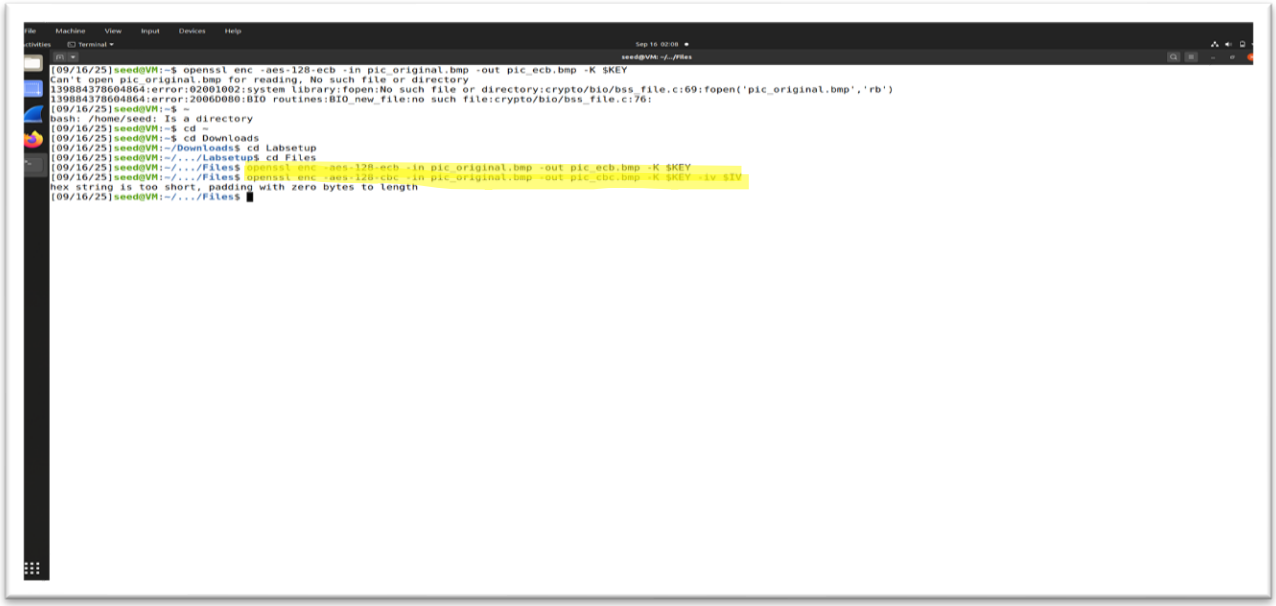


Screenshot 8: Encrypted message inside des.txt which was encrypted using DES.

Screenshot 9: Encrypted message inside bf.txt which was encrypted using Blowfish.

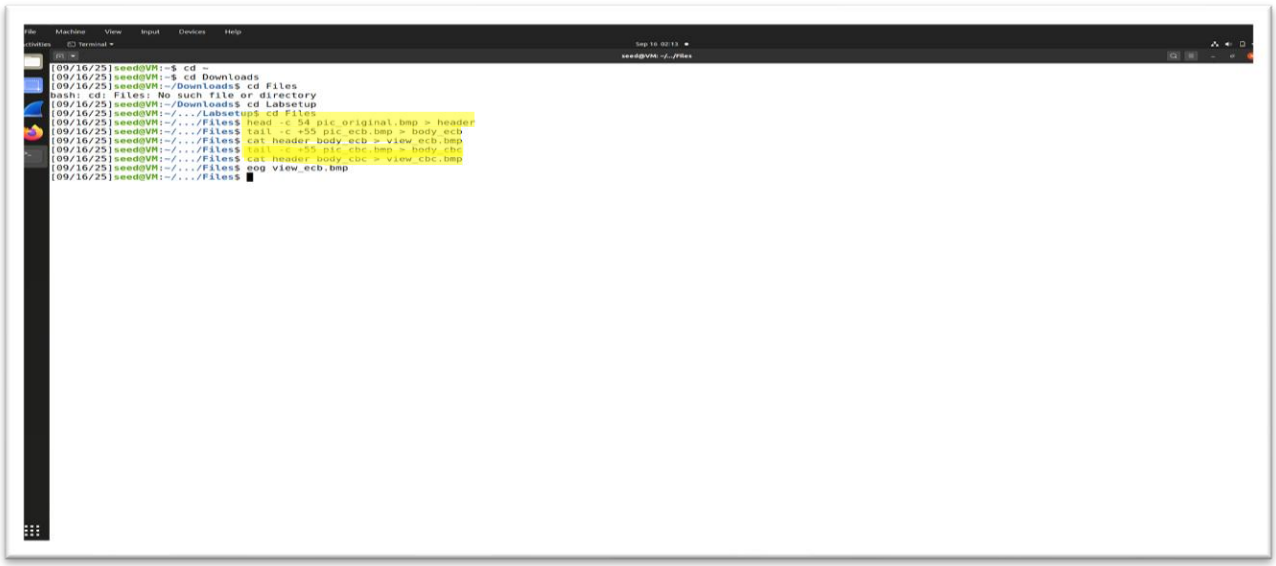
Task 3- Encryption Mode– ECB vs. CBC.

- Now we will encrypt pic_original.bmp using ECB and CBC by inputting “openssl enc -aes-128-cbc -in pic_original.bmp -out pic_cbc.bmp -K $KEY -iv $IV” command and enter same for ECB (Note- ECB algorithm doesn’t need IV so we don’t need to enter it in command) as highlighted in Screenshot 10.



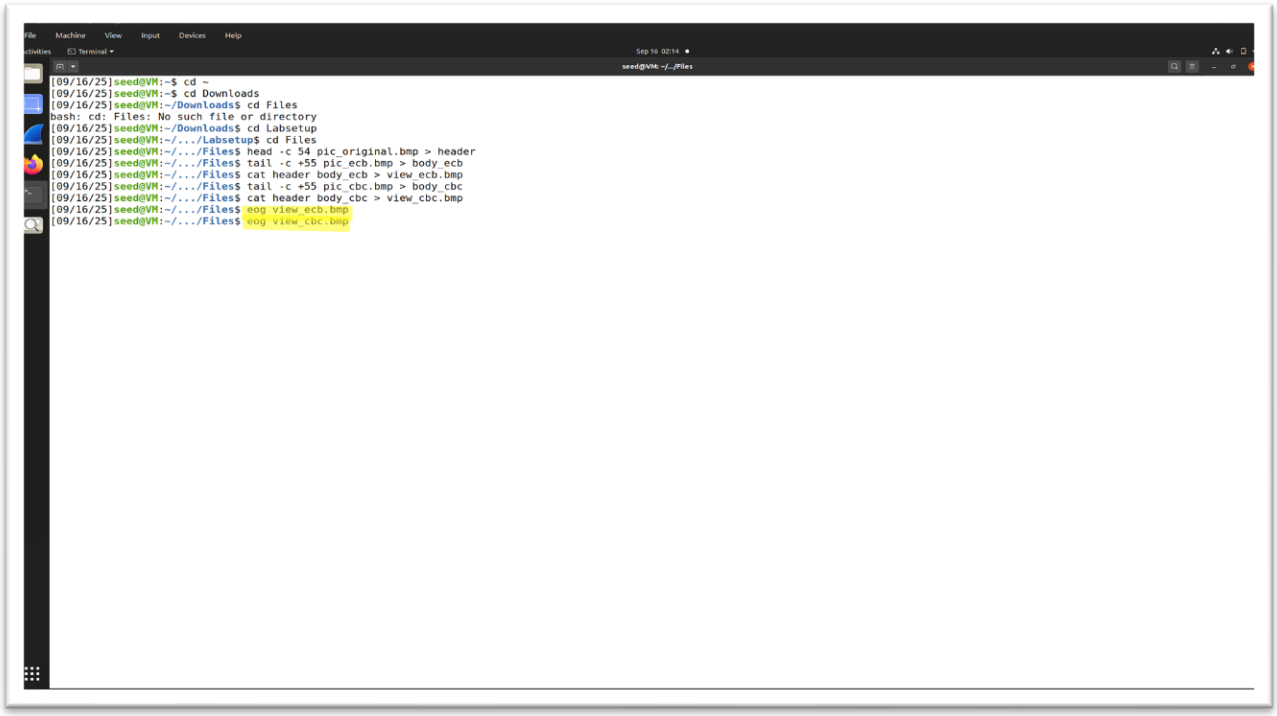
Screenshot 10: Using command to encrypt “pic original.bmp” using ECB and CBC.

- All .bmp files have 54- byte headers so in order to make encrypted files viewable, we enter commands as highlighted in Screenshot 11.

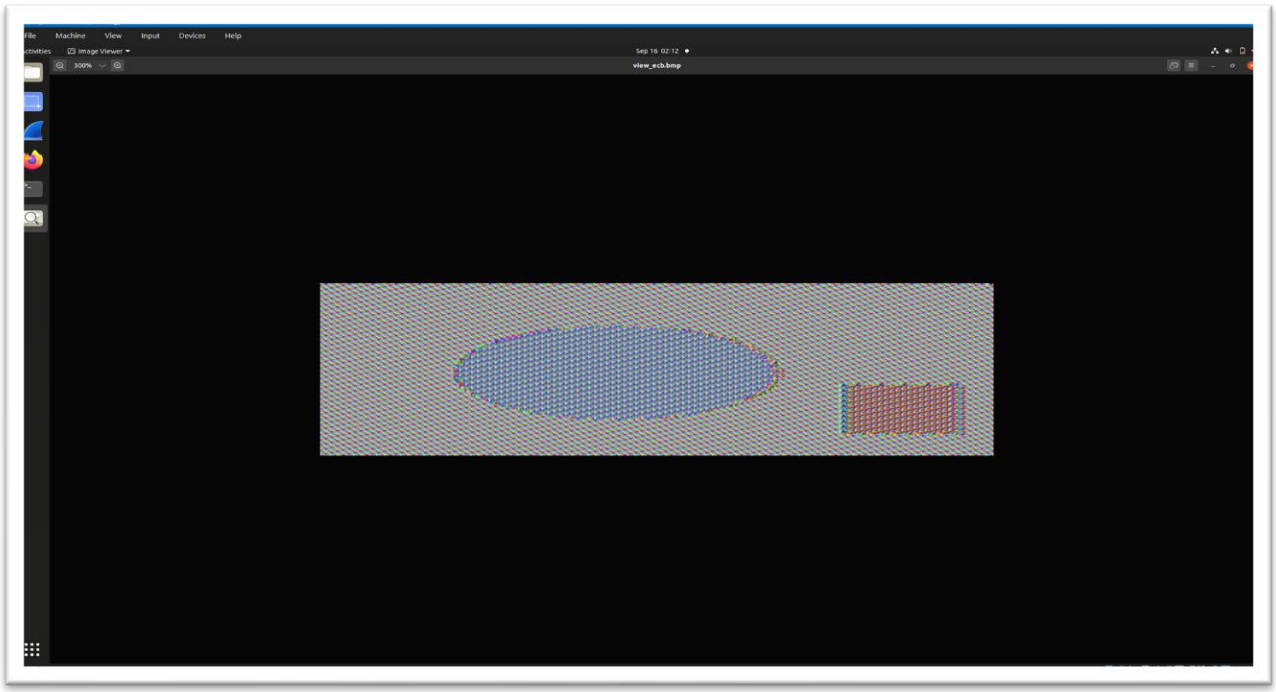


Screenshot 11: Using different commands to make .bmp file viewable.

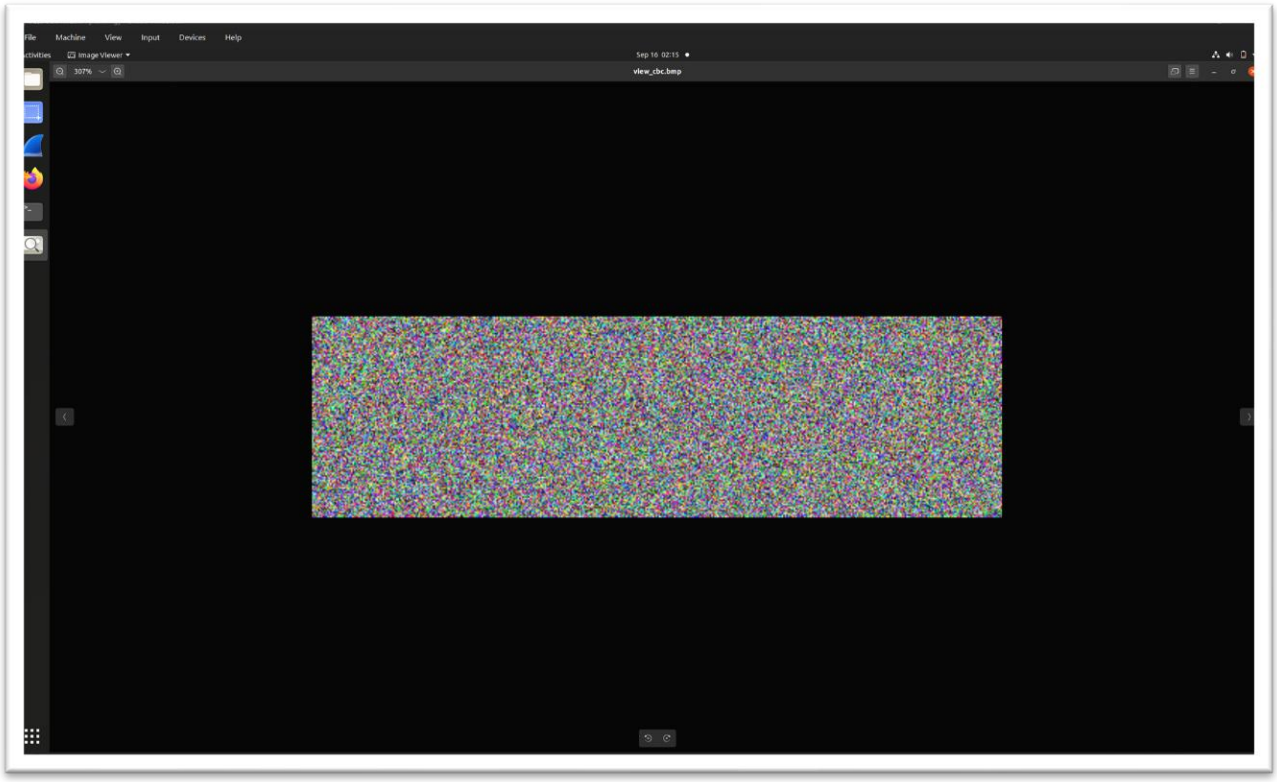
- After that, we will run image viewer using “eog view_ecb.bmp” command and do the same for CBC file which is highlighted in Screenshot 12. The result of this command is shown in Screenshot 13 and 14 for ECB and EBC respectively.



Screenshot 12: Using commands to run image viewer for both encrypted files.



Screenshot 13: Image observed after encrypting “pic_ original.bmp” using ECB.

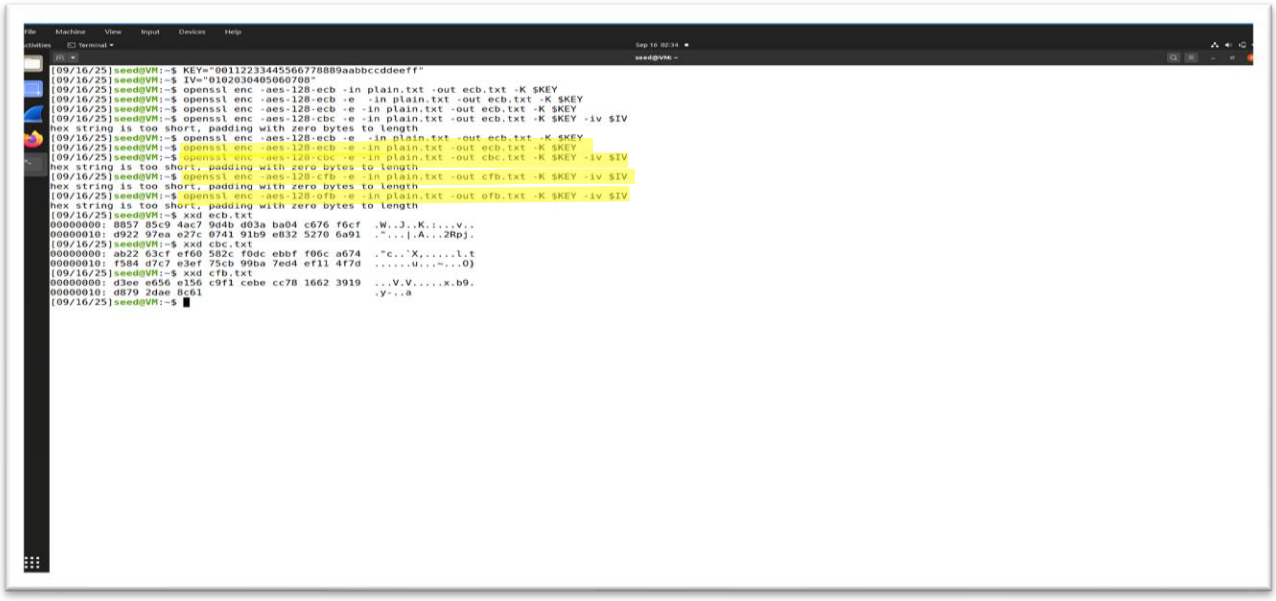


Screenshot 14: Image observed after encrypting "pic_original.bmp" using CBC.

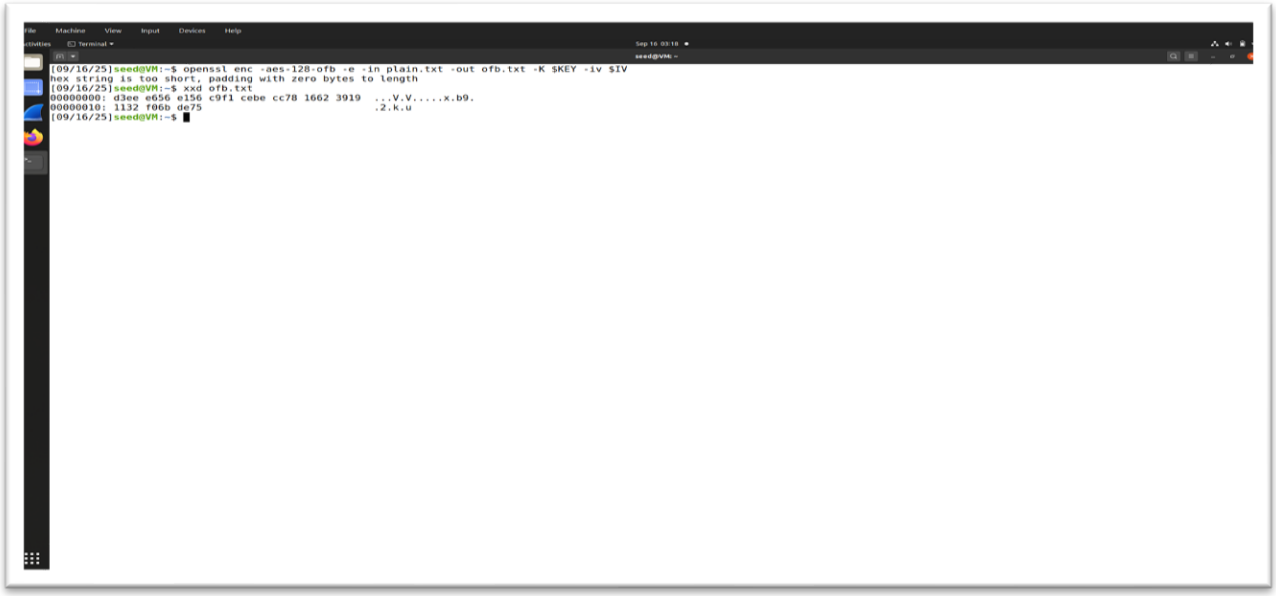
- So, when we observe the result image in Screenshot 13 of ECB, we can notice some visible structures (blue oval and red rectangle) which is revealing the pattern but Screenshot 14 of CBC, there is no visible patterns. Hence, ECB encryption still reveals plaintext information while CBC doesn't.

Task 4- Padding.

- Now, we will encrypt each file by using “openssl enc -aes-128-ecb -e -in plain.txt -out ecb.txt -K $KEY” (include -iv $IV for other three encryptions). So, we will enter commands as highlighted in Screenshot 15 and 16. Then we enter “xxd ecb.txt” command (and same for rest) which will show us the padding. We can observe from Screenshot 15 and 16 that ECB and CBC has some extra padding while CFB and OFB (which are stream ciphers) lacks it.

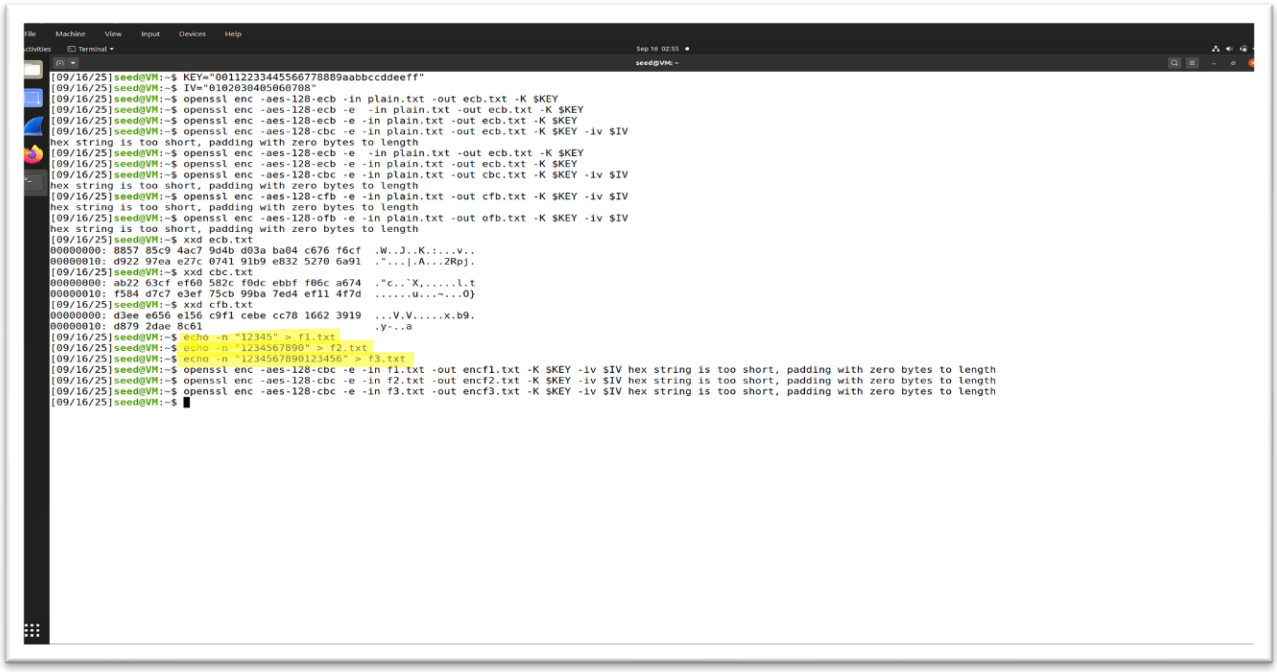


Screenshot 15: To check padding in all four different modes (ECB, CBC and CFB).



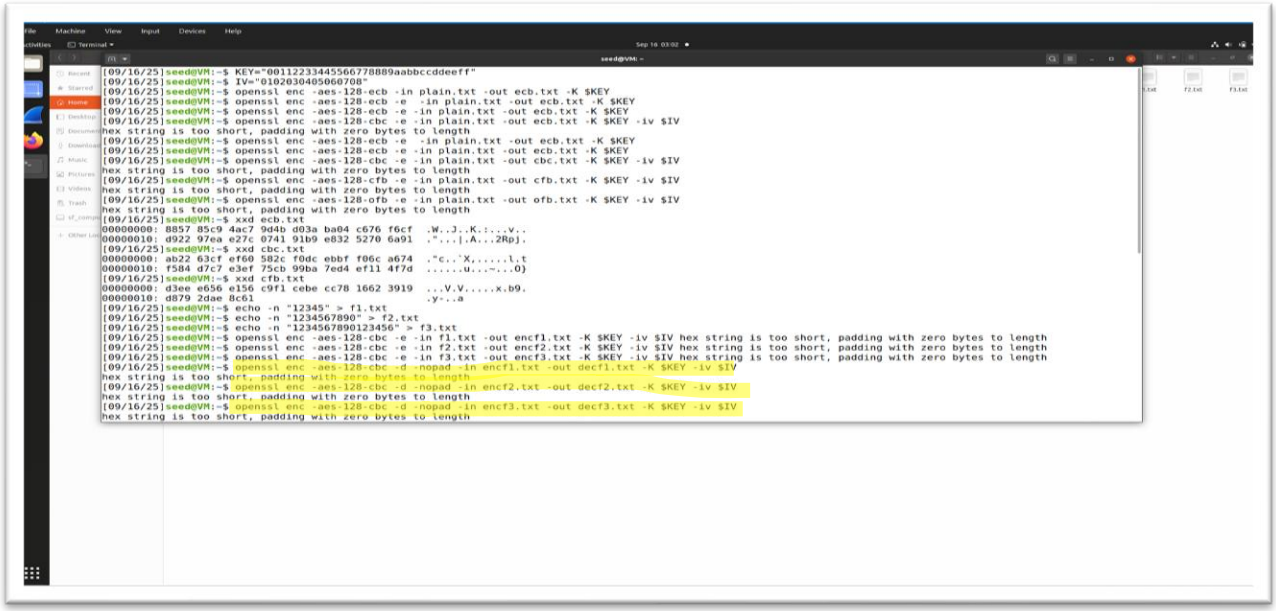
Screenshot 16: To check padding in all four different modes (OFB).

- Now, first we will create three new different-length plaintext using “echo -n "12345" > f1.txt” as highlighted in Screenshot 17. Then we will encrypt those three new files using CBC as observed in Screenshot 17.



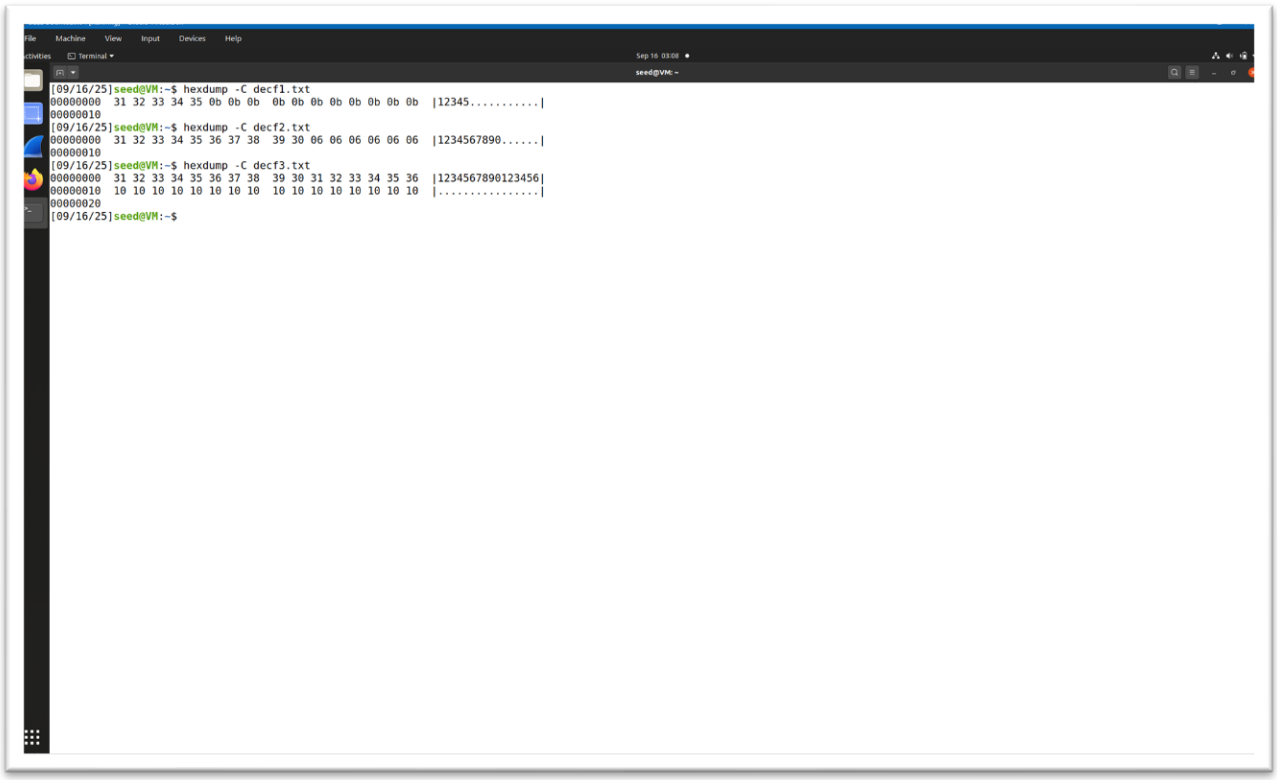
Screenshot 17: Different commands to create three files and encrypt them using CBC.

- Then we will decrypt with no padding removal process using “openssl enc -aes-128-cbc -d -nopad -in encf1.txt -out decf1.txt -K $KEY -iv $IV” (do same for other two encryptions) as highlighted in Screenshot 18.



Screenshot 18: Command “openssl enc -aes-128-cbc -d -nopad -in encf1.txt -out decf1.txt -K $KEY -iv $IV” to decrypt with no padded removal.

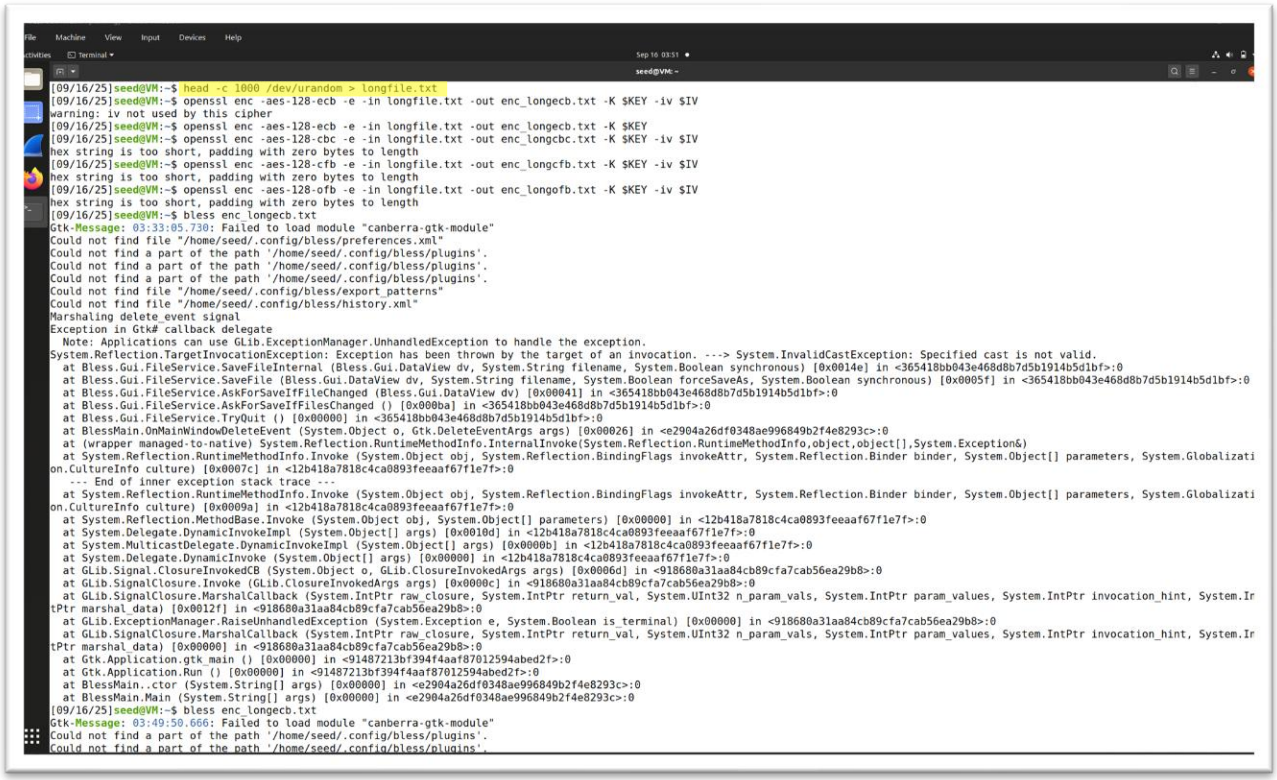
- Then finally, we can use “hexdump -C decf1.txt” command (do the same for other two files) as highlighted in Screenshot 19. So, we can observe that there are some padding bytes like 0b for 11-byte padding and decf3 doesn’t have any as length is maximum to fill padding in it as we can observe in Screenshot 19.



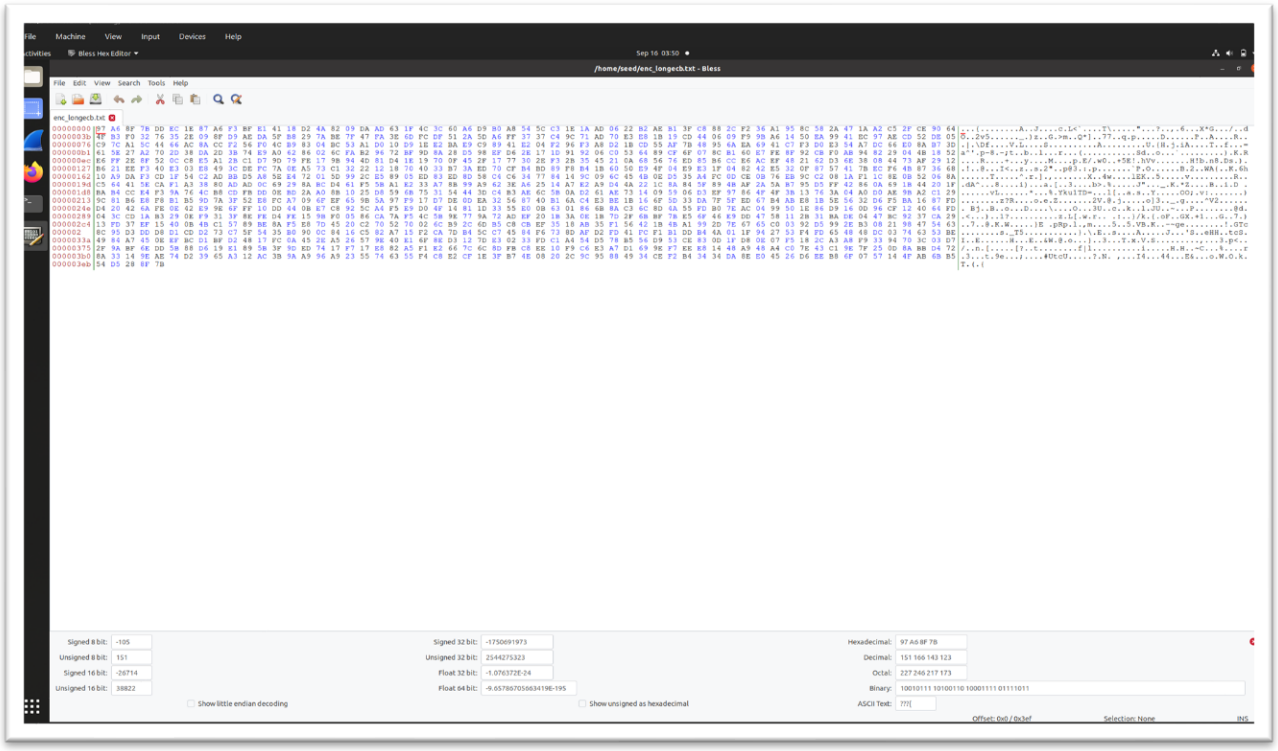
Screenshot 19: Using command “hexdump -C decf1.txt” to check for present padding bytes “0b” in them.

Task 5- Error Propagation– Corrupted Cipher Text.

- First, we will create a 1000 bytes long text file using “head -c 1000 /dev/urandom > longfile.txt” command as highlighted in Screenshot 20. Then we will encrypt file with AES for all four modes (ECB, CBC, CFB, OFB) using “openssl enc -aes-128-ecb -e -in longfile.txt -out enc_longecb.txt -K $KEY” (put -iv -IV on other three modes) as observed in Screenshot 20. Then we need to corrupt 55th byte inside all encrypted files using “bless enc_longecb.txt” and changing value of 55th byte as observed in Screenshot 21.

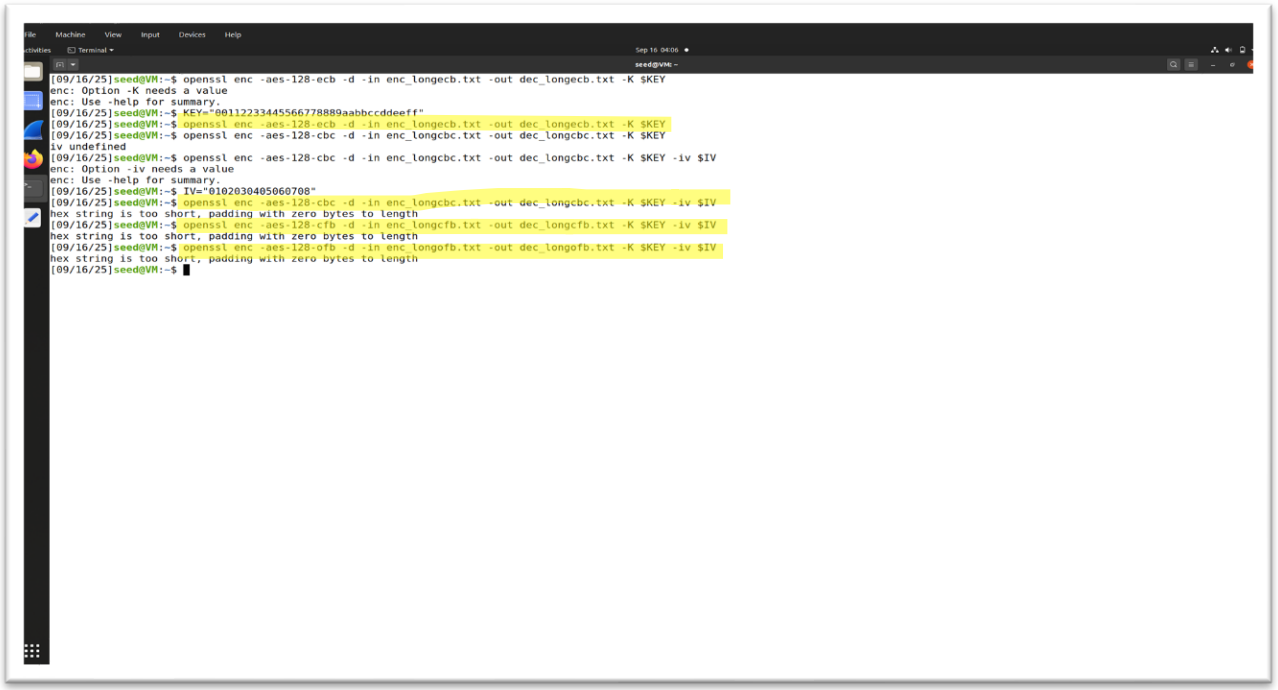


Screenshot 20: Using various commands to create 1000 bytes file, encrypt them with AES and then corrupt 55th byte inside it using “bless” command.



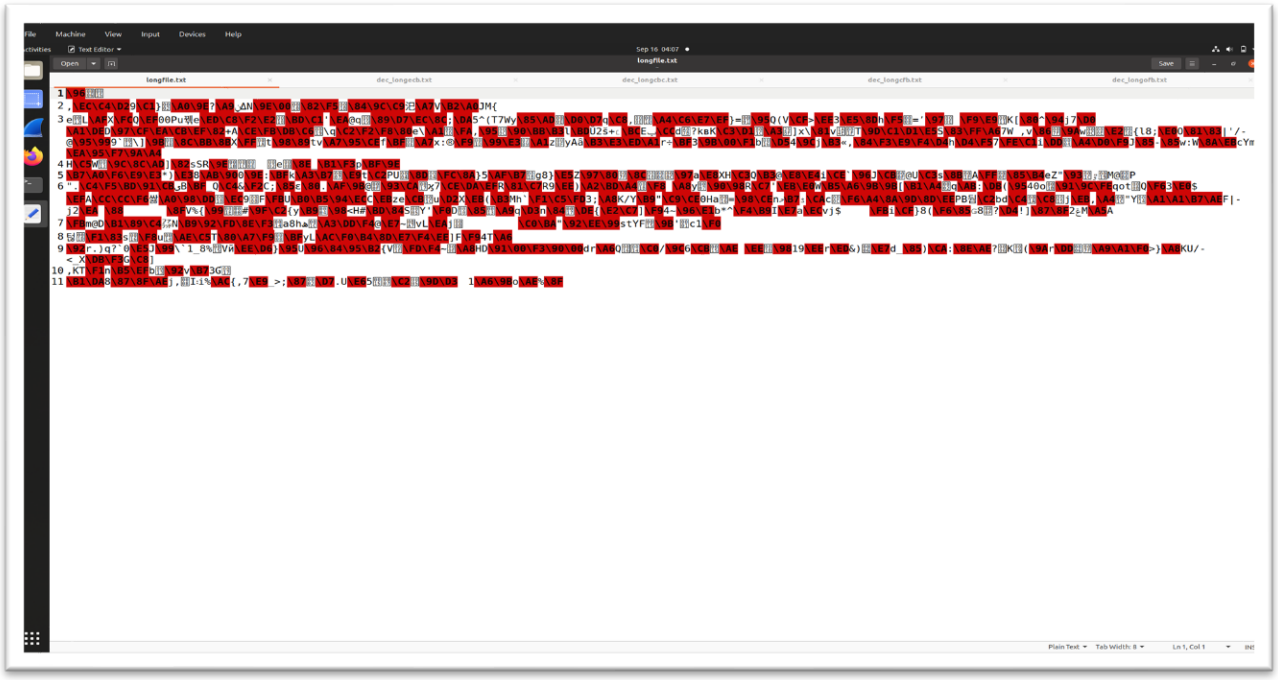
Screenshot 21: Corrupting the 55th byte in encrypted files using "bless" command.

- Then after this, we will decrypt all encrypted files using "openssl enc -aes-128-ecb -d -in enc_longecb.txt -out dec_longecb.txt -K $KEY" for all other modes (include -iv $IV in all other modes) as highlighted in Screenshot 22.



Screenshot 22: Decrypted all encrypted files after corrupting it.

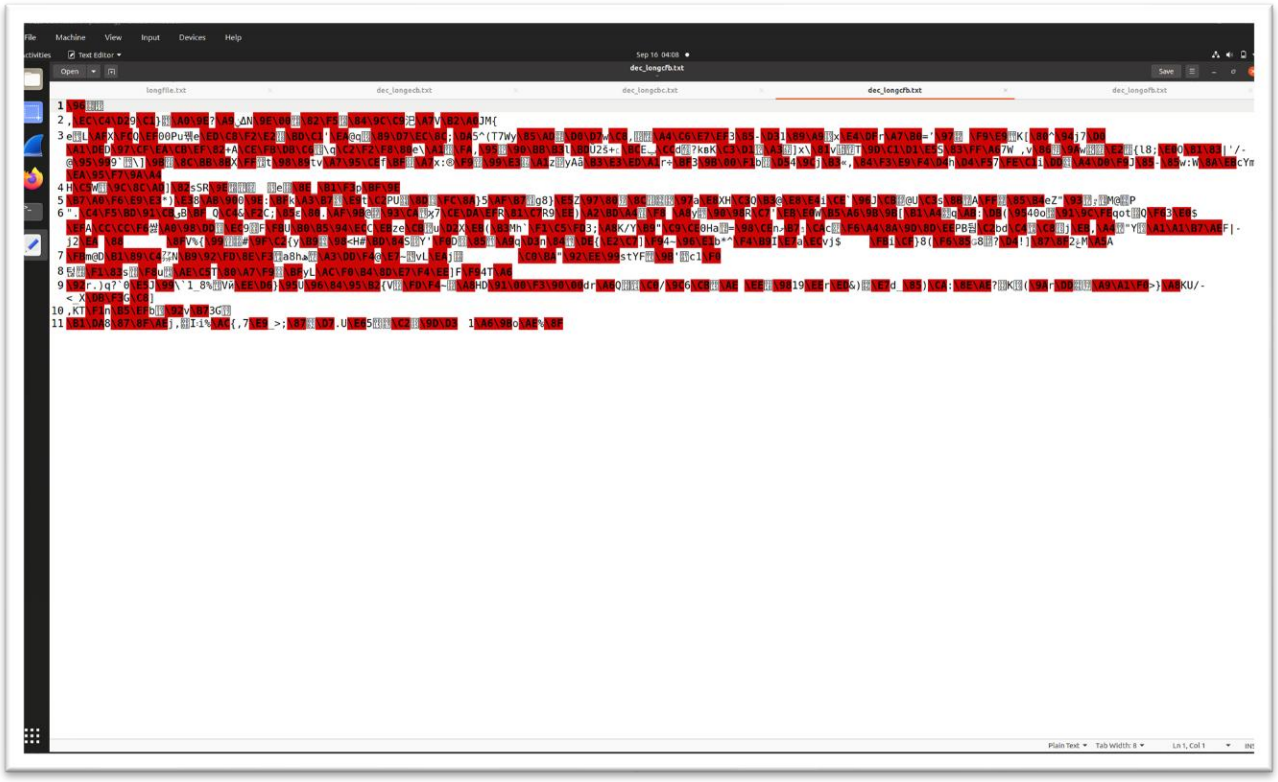
- Now, after that we have to observe any significant changes caused due to curropting files from original files. So, in Screenshot 23, it shows the original file's content and in Screenshots 24,25,26 and 27.



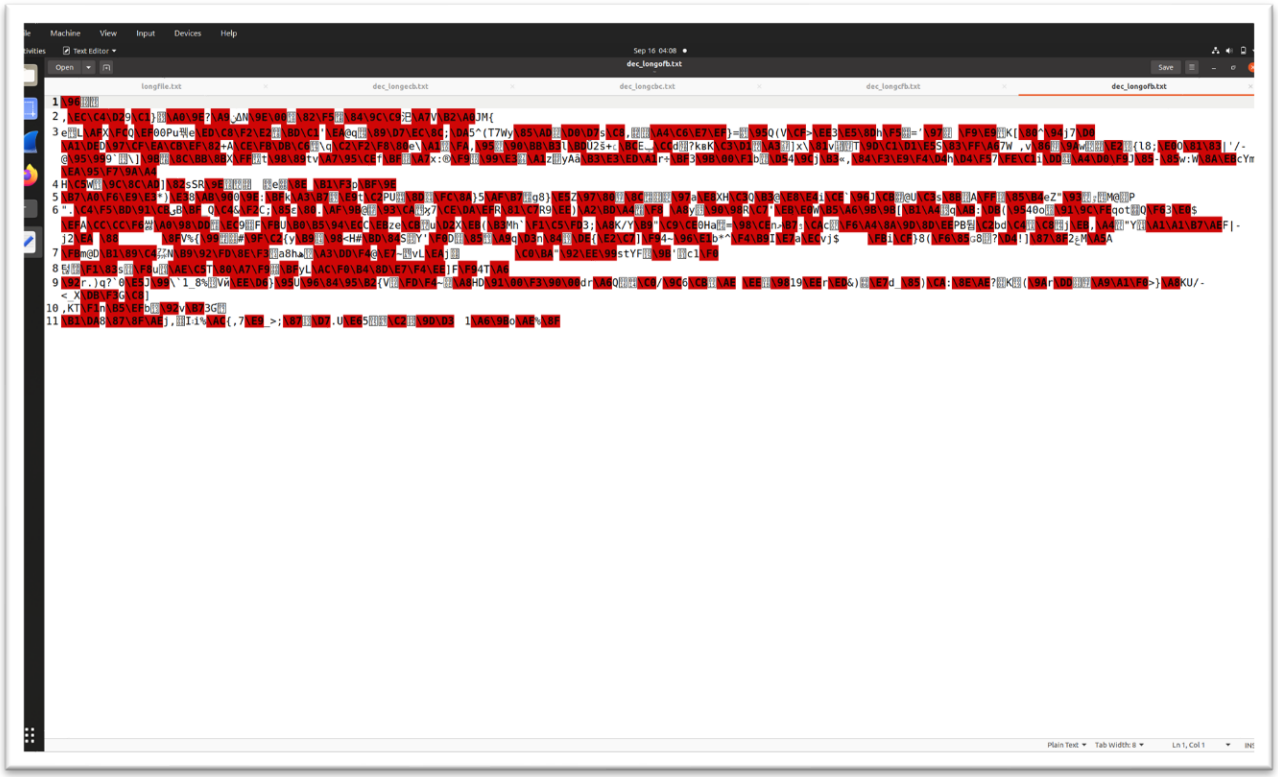
Screenshot 23: Content inside original file "longfile.txt".

Screenshot 24: Changed content inside decrypted file "dec_longecb.txt".

Screenshot 25: Changed content inside decrypted file “dec_ longcbc.txt”.



Screenshot 26: Changed content inside decrypted file “dec_ longcfb.txt”.



Screenshot 27: Changed content inside decrypted file "dec_longofb.txt".

- So, finally after all, we can observe and report from Screenshots 24,25,26 and 27 that:-
- ECB- Only the block which was corrupted are changed.
- CBC- The block with error and the next block so total two blocks are changed.
- CFB/OFB- Only the byte which was corrupted is altered.

Task 6- Initial Vector (IV) and Common Mistakes.

- **Given:-** P1= “This is a known message!”

  C1= a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

  C2= bf73bcd3509299d566c35b5d450337e1bb175f903fafc159

- **Find:-** P2
- So, we will use K= P1 ⊕ C1

  P2= C2 ⊕ K (K is same for OFB mode)
- Now, P1= “This is a known message!”, we need to find Hex(ASCII) value

Character	Decimal	Hex
T	84	54
h	104	68
i	105	69
s	115	73
(space)	32	20
i	105	69
s	115	73
(space)	32	20
a	97	61
(space)	32	20
k	107	6b
n	110	6e
o	111	6f
w	119	77
n	110	6e
(space)	32	20
m	109	6d
e	101	65
s	115	73
s	115	73
a	97	61
g	103	67
e	101	65
!	33	21

So, whole Hex value = 54 68 69 73 20 69 73 20 61 20 6b 6e 6f 77 6e 20 6d 65 73 73 61 67 65 21 with length= 24 bytes.

- K= P1 ⊕ C1

Now, XOR byte by byte:

- 54 ⊕ a4 = f0
- 68 ⊕ 69 = 01
- 69 ⊕ b1 = d8
- 73 ⊕ c5 = b6
- 20 ⊕ 02 = 22
- 69 ⊕ c1 = a8
- 73 ⊕ ca = b9
- 20 ⊕ b9 = 99
- 61 ⊕ 66 = 07
- 20 ⊕ 96 = b6
- 6b ⊕ 5e = 35
- 6e ⊕ 50 = 3e
- 6f ⊕ 42 = 2d
- 77 ⊕ 54 = 23
- 6e ⊕ 38 = 56
- 20 ⊕ e1 = c1
- 6d ⊕ bb = d6
- 65 ⊕ 1b = 7e
- 73 ⊕ 5f = 2c
- 73 ⊕ 90 = e3
- 61 ⊕ 37 = 56
- 67 ⊕ a4 = c3
- 65 ⊕ c1 = a4
- 21 ⊕ 59 = 78

So, K = f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a478

- P2= C2 ⊕ K

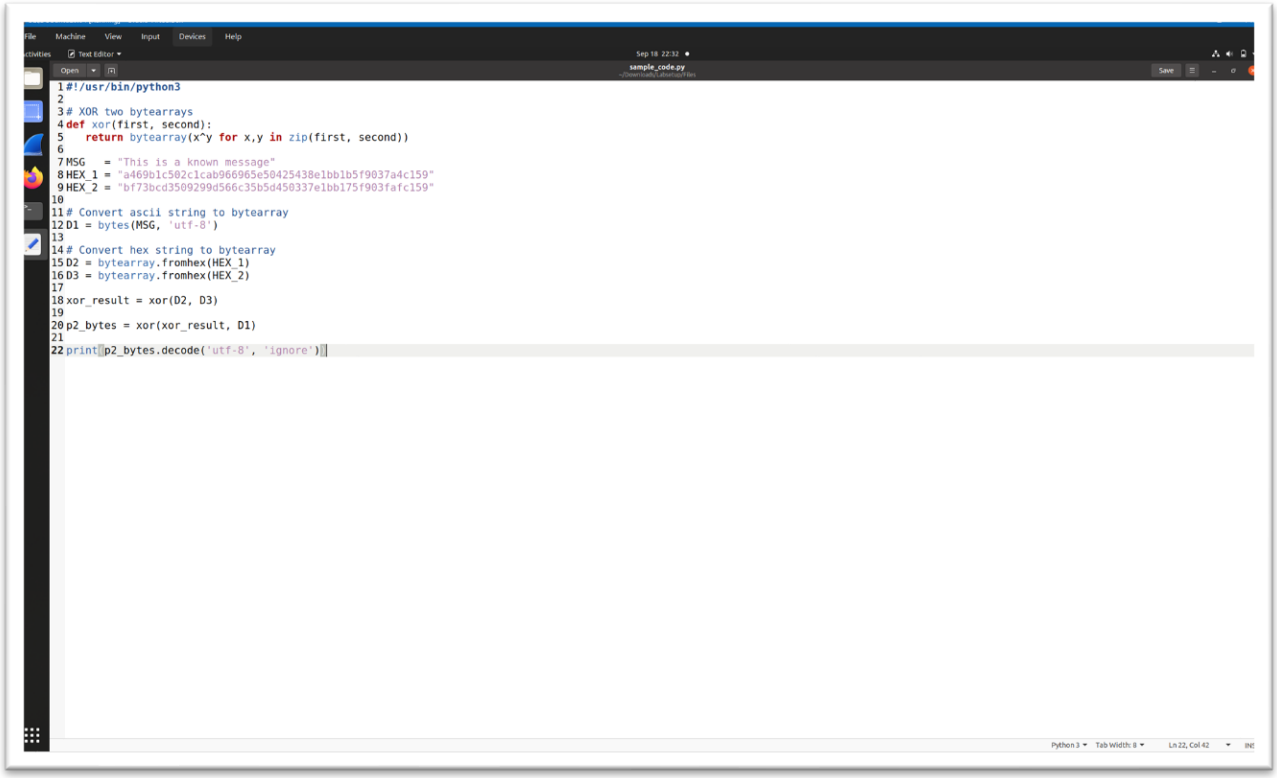
  Now XOR byte by byte:

- bf ⊕ f0 = 4f (ASCII O)
- 73 ⊕ 01 = 72 (ASCII r)
- bc ⊕ d8 = 64 (ASCII d)
- d3 ⊕ b6 = 65 (ASCII e)
- 50 ⊕ 22 = 72 (ASCII r)

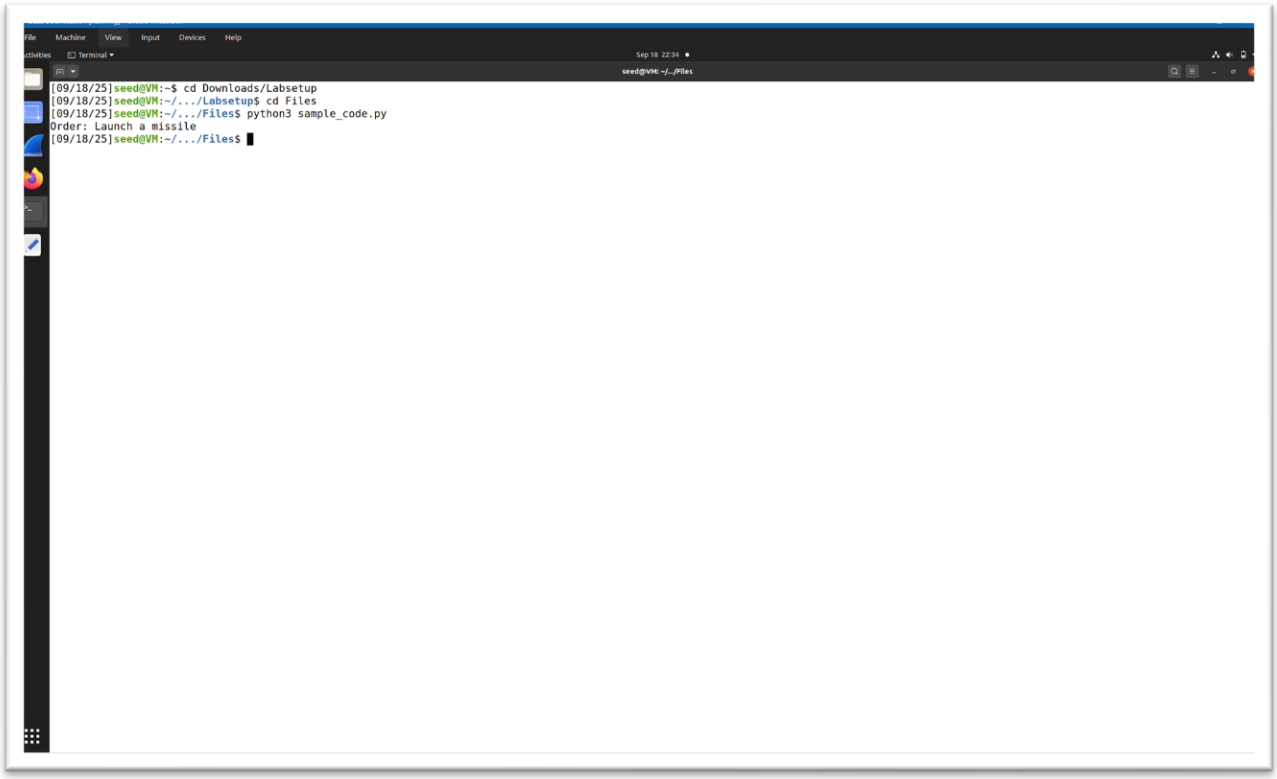
- 92 ⊕ a8 = 3a (ASCII :)
- 99 ⊕ b9 = 20 (space)
- d5 ⊕ 99 = 4c (ASCII L)
- 66 ⊕ 07 = 61 (ASCII a)
- c3 ⊕ b6 = 75 (ASCII u)
- 5b ⊕ 35 = 6e (ASCII n)
- 5d ⊕ 3e = 63 (ASCII c)
- 45 ⊕ 2d = 68 (ASCII h)
- 03 ⊕ 23 = 20 (space)
- 37 ⊕ 56 = 61 (ASCII a)
- e1 ⊕ c1 = 20 (space)
- bb ⊕ d6 = 6d (ASCII m)
- 17 ⊕ 7e = 69 (ASCII i)
- 5f ⊕ 2c = 73 (ASCII s)
- 90 ⊕ e3 = 73 (ASCII s)
- 3f ⊕ 56 = 69 (ASCII i)
- af ⊕ c3 = 6c (ASCII l)
- c1 ⊕ a4 = 65 (ASCII e)
- 59 ⊕ 78 = 21 (ASCII !)

➢ So, P2= "Order: Launch a missile!"

- Here are some changes which I did to “sample_code.py” in order to get P2 as seen in Screenshot 28.



Screenshot 28: Newly updated “sample_code.py” to find P2.



Screenshot 29: Found P2 which is “Order: Launch a missile”.

- First, we will choose the guess from “Yes” or “No”.
- Bob’s ciphertext= our ciphertext which is C1= E(P(Guess) ⊕IV1) and C2= E(P ⊕IV2)
- So, P= IV1 ⊕ IV2 ⊕ P(Guess) where IV1= Bob’s IV, IV2= next IV and P= Hex of Guess from “Yes” or “No”.
- ASCII hex value for “Yes” is 596573 and for “No” is 4e6f.
- We will connect to Oracle with commands nc 10.9.0.80 3000 and receive Bob’s ciphertext, IV used and next IV as observed in Screenshot 28. Then we need to enter your plaintext which we calculated from P= IV1 ⊕ IV2 ⊕ P(Guess) formula (Taking guess as Yes) .
- After that we will receive ciphertext from oracle and compare ciphertext with Bob’s secret.
- If they match, then the secret is Yes.
- If they don’t match, then the secret is No.
- So, from Screenshot 30 we can see that the

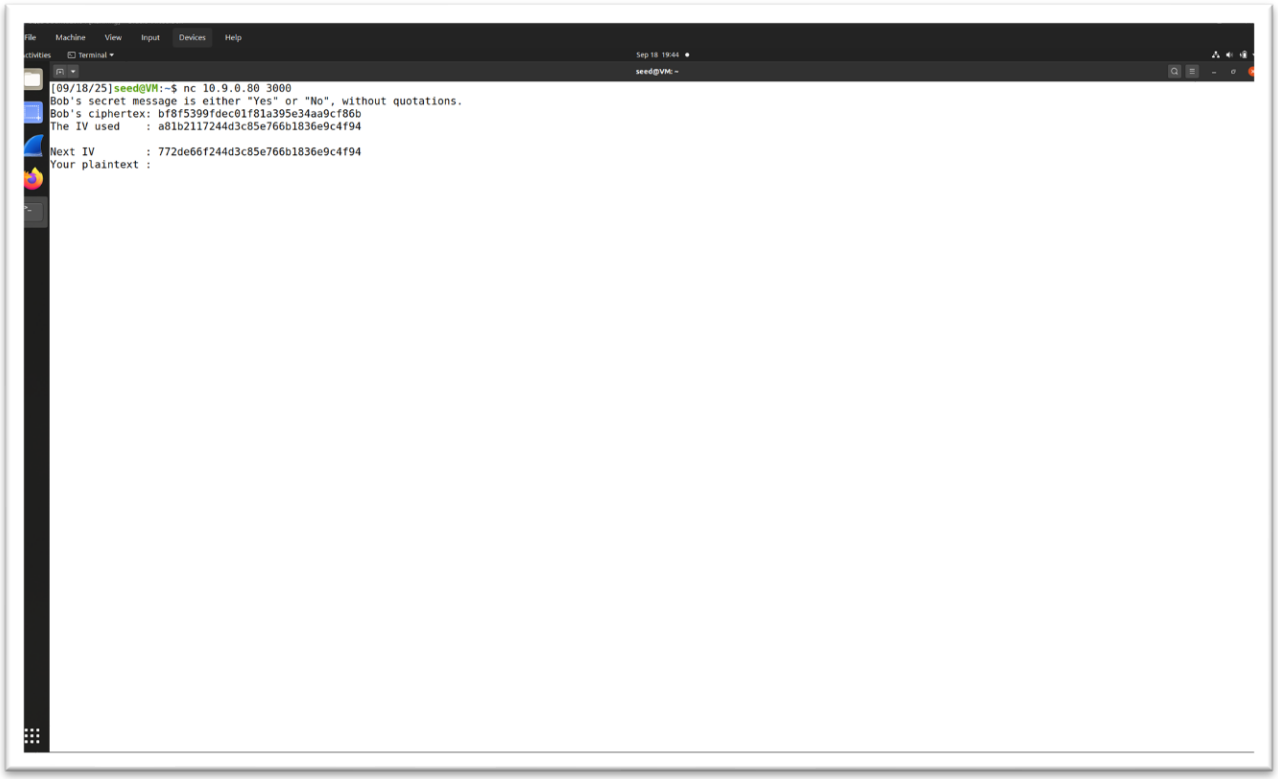
❖ Bob's secret ciphertext: bf8f5399fdec01f81a395e34aa9cf86b

❖ Bob's original IV: a81b2117244d3c85e766b1836e9c4f94

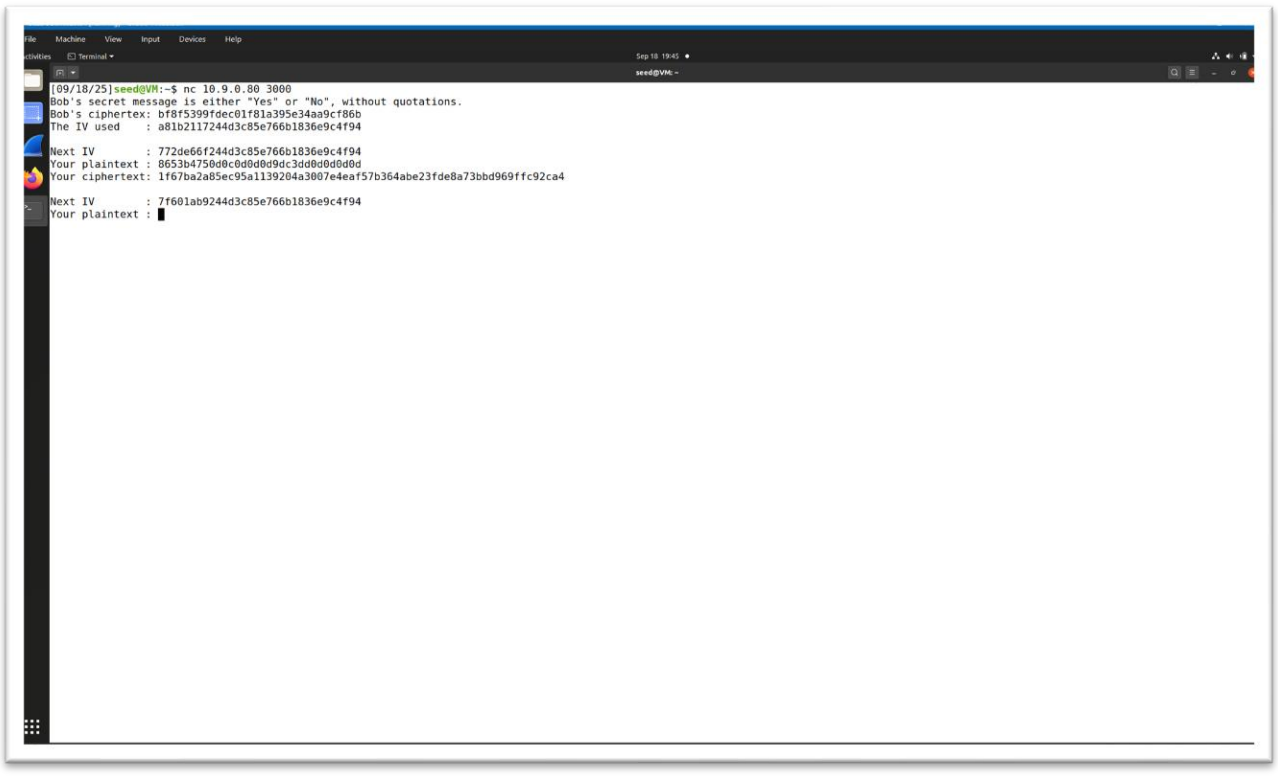
❖ Next IV: 772de66f244d3c85e766b1836e9c4f94

❖ And Secret guess is “Yes” (5965730d0d0d0d0d0d0d0d0d0d0d0d0d with padding).

- So, P= 5965730d0d0d0d0d0d0d0d0d0d0d0d0d ⊕ a81b2117244d3c85e766b1836e9c4f94 ⊕ 772de66f244d3c85e766b1836e9c4f94
- First calculate first part- 5965730d0d0d0d0d0d0d0d0d0d0d0d ⊕ a81b2117244d3c85e766b1836e9c4f94 = f17e521a29413188eafbf25a63919299 then use result from that for second part- f17e521a29413188eafbf25a63919299 ⊕ 772de66f244d3c85e766b1836e9c4f94 = 8653b4750d0c0d0d0d9dc3dd0d0d0d0d
- P= 8653b4750d0c0d0d0d9dc3dd0d0d0d0d which I entered in as observed in Screenshot 31 and the ciphertext doesn’t match with Bob’s ciphertext which means that the right secret was “No”.



Screenshot 30: Ciphertext received from Oracle by entering "nc 10.9.0.80 3000".



Screenshot 31: Result ciphertext which we will match with Bob's ciphertext to check for secret message.

References:-

- ChatGPT for some commands recommendations.
- Ask Ubuntu for some doubt during encryption.
- https://www.google.com to search steps for setup.