# INTERNET SECURITY LAB 8

## Encryption Lab

Akarsh Shetty Umesh Mudelkadi
317752264

Key -> 'ytnupmrvcqbhfaixzgdlesjwok' 'THENDIGAMSFRVCLOUBYWPKQMJX'

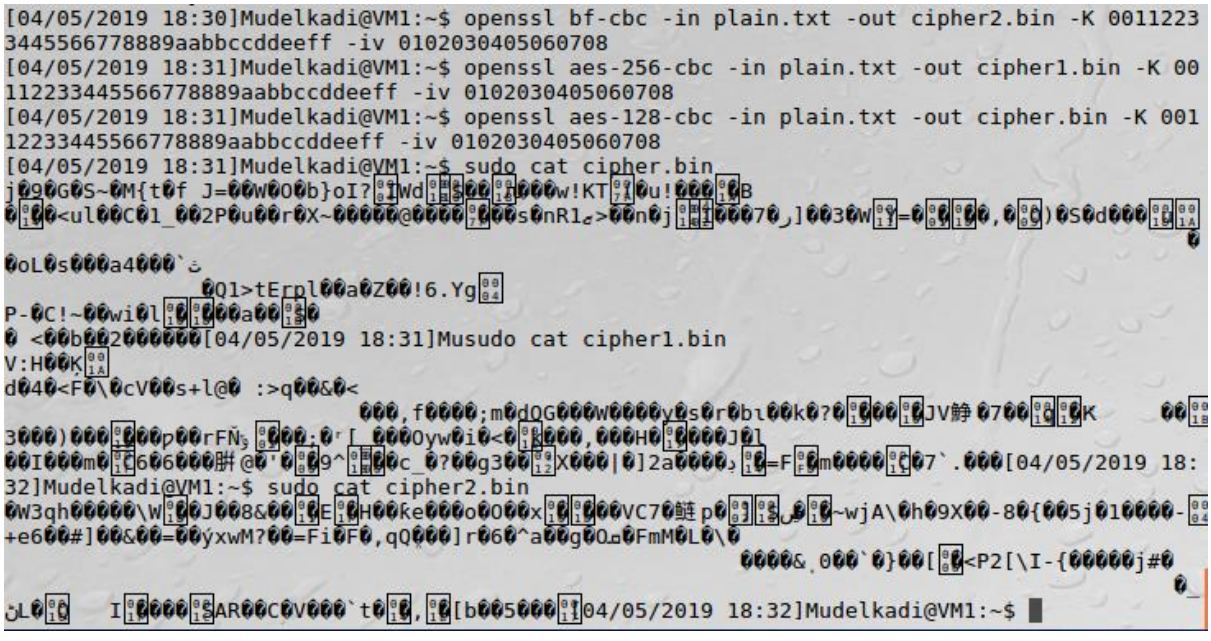
**Task 1: Frequency Analysis Against Monoalphabetic Substitution Cipher:**

```
[04/05/2019 18:06]Mudelkadi@VM1:~$ tr 'ytnupmrvcqbhfaixzgdlesjwok'
 'THENDIGAMSFRVCLOUBYWPKQMJX'<in.txt> out.txt
[04/05/2019 18:07]Mudelkadi@VM1:~$ sudo cat  out.txt
THE OSCARS TURN  ON SUNDAY WHICH SEEMS ABOUT RIGHT AFTER THIS LONG
 STRANGE
AWARDS TRIP THE BAGGER FEELS LIKE A NONAGENARIAN TOO

THE AWARDS RACE WAS BOOKENDED BY THE DEMISE OF HARVEY WEINSTEIN AT
 ITS OUTSET
AND THE APPARENT IMPLOSION OF HIS FILM COMPANY AT THE END AND IT W
AS SHAPED BY
THE EMERGENCE OF METOO TIMES UP BLACKGOWN POLITICS ARMCANDY ACTIVI
SM AND
A NATIONAL CONVERSATION AS BRIEF AND MAD AS A FEVER DREAM ABOUT WH
```

This is done by guessing, as you can see the encryption key in the picture for retrieving the actual info.

**Task 2: Encryption using Different Ciphers and Modes**

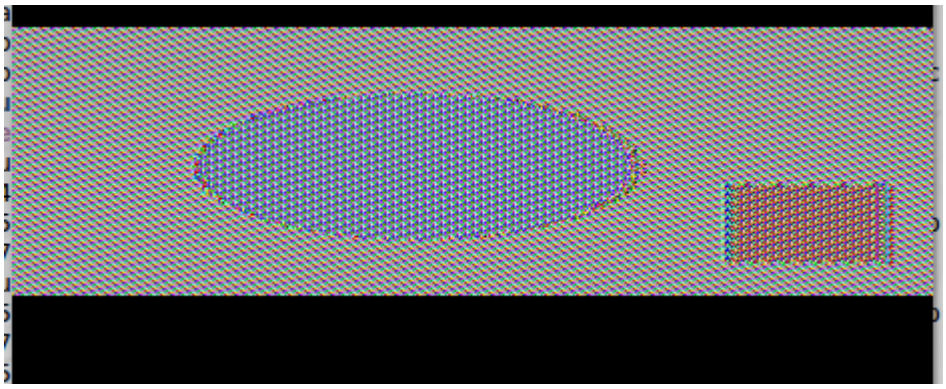



Used bf-cbc, aes-256-cbc and aes-128-cbc methods for encryption.

3) ECB

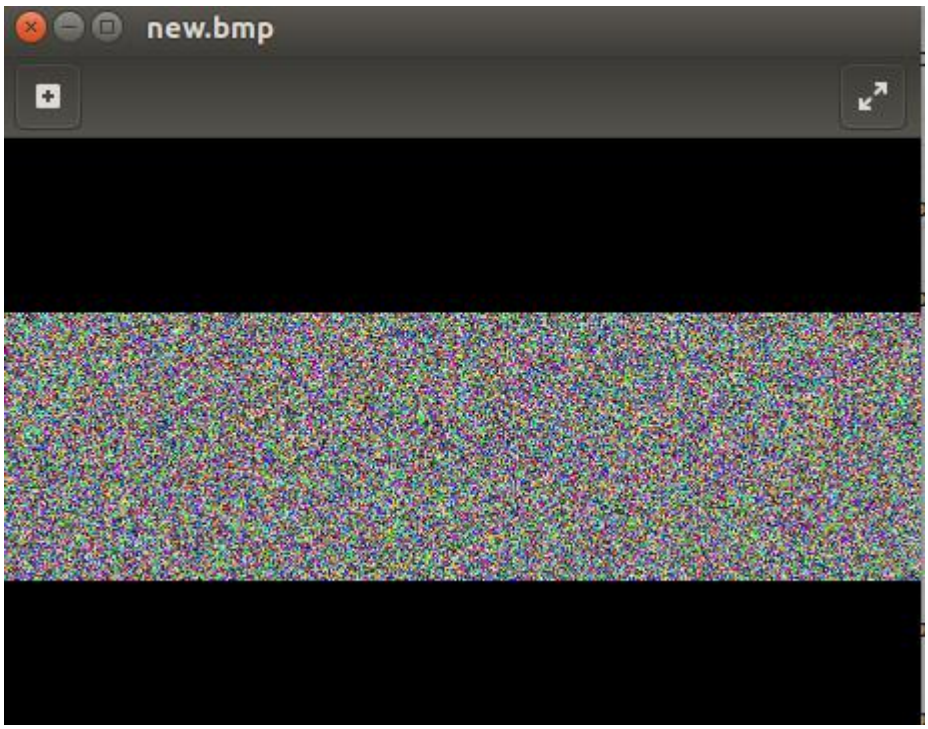
```
[04/05/2019 19:15]Mudelkadi@VM1:~$ openssl aes-128-ecb -in pic_original.bmp -out ecb.bmp -K
 00112233445566778889aabbccddeeff
```

```
[04/05/2019 19:17]Mudelkadi@VM1:~$ head -c 54 pic_original.bmp > header
[04/05/2019 19:19]Mudelkadi@VM1:~$ tail -c +55 ecb.bmp > body
[04/05/2019 19:19]Mudelkadi@VM1:~$ cat header body > new.bmp
[04/05/2019 19:19]Mudelkadi@VM1:~$ eog new.bmp
```



CBC

```
[04/05/2019 19:25]Mudelkadi@VM1:~$ openssl aes-128-cbc -in pic_original.bmp -out cbc.bmp -K
 00112233445566778889aabbccddeeff -iv 0102030405060708
[04/05/2019 19:25]Mudelkadi@VM1:~$ head -c 54 pic_original.bmp > header
[04/05/2019 19:25]Mudelkadi@VM1:~$ tail -c +55 cbc.bmp > body
[04/05/2019 19:26]Mudelkadi@VM1:~$ cat header body > new.bmp
[04/05/2019 19:26]Mudelkadi@VM1:~$ eog new.bmp
```




From the above outputs of ECB and CBC we can infer that ECB doesn't encrypt properly and lets us know the actual message whereas CBC gives granular output thus not revealing the actual message. This is because in ECB all the similar plaintexts gives similar ciphertexts whereas the

inputs in CBC are XOR of previous ciphertext and current plaintext and this happens canonically.

4)

a)

```
[04/05/2019 20:38]Mudelkadi@VM1:~$ openssl aes-128-cbc  -in f2.txt  -out f2_cbc.txt -K 0011
2233445566778889aabbccddeeff -iv 0102030405060708
[04/05/2019 20:38]Mudelkadi@VM1:~$ openssl aes-128-cbc  -in f3.txt  -out f3_cbc.txt -K 0011
2233445566778889aabbccddeeff -iv 0102030405060708
[04/05/2019 20:38]Mudelkadi@VM1:~$ openssl aes-128-ecb  -in f2.txt  -out f2_ecb.txt -K 0011
2233445566778889aabbccddeeff
[04/05/2019 20:39]Mudelkadi@VM1:~$ openssl aes-128-ecb  -in f3.txt  -out f3_ecb.txt -K 0011
2233445566778889aabbccddeeff
[04/05/2019 20:39]Mudelkadi@VM1:~$ openssl aes-128-cfb  -in f2.txt  -out f2_cfb.txt -K 0011
2233445566778889aabbccddeeff
iv undefined
[04/05/2019 20:39]Mudelkadi@VM1:~$ openssl aes-128-cfb  -in f2.txt  -out f2_cfb.txt -K 0011
2233445566778889aabbccddeeff -iv 0102030405060708
[04/05/2019 20:39]Mudelkadi@VM1:~$ openssl aes-128-cfb  -in f3.txt  -out f3_cfb.txt -K 0011
2233445566778889aabbccddeeff -iv 0102030405060708
[04/05/2019 20:40]Mudelkadi@VM1:~$ openssl aes-128-ofb  -in f2.txt  -out f2_ofb.txt -K 0011
2233445566778889aabbccddeeff -iv 0102030405060708
[04/05/2019 20:40]Mudelkadi@VM1:~$ openssl aes-128-ofb  -in f3.txt  -out f3_ofb.txt -K 0011
2233445566778889aabbccddeeff -iv 0102030405060708
```

```
-rw-rw-r-- 1 seed seed 16 Apr  5 20:38 f2_cbc.txt
-rw-rw-r-- 1 seed seed 10 Apr  5 20:39 f2_cfb.txt
-rw-rw-r-- 1 seed seed 16 Apr  5 20:39 f2_ecb.txt
-rw-rw-r-- 1 seed seed 10 Apr  5 20:40 f2_ofb.txt
-rw-rw-r-- 1 seed seed 10 Apr  5 20:07 f2.txt
```

```
-rw-rw-r-- 1 seed seed 32 Apr  5 20:38 f3_cbc.txt
-rw-rw-r-- 1 seed seed 16 Apr  5 20:40 f3_cfb.txt
-rw-rw-r-- 1 seed seed 32 Apr  5 20:39 f3_ecb.txt
-rw-rw-r-- 1 seed seed 16 Apr  5 20:40 f3_ofb.txt
-rw-rw-r-- 1 seed seed 16 Apr  5 20:09 f3.txt
```

We can infer that padding doesn't happen with CFB and OFB, whereas padding happens for CBC and ECB until it reaches next multiple of 16 bytes.

b)

```
[04/05/2019 20:29]Mudelkadi@VM1:~$ cat f1.txt
anula[04/05/2019 20:29]Mudelkadi@VM1:~$ cat f2.txt
akarshetty[04/05/2019 20:29]Mudelkadi@VM1:~$ cat f3.txt
padminiumesharju[04/05/2019 20:29]M$ hexdump -C f1_aes_decrypt.txt
$: command not found
[04/05/2019 20:30]Mudelkadi@VM1:~$ hexdump -C f1_aes_decrypt.txt
00000000  61 6e 75 6c 61 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |anula...........|
00000010
[04/05/2019 20:30]Mudelkadi@VM1:~$ hexdump -C f2_aes_decrypt.txt
00000000  61 6b 61 72 73 68 65 74  74 79 06 06 06 06 06 06  |akarshetty......|
00000010
[04/05/2019 20:30]Mudelkadi@VM1:~$ hexdump -C f3_aes_decrypt.txt
00000000  70 61 64 6d 69 6e 69 75  6d 65 73 68 61 72 6a 75  |padminiumesharju|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
[04/05/2019 20:30]Mudelkadi@VM1:~$
```

We can see that the extra padding happens until the total size reaches next multiple of 16.

5)

Encryption using **CBC.**

```
[04/05/2019 21:34]Mudelkadi@VM1:~$ openssl aes-128-cbc  -in task5.txt  -out task5_encry.txt
 -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

```
05 80 17 67 F4 45 7C 14 07 72 79 C1 03 4E 97 D3 A9 7D
E2 5E 6C 4D AF A7 E1 1C 81 49 17 61 A6 FC C8 97 A0 7B
F9 8D 51 28 7D DB 71 ED F4 C1 7A B3 D8 8C 82 E4 39 CC
3A E3 DD 94 4D 3C 98 FD 2E 8B 82 A6 CB 01 D2 41 6B 14
F6 0F 19 F3 4C 72 01 EE D8 44 AE 24 04 06 30 FA 53 D6
7E CB 52 B1 5E 0B E0 88 89 4B 13 6C 30 BD E8 6C 63 7D
B5 7B 5D 21 65 D2 3E 6C 13 06 14 C5 EF F8 D8 35 38 43
FF D4 F2 02 08 CD 76 56 0B 2B BC E1 FB 42 5A 0A 86 3B
```

Changed the 50th byte from 82 to 93.

```
[04/05/2019 21:38]Mudelkadi@VM1:~$ hexdump -C task5_decry.txt
00000000  49 66 20 74 68 69 73 20  69 73 20 74 6f 20 62 65  |If this is to be|
00000010  20 4c 69 76 65 72 70 6f  6f 6c e2 80 99 73 20 79  | Liverpool...s y|
00000020  65 61 72 2c 20 69 74 20  6d 61 79 20 62 65 20 72  |ear, it may be r|
00000030  65 73 69 6c 69 65 6e 63  65 20 61 6e 64 20 61 6e  |esilience and an|
00000040  20 61 64 6d 69 72 61 62  6c 65 20 61 62 69 6c 69  | admirable abili|
00000050  74 79 20 74 6f 20 62 6f  75 6e 63 65 20 62 61 63  |ty to bounce bac|
00000060  6b 20 66 72 6f 6d 20 73  65 74 62 61 63 6b 73 20  |k from setbacks |
```

Decryption before corrupting encrypted data.

```
[04/05/2019 21:35]Mudelkadi@VM1:~$ hexdump -C task5_decry.txt
00000000  49 66 20 74 68 69 73 20  69 73 20 74 6f 20 62 65  |If this is to be|
00000010  20 4c 69 76 65 72 70 6f  6f 6c e2 80 99 73 20 79  | Liverpool...s y|
00000020  65 61 72 2c 20 69 74 20  6d 61 79 20 62 65 20 72  |ear, it may be r|
00000030  a1 d9 58 4d f6 fb 5d 53  3d cf ab f1 32 2b 04 b4  |..XM..]S=...2+..|
00000040  20 61 6f 6d 69 72 61 62  6c 65 20 61 62 69 6c 69  | aomirable abili|
00000050  74 79 20 74 6f 20 62 6f  75 6e 63 65 20 62 61 63  |ty to bounce bac|
00000060  6b 20 66 72 6f 6d 20 73  65 74 62 61 63 6b 73 20  |k from setbacks |
```

Decryption after corrupting encrypted data.

We can infer that after corrupting the 50th byte data, two blocks(4th and 5th ) gets corrupted while decrypting.

**ECB.**

```
[04/05/2019 21:55]Mudelkadi@VM1:~$ hexdump -C task5_decry.txt
00000000  49 66 20 74 68 69 73 20  69 73 20 74 6f 20 62 65  |If this is to be|
00000010  20 4c 69 76 65 72 70 6f  6f 6c e2 80 99 73 20 79  | Liverpool...s y|
00000020  1e 92 ce c2 fa 8e 64 6d  f6 af ad c6 4c 77 c9 14  |......dm....Lw..|
00000030  65 73 69 6c 69 65 6e 63  65 20 61 6e 64 20 61 6e  |esilience and an|
00000040  20 61 64 6d 69 72 61 62  6c 65 20 61 62 69 6c 69  | admirable abili|
```

```
[04/05/2019 21:56]Mudelkadi@VM1:~$ hexdump -C task5_decry.txt
00000000  49 66 20 74 68 69 73 20  69 73 20 74 6f 20 62 65  |If this is to be|
00000010  20 4c 69 76 65 72 70 6f  6f 6c e2 80 99 73 20 79  | Liverpool...s y|
00000020  65 61 72 2c 20 69 74 20  6d 61 79 20 62 65 20 72  |ear, it may be r|
00000030  65 73 69 6c 69 65 6e 63  65 20 61 6e 64 20 61 6e  |esilience and an|
00000040  20 61 64 6d 69 72 61 62  6c 65 20 61 62 69 6c 69  | admirable abili|
```

We can infer that after corrupting the 50th byte data, one block(3d ) gets corrupted while decrypting.

**CFB.**

```
[04/05/2019 22:02]Mudelkadi@VM1:~$ hexdump -C task5_decry.txt
00000000  49 66 20 74 68 69 73 20  69 73 20 74 6f 20 62 65  |If this is to be|
00000010  20 4c 69 76 65 72 70 6f  6f 6c e2 80 99 73 20 79  | Liverpool...s y|
00000020  65 61 72 2c 20 69 74 20  6d 61 79 20 62 65 20 72  |ear, it may be r|
00000030  65 73 69 6c 69 65 6e 63  65 20 61 6e 64 20 61 6e  |esilience and an|
00000040  20 61 64 6d 69 72 61 62  6c 65 20 61 62 69 6c 69  | admirable abili|
00000050  74 79 20 74 6f 20 62 6f  75 6e 63 65 20 62 61 63  |ty to bounce bac|
00000060  6b 20 66 72 6f 6d 20 73  65 74 62 61 63 6b 73 20  |k from setbacks |
```

```
[04/05/2019 22:05]Mudelkadi@VM1:~$ hexdump -C task5_decry.txt
00000000  49 66 20 74 68 69 73 20  69 73 20 74 6f 20 62 65  |If this is to be|
00000010  20 4c 69 76 65 72 70 6f  6f 6c e2 80 99 73 20 79  | Liverpool...s y|
00000020  65 61 72 2c 20 69 74 20  6d 61 79 20 62 65 20 72  |ear, it may be r|
00000030  65 73 62 6c 69 65 6e 63  65 20 61 6e 64 20 61 6e  |esblience and an|
00000040  a3 a7 94 7b 35 a4 64 40  17 f6 1e 6b eb ee 3a bb  |...{5.d@...k..:.|
00000050  74 79 20 74 6f 20 62 6f  75 6e 63 65 20 62 61 63  |ty to bounce bac|
00000060  6b 20 66 72 6f 6d 20 73  65 74 62 61 63 6b 73 20  |k from setbacks |
```

We can infer that after corrupting the 50th byte data, two blocks(4th and 5th ) gets corrupted while decrypting.

**OFB.**

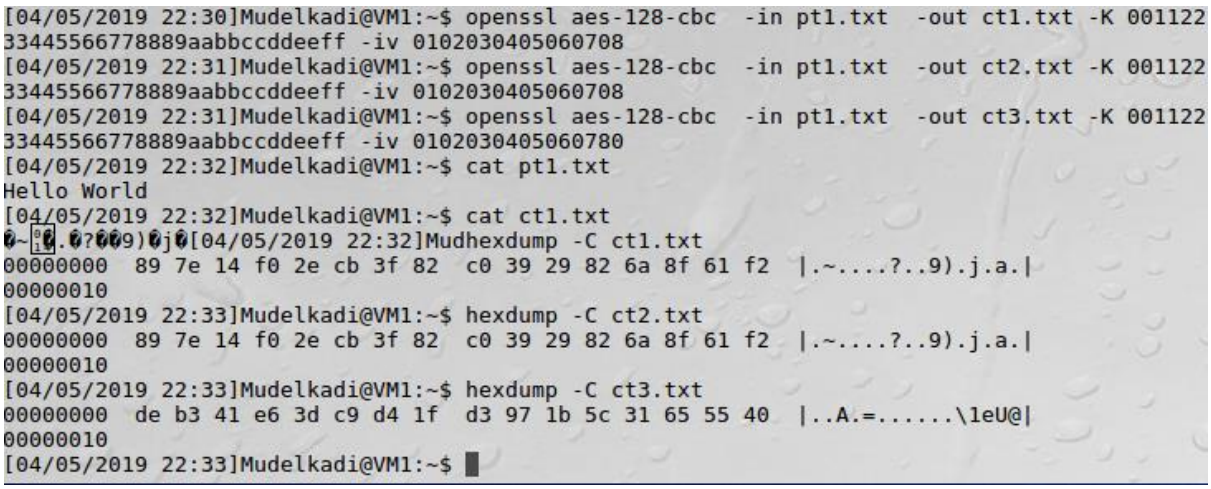
```
[04/05/2019 22:08]Mudelkadi@VM1:~$ hexdump -C task5_decry.txt
00000000  49 66 20 74 68 69 73 20  69 73 20 74 6f 20 62 65  |If this is to be|
00000010  20 4c 69 76 65 72 70 6f  6f 6c e2 80 99 73 20 79  | Liverpool...s y|
00000020  65 61 72 2c 20 69 74 20  6d 61 79 20 62 65 20 72  |ear, it may be r|
00000030  65 73 69 6c 69 65 6e 63  65 20 61 6e 64 20 61 6e  |esilience and an|
00000040  20 61 64 6d 69 72 61 62  6c 65 20 61 62 69 6c 69  | admirable abili|
00000050  74 79 20 74 6f 20 62 6f  75 6e 63 65 20 62 61 63  |ty to bounce bac|
00000060  6b 20 66 72 6f 6d 20 73  65 74 62 61 63 6b 73 20  |k from setbacks |
```

```
[04/05/2019 22:10]Mudelkadi@VM1:~$ hexdump -C task5_decry.txt
00000000  49 66 20 74 68 69 73 20  69 73 20 74 6f 20 62 65  |If this is to be|
00000010  20 4c 69 76 65 72 70 6f  6f 6c e2 80 99 73 20 79  | Liverpool...s y|
00000020  65 61 72 2c 20 69 74 20  6b 61 79 20 62 65 20 72  |ear, it kay be r|
00000030  65 73 69 6c 69 65 6e 63  65 20 61 6e 64 20 61 6e  |esilience and an|
00000040  20 61 64 6d 69 72 61 62  6c 65 20 61 62 69 6c 69  | admirable abili|
00000050  74 79 20 74 6f 20 62 6f  75 6e 63 65 20 62 61 63  |ty to bounce bac|
00000060  6b 20 66 72 6f 6d 20 73  65 74 62 61 63 6b 73 20  |k from setbacks |
```

We can infer that after corrupting the 50th byte data, one block(3d ) gets corrupted while decrypting.

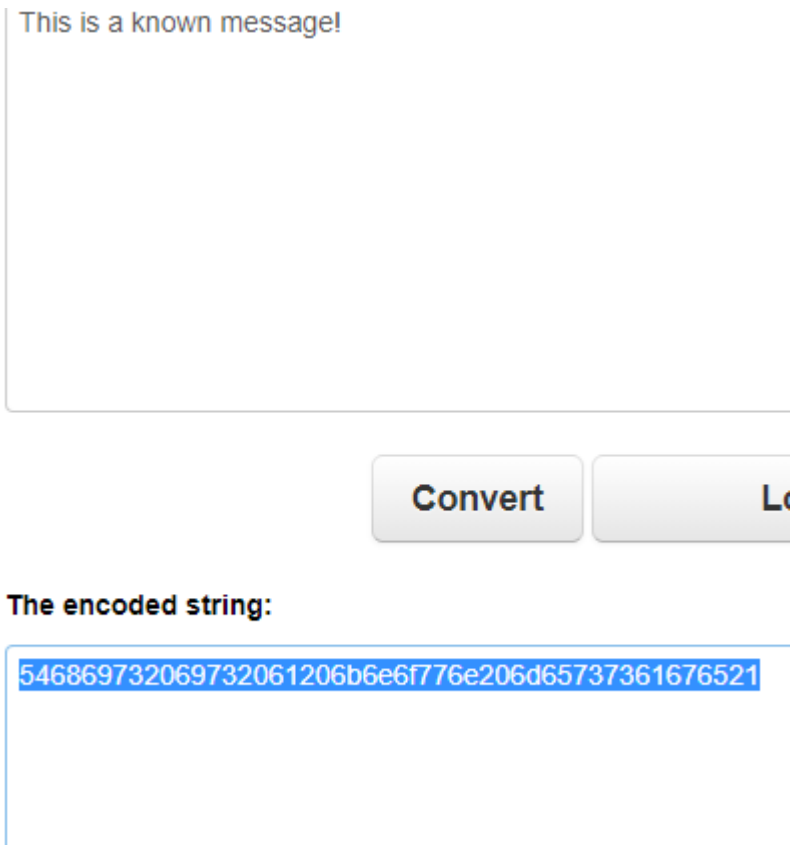
Therefore, there is 2 blocks changes in CBC and CFB while decrypting the corrupted 50th byte because it uses previous block for encryption but for OFB and ECB there is only 1 block change because it doesn't use previous block for encryption.

6.1)

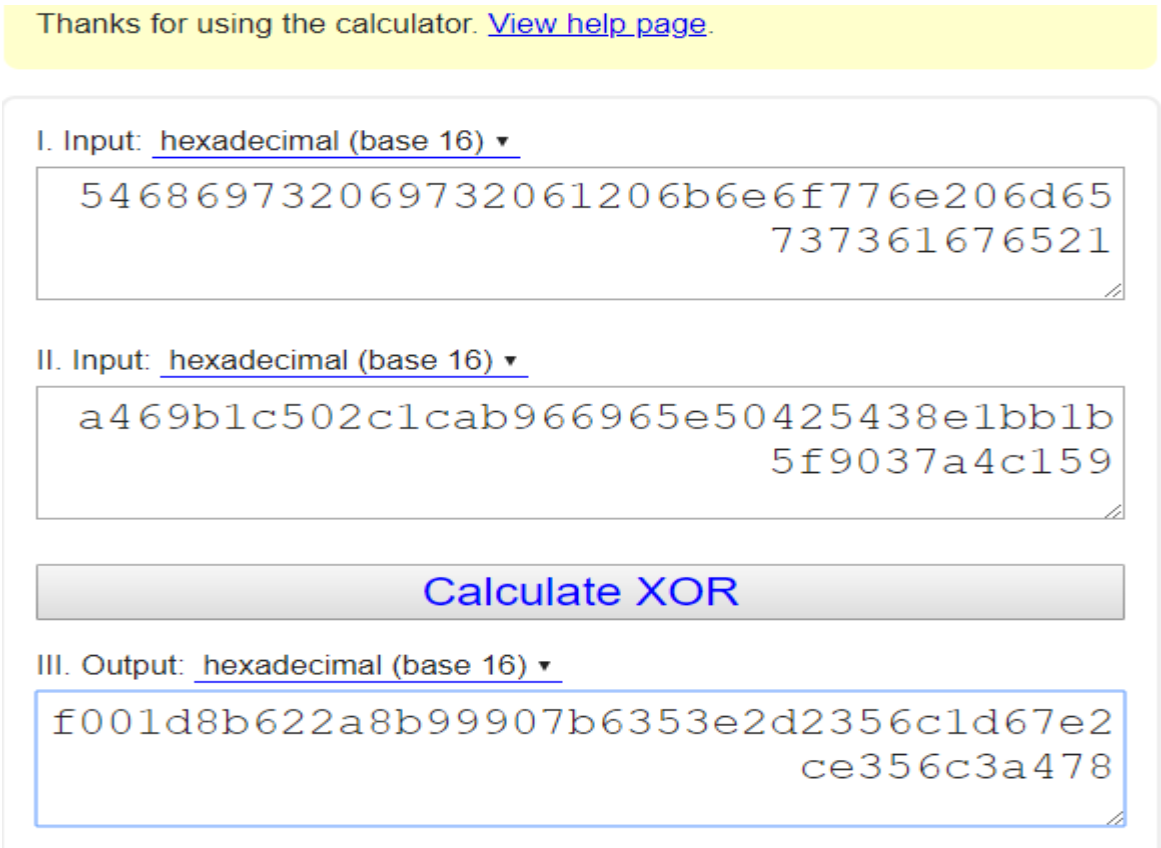



Pt1.txt is my plaintext. Ct1.txt and ct2.txt is ciphertext with same IV. Ct3.txt is ciphertext with different IV with just last 2 digits of IV interchanged. We can observe that ct1.txt and ct2.txt are same and it becomes easier for attacker to assume or crack the plaintext but when we use a different IV it gives totally a different ciphertext thus difficult to crack the plaintext.

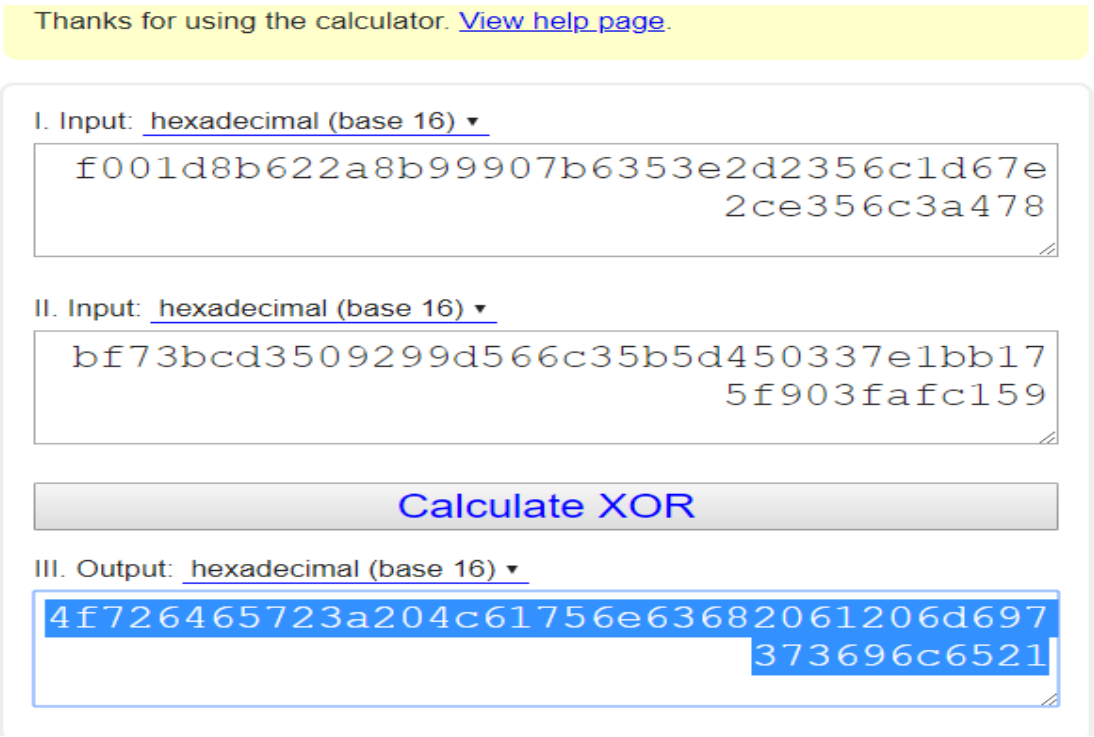
6.2)



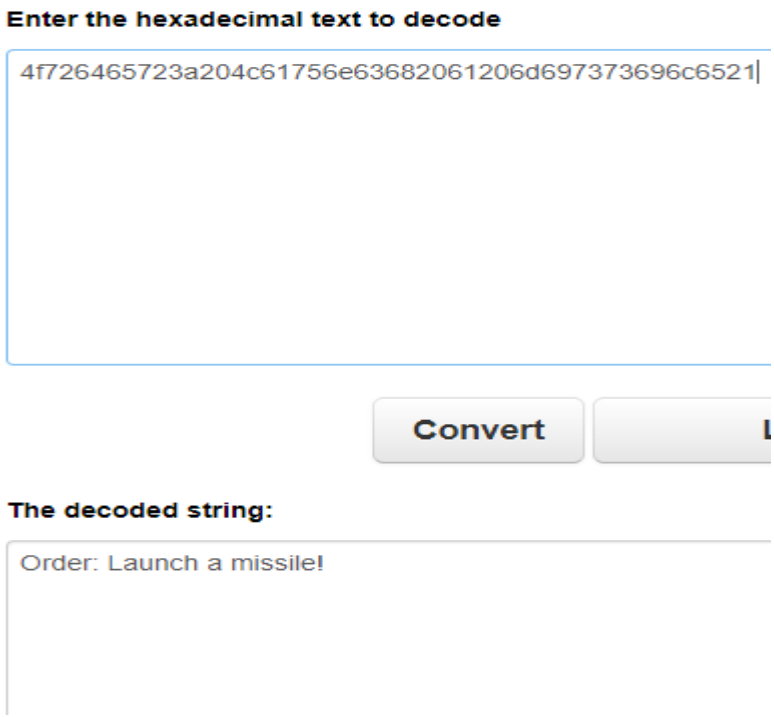

Got hexadecimal value for the plaintext 1



Got XOR value for hexadecimal of plaintext 1 and ciphertext 1.




Got XOR value for previously calculated result and ciphertext 2.



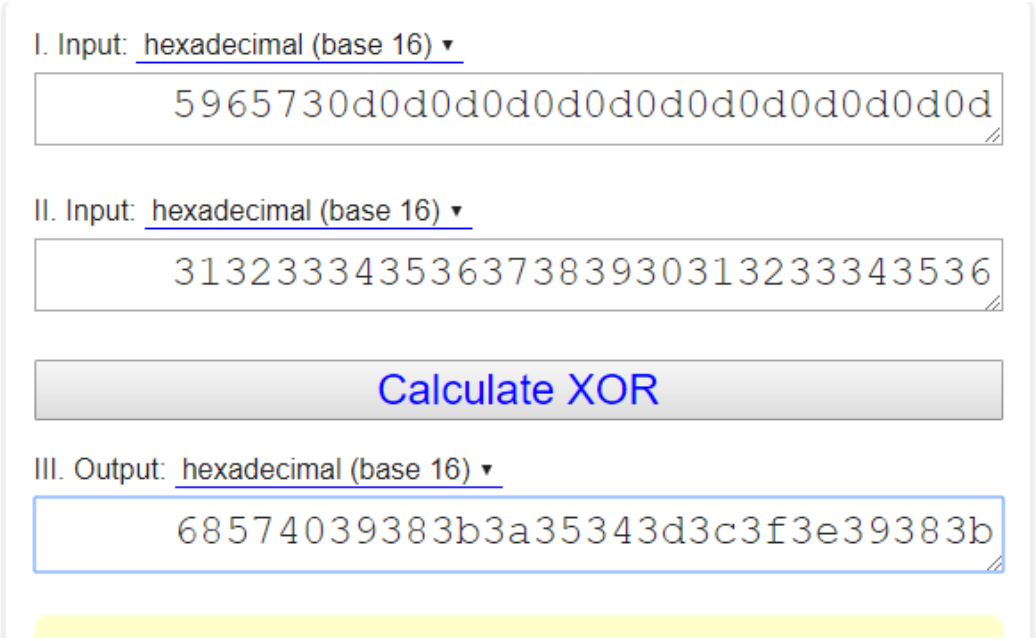

Got the plaintext 2 string value.

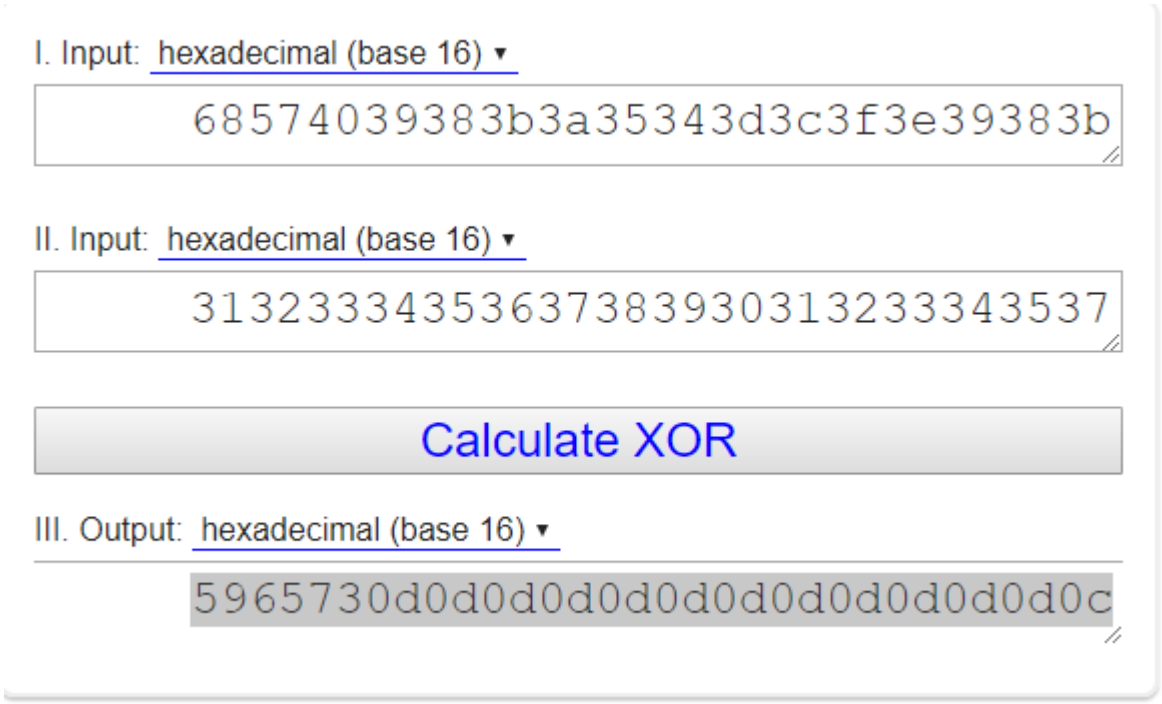
**Observation:** As the IV is same and the result is of XOR of plaintext and ciphertext of one string will be same for all the remaining strings. Thus, by getting the XOR value of plaintext and ciphertext of first string, the result got can be XORed with 2nd ciphertext to get the next plain text. The above pictures demonstrate how the result is obtained.

For CFB mode of encryption, the block encryption is XORed with the plaintext and sent as input to next string encryption and thus we need to know the next plaintext which will be XORed to get the ciphertext 2. Therefore we cannot obtain the plaintext in CFB mode.

6.3)

```
[04/08/2019 20:49]Mudelkadi@VM1:~$ hexdump -C samp_decry.txt
00000000  59 65 73 0d 0d 0d 0d 0d  0d 0d 0d 0d 0d 0d 0d 0d  |Yes.........
....|
00000010
```






```
[04/08/2019 20:55]Mudelkadi@VM1:~$ hexdump -C samp_decry.txt
00000000  59 65 73 0d 0d 0d 0d 0d  0d 0d 0d 0d 0d 0d 0d 0c  |Yes.........
....|
00000010
[04/08/2019 20:55]Mudelkadi@VM1:~$ openssl aes-128-cbc -in samp_decry.txt
 -out fin_encry.txt -K 00112233445566778899aabbccddeeff -iv  313233343536
37383930313233343537
[04/08/2019 20:56]Mudelkadi@VM1:~$ hexdump -C fin_encry.txt
00000000  be f6 55 65 57 2c ce e2  a9 f9 55 31 54 ed 94 98  |..UeW,....U1
T...|
00000010  34 02 de 3f 0d d1 6c e7  89 e5 47 57 79 ac a4 05  |4..?..l...GW
y...|
00000020
[04/08/2019 20:56]Mudelkadi@VM1:~$
```

**Answer:** First I got the decrypted value of "Yes" with padding as you can see in first picture. To find what is going inside the block for encryption, we can do this by XORing hex value of decrypted "Yes" with IV1 and IV2. The result is encrypted with IV2 for cancelling out IV2. Encrypting with the same key used and IV2 gives us the same ciphertext as used by original text. Thus using predictable IVs are not safe.

For info: I changed the same file samp.decry to last letter to c as we cannot just paste a string as we need hex value, therefore I used ghex command for changing.

7)

Code:

```
from Crypto.Cipher import *

plaintext = "This is a top secret."
plaintext+=chr(11) * 11 #should be padded to fit into whole block"
ciphertext="764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2"
IV="aabbccddeeff00998877665544332211"
decode_IV=IV.decode("hex")
f=open("words.txt","r")
l1=f.read().splitlines() #list of words in words.txt
for key in l1:
        word=key
        if len(word)>16: #as key doesn't exceed length 16
                continue
        if len(word)%16 !=0:
                length=len(word)
                while length%16!=0:
                        word += "#" #add till the key becomes the size of
16
                        length+=1
        new_ciphertext = AES.new(word,AES.MODE_CBC,decode_IV).encrypt(plaintext)
        nct_hex=str(new_ciphertext.encode("hex"))
        if (nct_hex==ciphertext):
                print('Key = '+key)
```

```
[04/08/2019 23:02]Mudelkadi@VM1:~$ sudo python progcrypto.py
Key = Syracuse
[04/08/2019 23:05]Mudelkadi@VM1:~$
```

Used words.txt for finding the actual key. Have converted values into HEX wherever needed. If the words length is greater than 16 they are not checked (as mentioned in the question that the key doesn't exceed length 16).