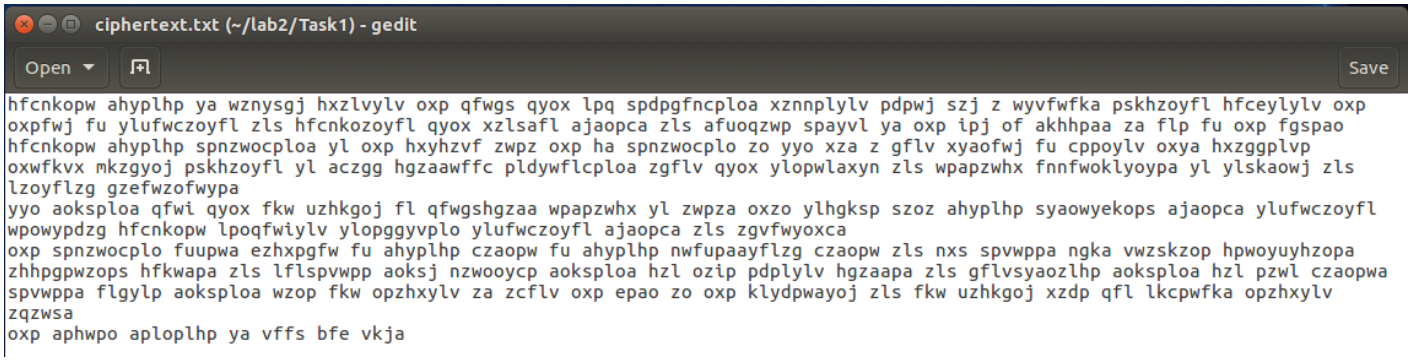
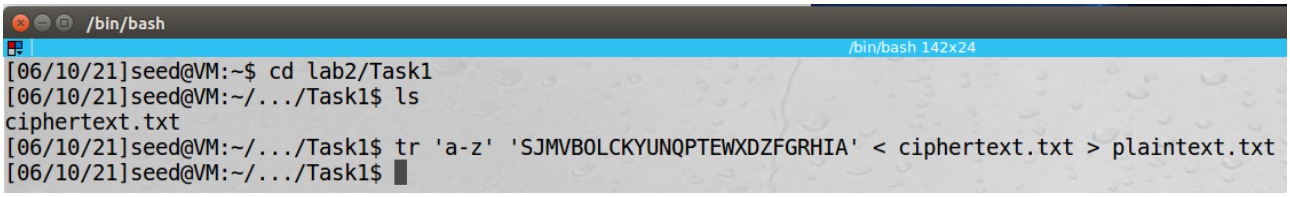
Task 1: Frequency Analysis

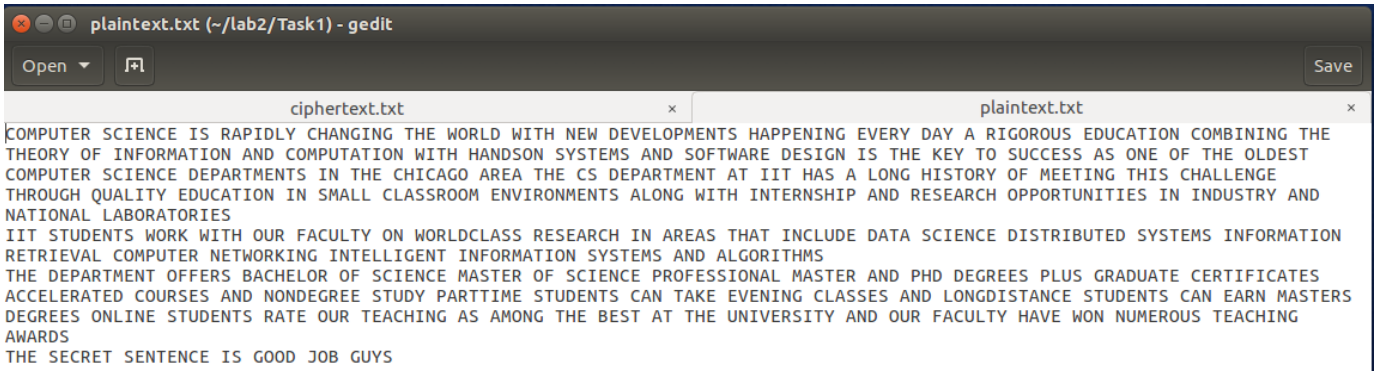
The text file content of ciphertext.txt



In order to decrypt the ciphertext, the following command should be performed:



Plaintext



Task 2: Encryption using Different Ciphers and Modes

Let's create 3 different files with the same content of "Hello World"

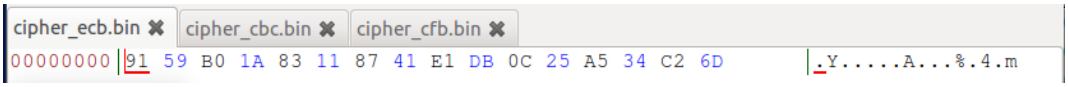
```
[06/10/21]seed@VM:~/.../Task2$ echo "Hello World" > plain_ecb.txt
[06/10/21]seed@VM:~/.../Task2$ echo "Hello World" > plain_cbc.txt
[06/10/21]seed@VM:~/.../Task2$ echo "Hello World" > plain_cfb.txt
```

- Let's start with cipher -aes-128 with 3 modes: ECB, CBC, CFB

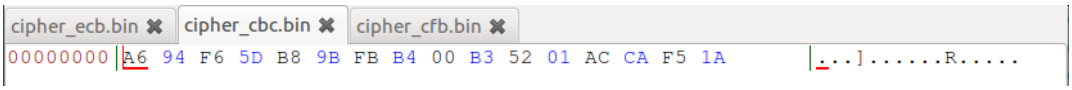
```
[06/10/21]seed@VM:~/.../Task2$ openssl enc -aes-128-ecb -e -in plain_ecb.txt -out cipher_ecb.bin -K 000102030405060708090aabbccddeeff -iv 0a0b0c0d0e0f010203040506070809
warning: iv not use by this cipher
[06/10/21]seed@VM:~/.../Task2$ openssl enc -aes-128-cbc -e -in plain_cbc.txt -out cipher_cbc.bin -K 000102030405060708090aabbccddeeff -iv 0a0b0c0d0e0f010203040506070809
[06/10/21]seed@VM:~/.../Task2$ openssl enc -aes-128-cfb -e -in plain_cfb.txt -out cipher_cfb.bin -K 000102030405060708090aabbccddeeff -iv 0a0b0c0d0e0f010203040506070809
```

Using “Bless”, we can take a look inside “cipher_ecb.bin”, “cipher_cbc.bin”, and “cipher_cfb.bin”

- ECB

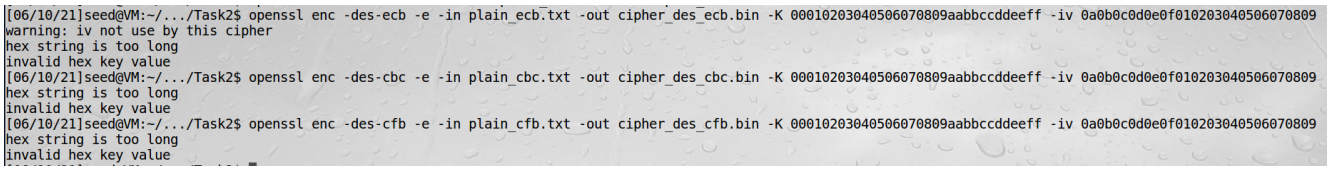


- CBC



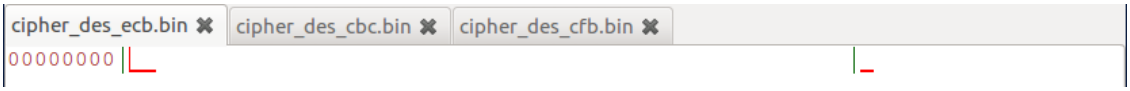
- CFB

- Next, we will take a look at cipher -des with 3 modes: ECB, CBC, CFB

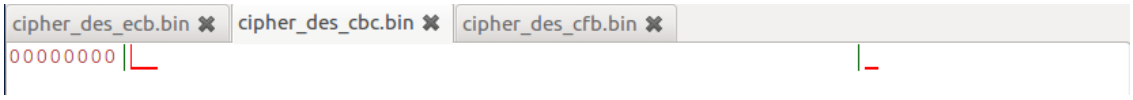


Again, we can take a look inside” cipher_des_ecb.bin”, “cipher_des_cbc.bin”, and “cipher_des_cfb.bin” using “Bless”

- ECB



- CBC



- CFB

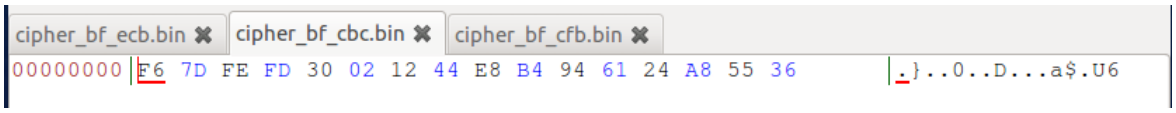
- Finally, we will take a look at cipher -bf with 3 modes: ECB, CBC, CFB

```
[06/10/21]seed@VM:~/.../Task2$ openssl enc -bf-ecb -e -in plain_ecb.txt -out cipher_bf_ecb.bin -K 00010203040506070809aabbccddeeff -iv 0a0b0c0d0e0f010203040506070809
warning: iv not use by this cipher
[06/10/21]seed@VM:~/.../Task2$ openssl enc -bf-cbc -e -in plain_cbc.txt -out cipher_bf_cbc.bin -K 00010203040506070809aabbccddeeff -iv 0a0b0c0d0e0f010203040506070809
[06/10/21]seed@VM:~/.../Task2$ openssl enc -bf-cfb -e -in plain_cfb.txt -out cipher_bf_cfb.bin -K 00010203040506070809aabbccddeeff -iv 0a0b0c0d0e0f010203040506070809
```

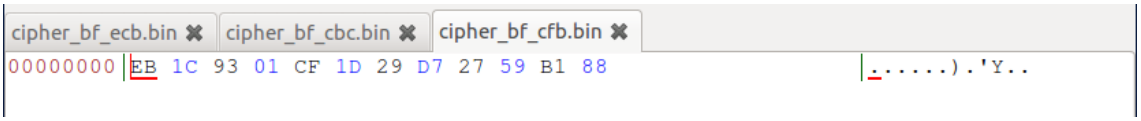
Again, we can take a look inside" cipher_bf_ecb.bin", "cipher_bf_cbc.bin", and "cipher_bf_cfb.bin" using "Bless"

- ECB

- CBC

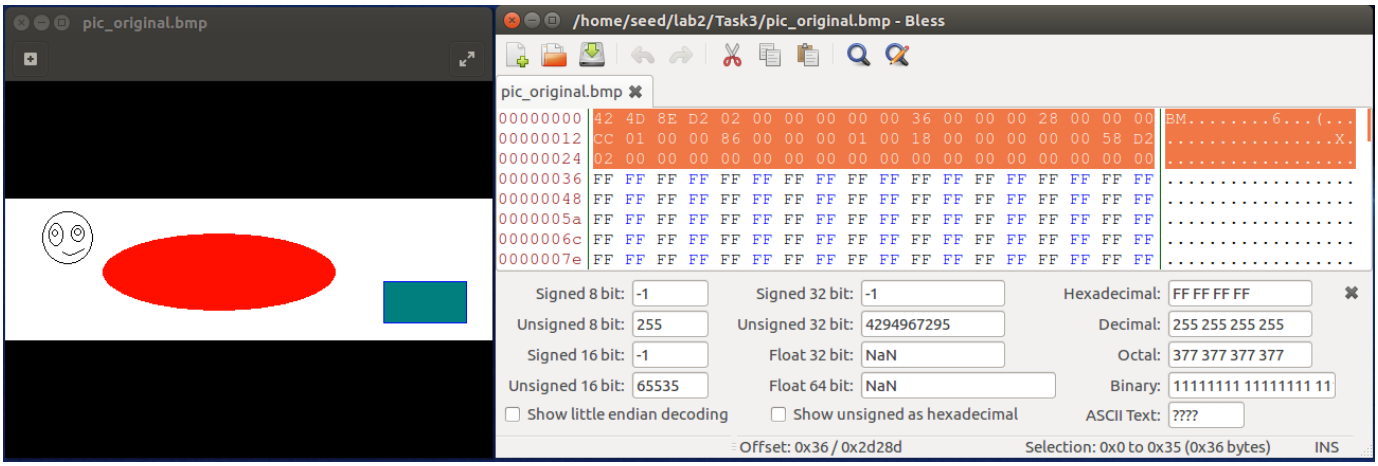


- CFB



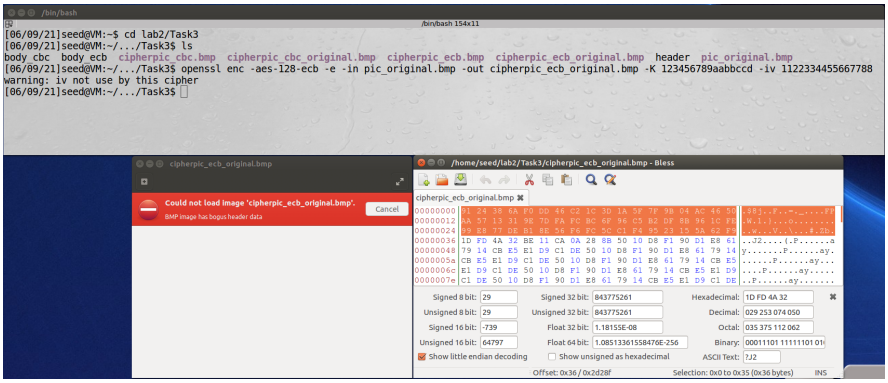
Task 3: Encryption Mode – ECB vs. CBC

The picture below shows the original content of "pic_original.bmp" downloaded from Blackboard. Using Bless, we can see the hex. The highlighted part of the hex shows the first 54 bytes that compose **header** section for both "cbc" and "ebc" modes

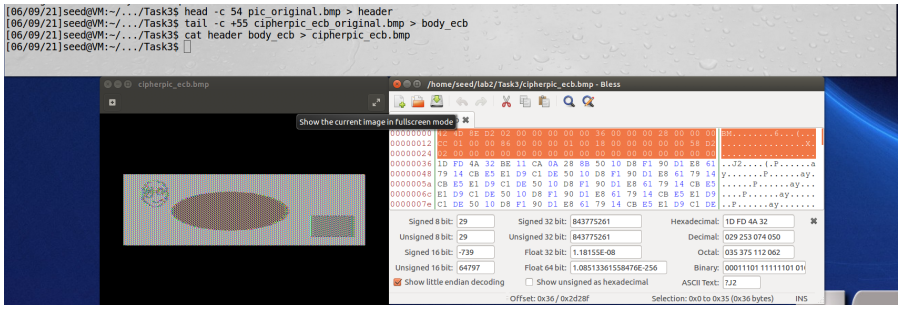


ECB

When we encrypt “pic_original.bmp” using **-aes-128-ecb**, we create a file called “cipherpic_ecb_original.bmp” When we try to open the file, we get the error message shown below. To solve the problem, the highlighted part of the hex file of “cipherpic_ecb_original.bmp” needs to be replaced with **header** and the tail after the 54th byte should be kept the same.

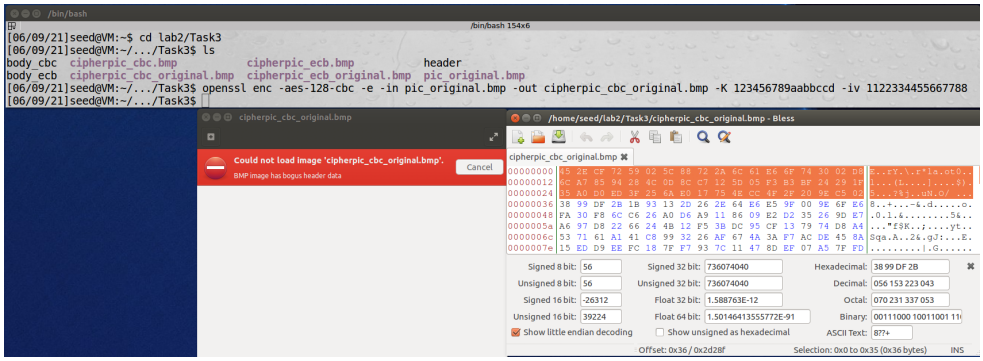


After some manipulations (using the code below), we create a new image that contains a **header** from original “pic_original.bmp” and **body_ecb** (the content of “cipherpic_ecb_original.bmp” that goes from offset 55 to the end of the file), thus resulting with a new file called “**cipherpic_ecb.bmp**”

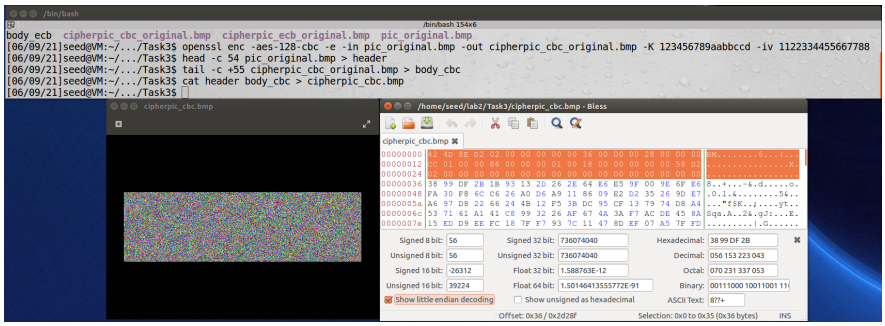


CBC

When we encrypt “pic_original.bmp” using **-aes-128-cbc**, we create a file called “cipherpic_cbc_original.bmp” When we try to open the file, we get the error message shown below. To solve the problem, the highlighted part of the hex file of “cipherpic_ecb_original.bmp” needs to be replaced with the already created **header** and the tail after the 54th byte should be kept the same.

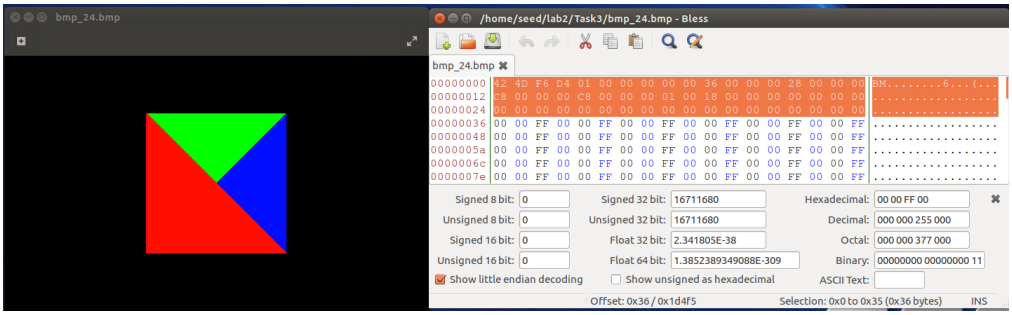


After some manipulations (using the code below), we modify **header** from original "pic_original.bmp" and create **body_cbc** (the content of "cipherpic_cbc_original.bmp" that goes from offset 55 to the end of the file), thus resulting with a new file called "**cipherpic_cbc.bmp**"



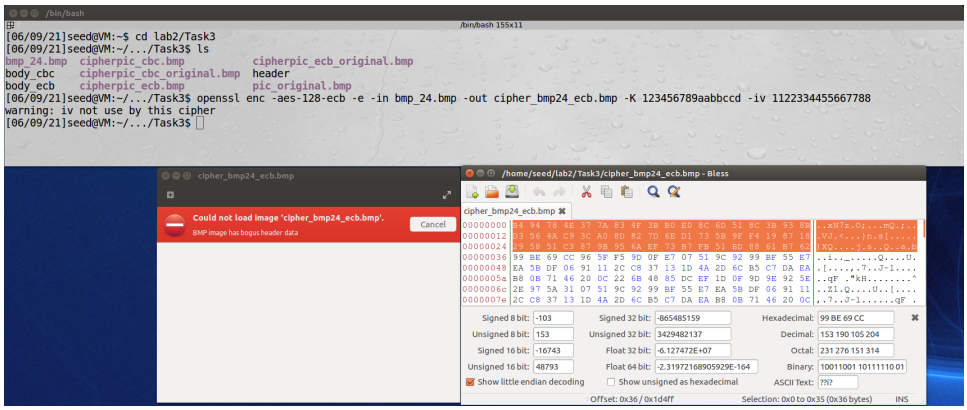
New Image

The picture below shows the original content of "bmp_24.bmp" found on the Internet. Using Bless, we can see the hex. The highlighted part of the hex shows the first 54 bytes that compose **header** section for both "cbc" and "ebc" modes

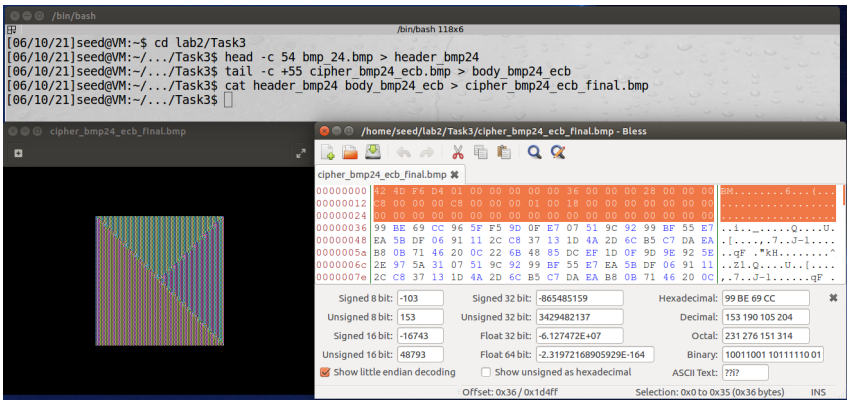


ECB

When we encrypt "bmp_24.bmp" using **-aes-128-ecb**, we create a file called "cipher_bmp24_ecb.bmp" When we try to open the file, we get the error message shown below. To solve the problem, the highlighted part of the hex file of "cipher_bmp24_ecb.bmp" needs to be replaced with **header_bmp24** and the tail after the 54th byte should be kept the same.

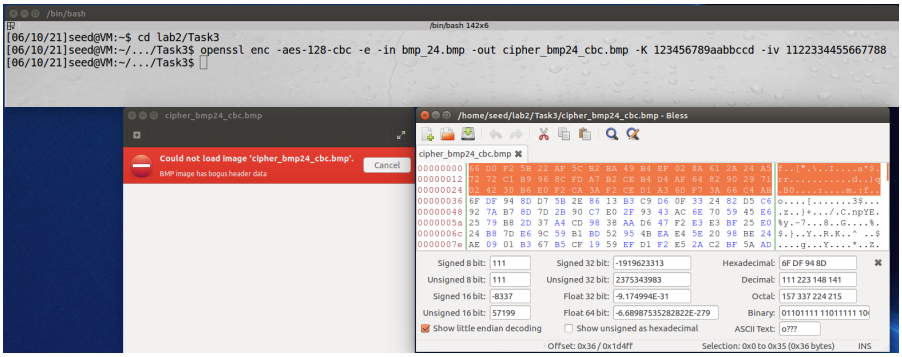


After some manipulations (using the code below), we create a new image that contains a **header_bmp24** from original “bmp_24.bmp” and **body_bmp24_ecb** (the content of “cipher_bmp24_ecb.bmp” that goes from offset 55 to the end of the file), thus resulting with a new file called “**cipher_bmp24_ecb_final.bmp**”

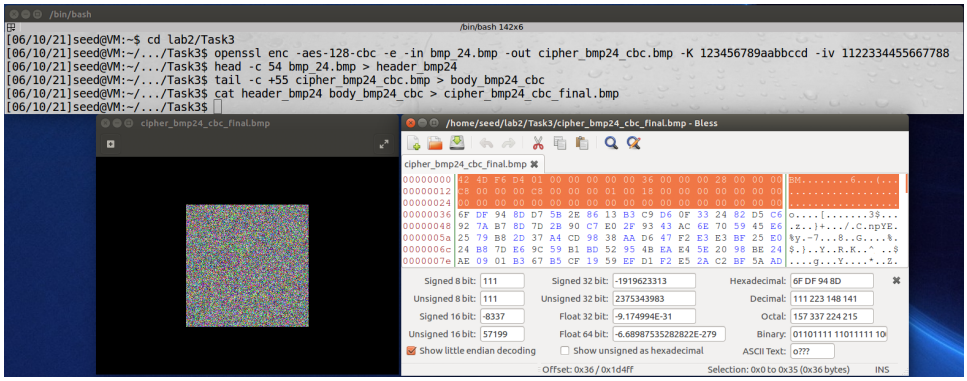


CBC

When we encrypt “bmp_24.bmp” using **-aes-128-cbc**, we create a file called “cipher_bmp24_cbc.bmp” When we try to open the file, we get the error message shown below. To solve the problem, the highlighted part of the hex file of “cipher_bmp24_cbc.bmp” needs to be replaced with already created **header_bmp24** and the tail after the 54th byte should be kept the same.



After some manipulations (using the code below), we modify **header_bmp24** from original “bmp_24.bmp” and create **body_bmp24_cbc** (the content of “cipher_bmp24_cbc.bmp” that goes from offset 55 to the end of the file), thus resulting with a new file called “**cipher_bmp24_cbc_final.bmp**”

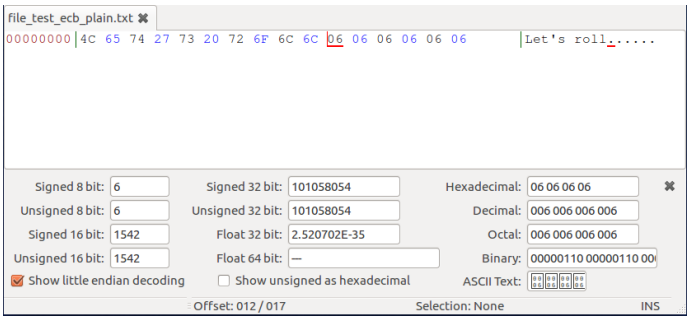


Task 4: Padding

1) Use ECB, CBC, CFB, and OFB modes to encrypt a file (you can pick any cipher). Please report which modes have paddings and which ones do not. For those that do not need paddings, please explain why.

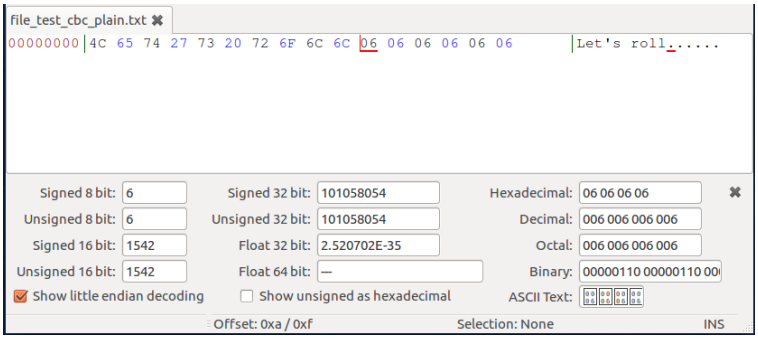
- ECB uses padding (5-bytes of "06" are added)

```
[06/10/21]seed@VM:~/.../Task4$ echo -n "Let's roll" > file_test_ciphers.txt
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-ecb -e -in file_test_ciphers.txt -out file_test_ecb.txt
enter aes-128-ecb encryption password:
Verifying - enter aes-128-ecb encryption password:
[06/10/21]seed@VM:~/.../Task4$ cat file_test_ecb.txt
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-ecb -d -nopad -in file_test_ecb.txt -out file_test_ecb_plain.txt
enter aes-128-ecb decryption password:
[06/10/21]seed@VM:~/.../Task4$ cat file_test_ecb_plain.txt
Let's roll▒▒▒▒▒▒▒06/10/21]seed@VM:~/.../Task4$ hexdump -C file_test_ecb_plain.txt
00000000  4c 65 74 27 73 20 72 6f  6c 6c 06 06 06 06 06 06  |Let's roll......|
00000010
[06/10/21]seed@VM:~/.../Task4$ xxd file_test_ecb_plain.txt
00000000: 4c65 7427 7320 726f 6c6c 0606 0606 0606  Let's roll......
```



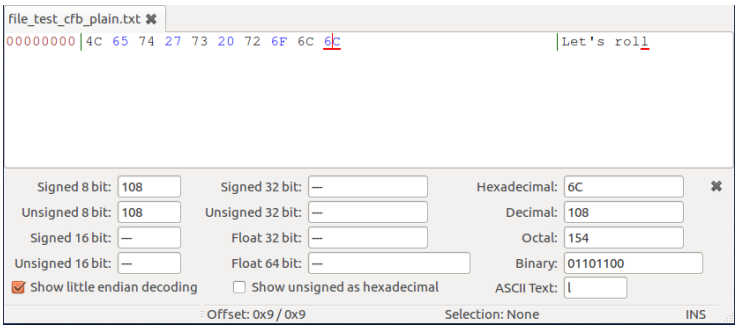
- CBC uses padding (5-bytes of "06" are added)

```
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -e -in file_test_ciphers.txt -out file_test_cbc.txt
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
[06/10/21]seed@VM:~/.../Task4$ cat file_test_cbc.txt
▒▒▒▒▒▒K▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒[06/10/21]se
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -d -nopad -in file_test_cbc.txt -out file_test_cbc_plain.txt
enter aes-128-cbc decryption password:
[06/10/21]seed@VM:~/.../Task4$ cat file_test_cbc_plain.txt
Let's roll▒▒▒▒▒▒▒06/10/21]seed@
[06/10/21]seed@VM:~/.../Task4$ hexdump -C filr_test_cbc_plain.txt
hexdump: filr_test_cbc_plain.txt: No such file or directory
[06/10/21]seed@VM:~/.../Task4$ hexdump -C file_test_cbc_plain.txt
00000000  4c 65 74 27 73 20 72 6f  6c 6c 06 06 06 06 06 06  |Let's roll......|
00000010
[06/10/21]seed@VM:~/.../Task4$ xxd file_test_cbc_plain.txt
00000000: 4c65 7427 7320 726f 6c6c 0606 0606 0606  Let's roll......
```



- CFB

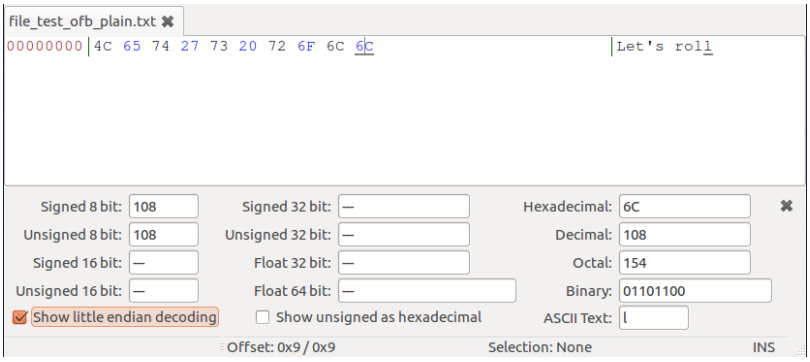
```
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-cfb -e -in file_test_ciphers.txt -out file_test_cfb.txt
enter aes-128-cfb encryption password:
Verifying - enter aes-128-cfb encryption password:
[06/10/21]seed@VM:~/.../Task4$ cat file_test_cfb.txt
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-cfb -d -nopad -in file_test_cfb.txt -out file_test_cfb_plain.txt
enter aes-128-cfb decryption password:
[06/10/21]seed@VM:~/.../Task4$ hexdump -C file_test_cfb_plain.txt
00000000  4c 65 74 27 73 20 72 6f  6c 6c                    |Let's roll|
0000000a
[06/10/21]seed@VM:~/.../Task4$ xxd file_test_cfb_plain.txt
00000000: 4c65 7427 7320 726f 6c6c                Let's roll
```



As it seen from code and decryption results from above, CFB mode does not perform padding

- OFB

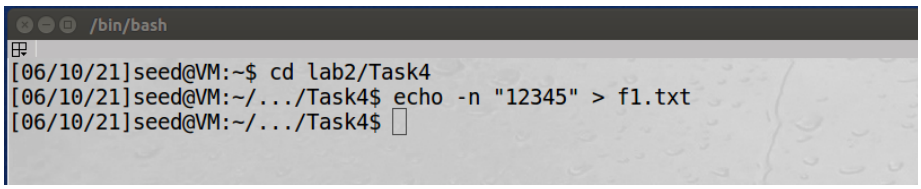
```
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-ofb -e -in file_test_ciphers.txt -out file_test_ofb.txt
enter aes-128-ofb encryption password:
Verifying - enter aes-128-ofb encryption password:
[06/10/21]seed@VM:~/.../Task4$ cat file_test_ofb.txt
Salted__ ▯▯▯▯y)▯▯▯▯wY)▯▯[06/10/2
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-ofb -d -nopad -in file_test_ofb.txt -out file_test_ofb_plain.txt
enter aes-128-ofb decryption password:
[06/10/21]seed@VM:~/.../Task4$ hexdump -C file_test_ofb_plain.txt
00000000  4c 65 74 27 73 20 72 6f  6c 6c                    |Let's roll|
0000000a
[06/10/21]seed@VM:~/.../Task4$ xxd file_test_ofb_plain.txt
00000000: 4c65 7427 7320 726f 6c6c                Let's roll
```



As it seen from code and decryption results from above, OFB mode also does not perform padding

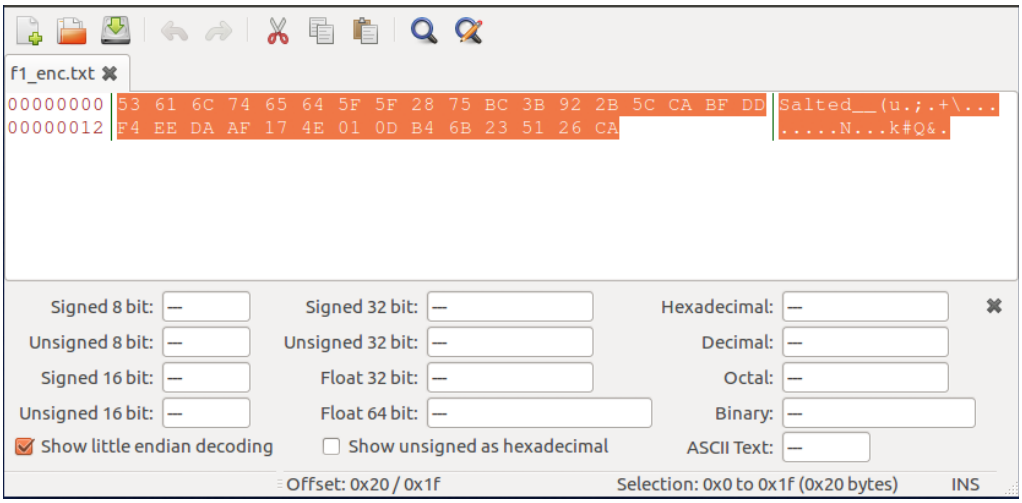
Explanation: CFB and OFB do not need padding because their ciphertexts will have the same size as the plaintexts

2) Let's create a file f1.txt that contains 5 bytes and has a content of "12345"



Next, we encrypt the message in a content file named "f1_enc.txt". The encrypted size of the data is 32 bytes.

```
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -e -in f1.txt -out f1_enc.txt
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
```



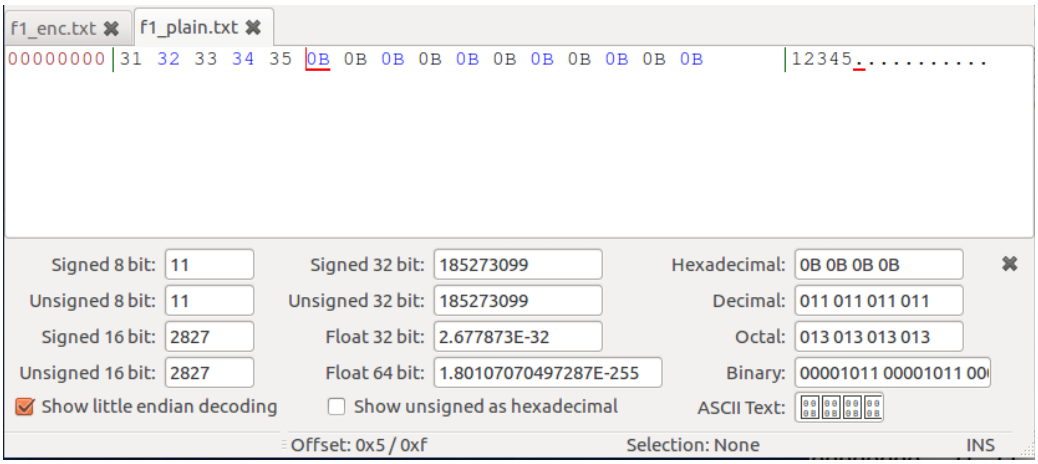
After that, we create an actual file “f1_enc.txt” and decrypt it to “f1_plain.txt”, using -nopad option in order to keep padding in the decryption message

```
[06/10/21]seed@VM:~/.../Task4$ cat f1_enc.txt
k#Q&[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -d -nopad -in f1_enc.txt -out f1_plain.txt
enter aes-128-cbc decryption password:
```

Finally, we create “f1_plain.txt” and perform hexdump to see what paddings are added to the file

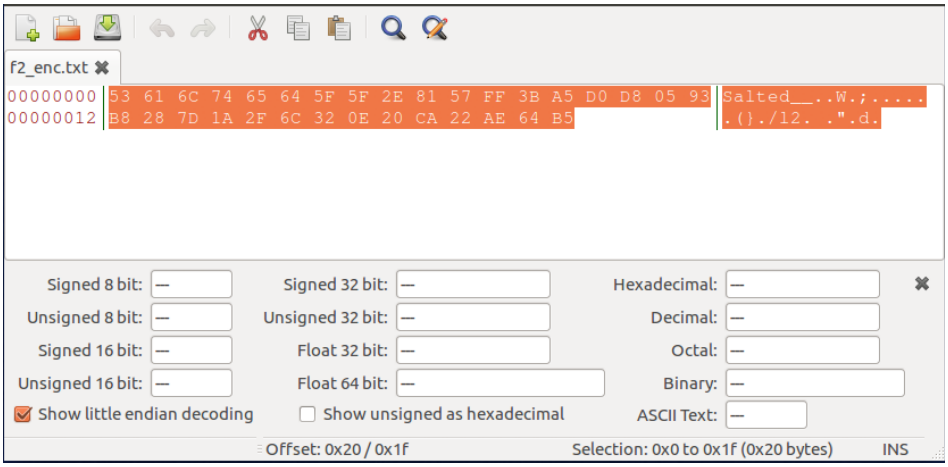
```
[06/10/21]seed@VM:~/.../Task4$ cat f1_plain.txt
12345
[06/10/21]seed@VM:~/.../Task4$ hexdump -C f1_plain.txt
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
00000010
[06/10/21]seed@VM:~/.../Task4$ xxd f1_plain.txt
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b  12345...........
```

Looking at data of file “f1_plain” using Bless, it shows that there are 11-byte paddings of “0B” in the file



Let's create a file f2.txt that contains 10 bytes and has a content of "1234567890". Next, we encrypt the message in a content file named "f2_enc.txt". The encrypted size of the data is also 32 bytes.

```
[06/10/21]seed@VM:~/.../Task4$ echo -n "1234567890" > f2.txt
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -e -in f2.txt -out f2_enc.txt
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
```



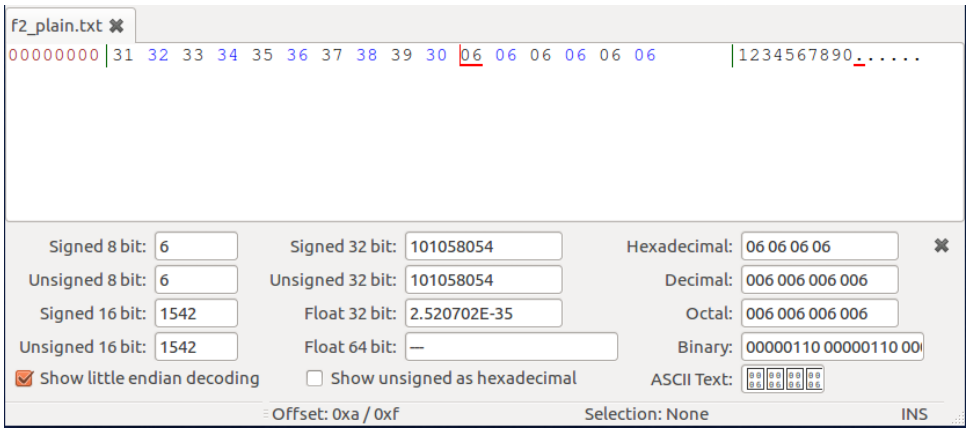
After that, we create an actual file "f2_enc.txt" and decrypt it to "f2_plain.txt", using -nopad option in order to keep padding in the decryption message

```
[06/10/21]seed@VM:~/.../Task4$ cat f2_enc.txt
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -d -nopad -in f2_enc.txt -out f2_plain.txt
enter aes-128-cbc decryption password:
```

Finally, we create "f2_plain.txt" and perform hexdump to see what paddings are added to the file

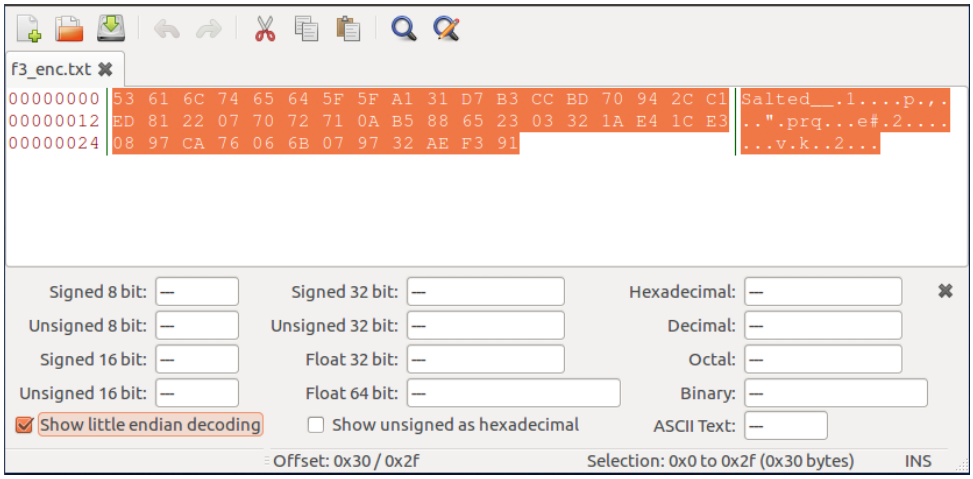
```
[06/10/21]seed@VM:~/.../Task4$ cat f2_plain.txt
1234567890[06/10/21]seed@VM:~/.../Task4$ hexdump -C f2_plain.txt
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06 06  |1234567890......|
00000010
[06/10/21]seed@VM:~/.../Task4$ xxd f2_plain.txt
00000000: 3132 3334 3536 3738 3930 0606 0606 0606  1234567890......
```

Looking at data of file "f2_plain" using Bless, it shows that there are 5-byte paddings of "06" in the file



Let's create a file f3.txt that contains 16 bytes and has a content of "1234567890abcdef". Next, we encrypt the message in a content file named "f3_enc.txt". The encrypted size of the data is 48 bytes.

```
[06/10/21]seed@VM:~/.../Task4$ echo -n "1234567890abcdef" > f3.txt
[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -e -in f3.txt -out f3_enc.txt
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
```



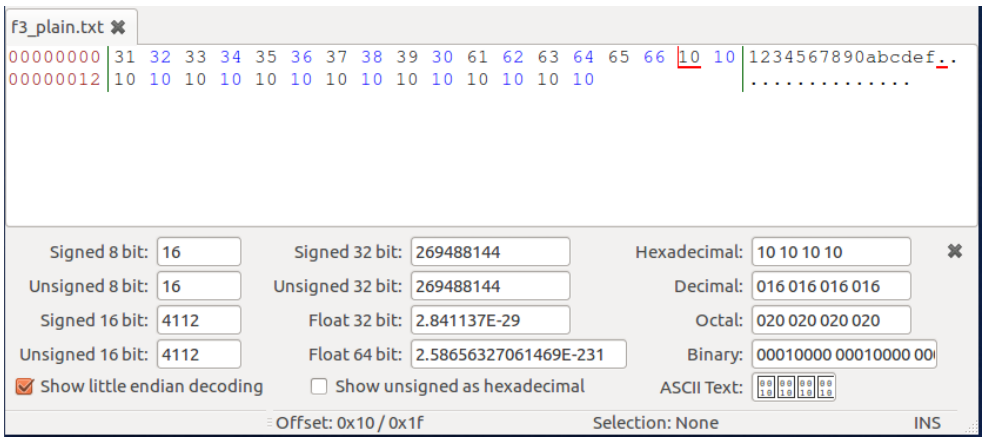
After that, we create an actual file "f3_enc.txt" and decrypt it to "f3_plain.txt", using -nopad option in order to keep padding in the decryption message

```
[06/10/21]seed@VM:~/.../Task4$ cat f3_enc.txt
Salted__ 1 p ,  "prq
  e#  v  2[06/10/21]seed@VM:~/.../Task4$ openssl enc -aes-128-cbc -d -nopad -in f3_enc.txt -out f3_plain.txt
enter aes-128-cbc decryption password:
```

Finally, we create "f3_plain.txt" and perform hexdump to see what paddings are added to the file

```
[06/10/21]seed@VM:~/.../Task4$ cat f3_plain.txt
1234567890abcdef[06/10/21]seed@VM:~/.../Task4$ hexdump -C f3_plain.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
[06/10/21]seed@VM:~/.../Task4$ xxd f3_plain.txt
00000000: 3132 3334 3536 3738 3930 6162 6364 6566  1234567890abcdef
00000010: 1010 1010 1010 1010 1010 1010 1010 1010  ................
```

Looking at data of file "f3_plain" using Bless, it shows that there are 16-byte paddings of "10" in the file



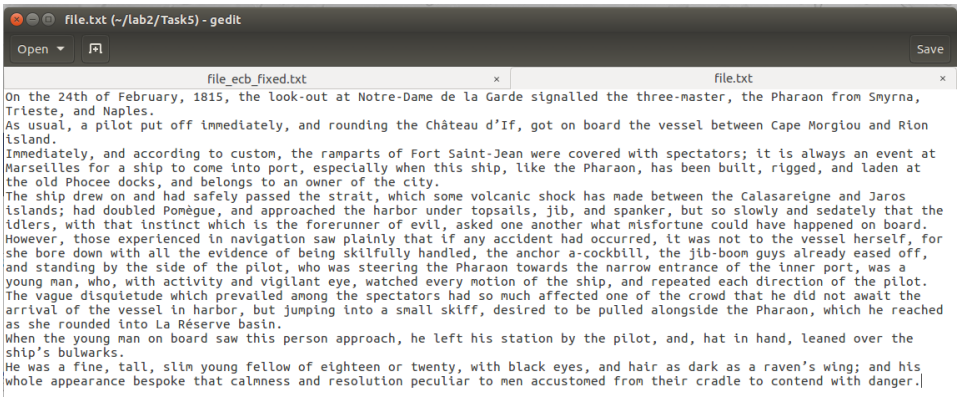
Task 5: Error Propagation

1) How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively?

   Answer:

   - In ECB mode, when any problem in a ciphertext happens, only one block is being affected. Each block decrypts independently from one another. However, the corrupted bit of the 55th byte in ciphertext blocks 8 bytes, and this might spread to all n bits in plaintext block 8 bytes since we do the decryption one block at a time.
   - In CBC mode, there was an effect on two blocks.
   - In CFB mode, there is a problem with every (n / r) number of blocks. Therefore, the error propagation criterion performs even worse.
   - In OFB mode, the feedback is only in the key-generation system. If the single digit of the 55th byte is corrupted, then only that byte or character is oxidized in the plaintext.

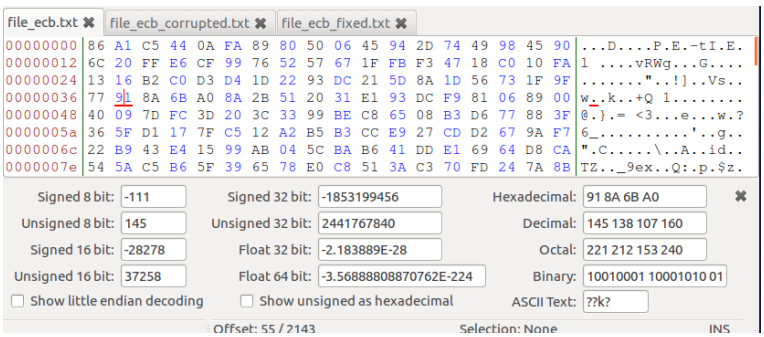
The text file content (from the first paragraph of “The Count Of Monte Cristo”)



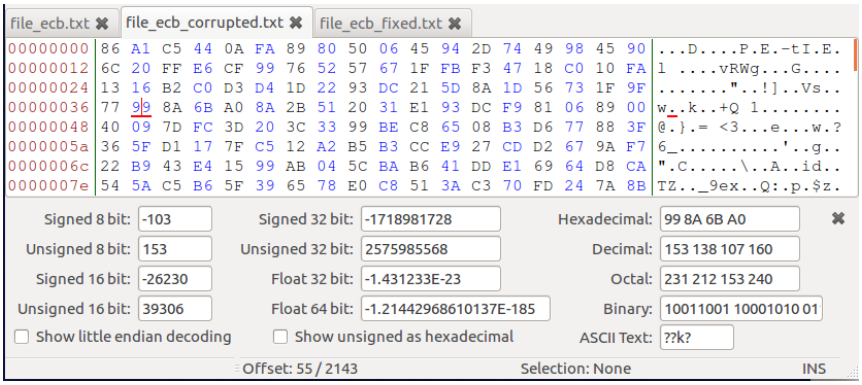
Next, we encrypt the text. Let’s use ECB mode first.

```
[06/10/21]seed@VM:~$ cd lab2/Task5
[06/10/21]seed@VM:~/.../Task5$ ls
file.txt
[06/10/21]seed@VM:~/.../Task5$ openssl enc -aes-128-ecb -e -in file.txt -out file_ecb.txt -K 0123456789abcdef
```

The click bar is at offset 55 [00000036 / 91]



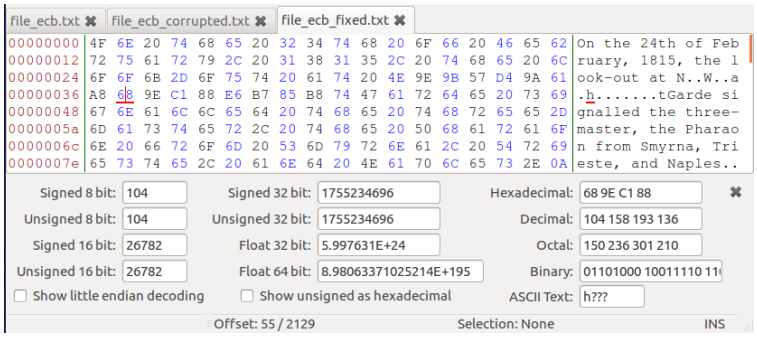
At offset 55 of encrypted file “file_ecb.txt”, let’s place 9 so that new 55th offset becomes “99” and change name of a new file to be “file_ecb_corrupted.txt”

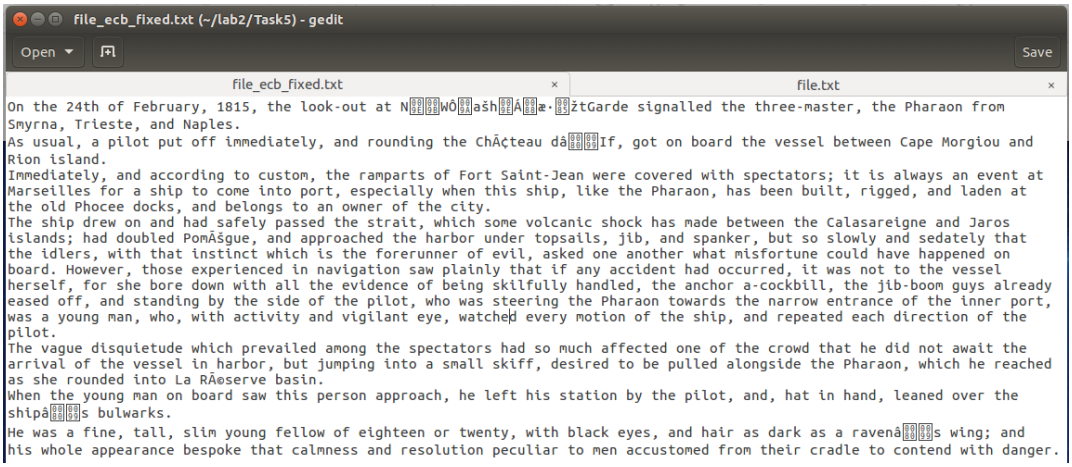


Using the code below, we decrypt “file_ecb_corrupted.txt” into “file_ecb_fixed.txt”

```
[06/10/21]seed@VM:~/.../Task5$ openssl enc -aes-128-ecb -d -in file_ecb_corrupted.txt -out file_ecb_fixed.txt -K 0123456789abcdef
[06/10/21]seed@VM:~/.../Task5$
```

This is the content of “file_ecb_fixed.txt”

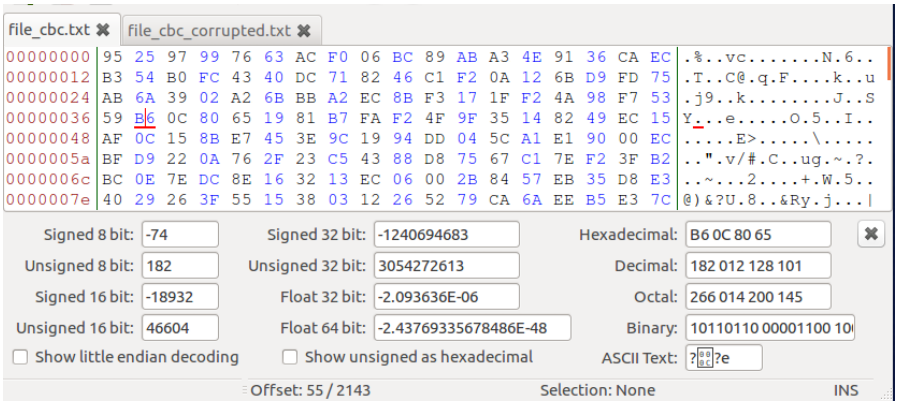




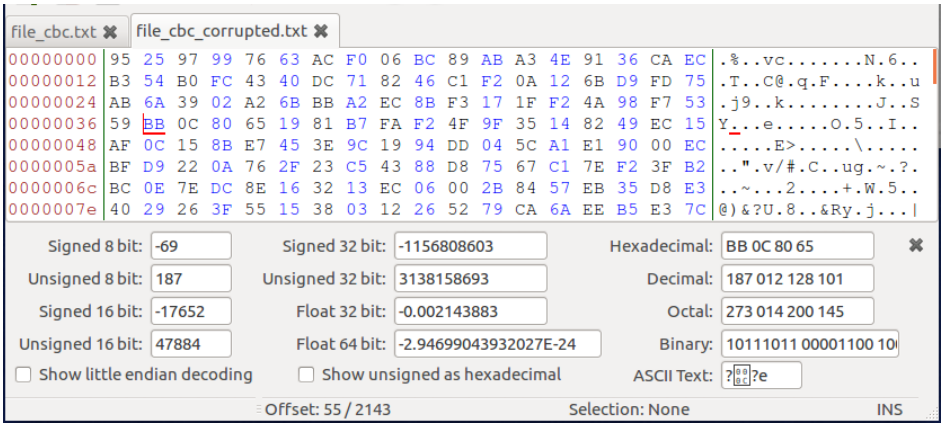
Next, let’s use CBC mode.

```
[06/10/21]seed@VM:~/.../Task5$ openssl enc -aes-128-cbc -e -in file.txt -out file_cbc.txt -K 0123456789abcdef -iv 012345678aabbccddeeff
[06/11/21]seed@VM:~/.../Task5$
```

The click bar is at offset 55 [00000036 / B6]



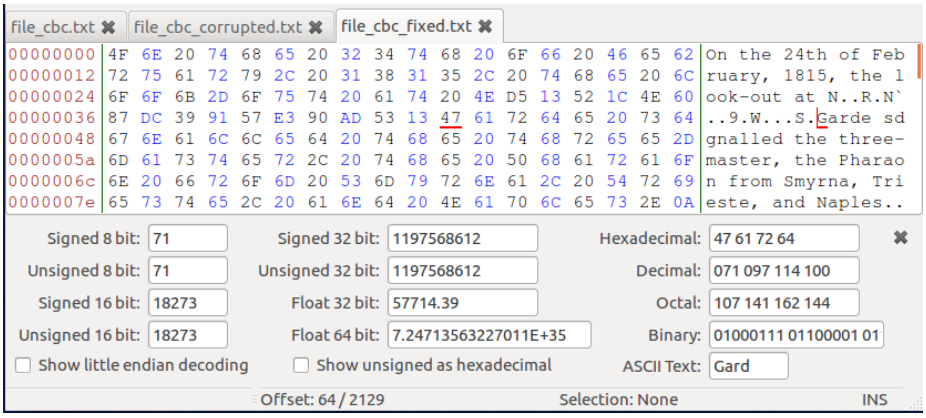
At offset 55 of encrypted file “file_cbc.txt”, let’s place 9 so that new 55th offset becomes “BB” and change name of a new file to be “file_cbc_corrupted.txt”

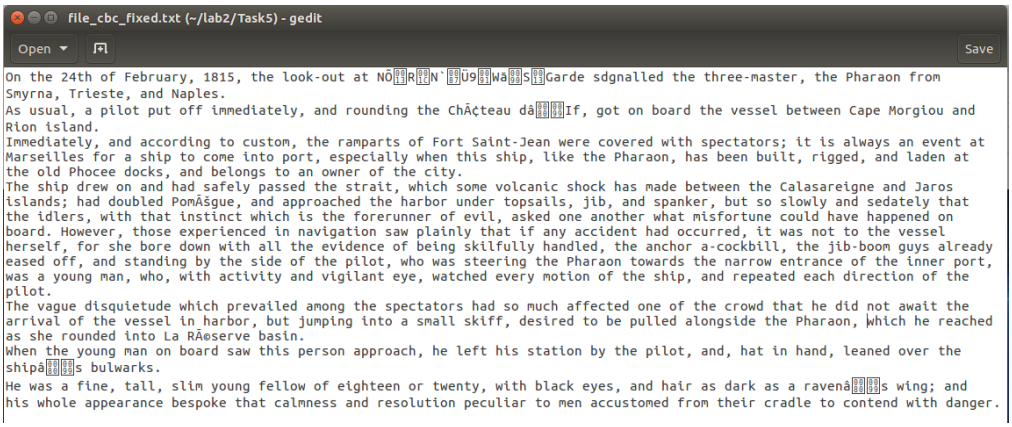


Using the code below, we decrypt “file_ecb_corrupted.txt” into “file_cbc_fixed.txt”

```
[06/11/21]seed@VM:~/.../Task5$ openssl enc -aes-128-cbc -d -in file_cbc_corrupted.txt -out file_cbc_fixed.txt -K 0123456789abcdef -iv 012345678aabbccddeeff
[06/11/21]seed@VM:~/.../Task5$
```

This is the content of “file_cbc_fixed.txt”

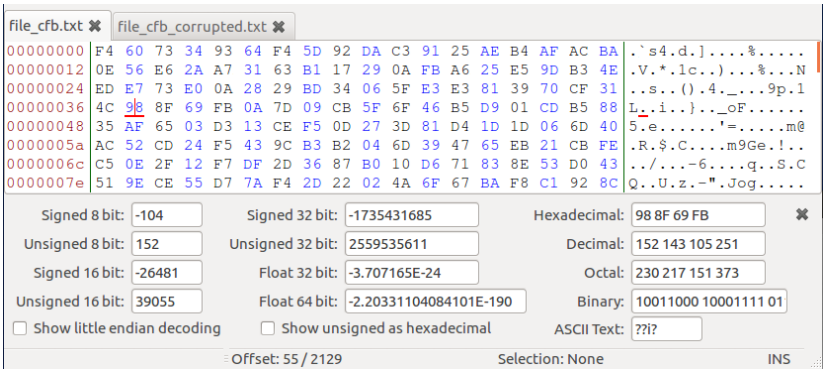




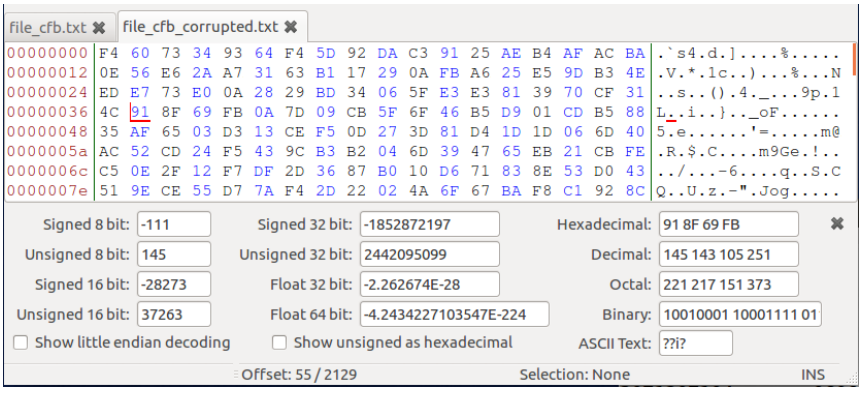
Next, let's use CFB mode.

```
[06/11/21]seed@VM:~/.../Task5$ openssl enc -aes-128-cfb -e -in file.txt -out file_cfb.txt -K 1023456789abcdfe -iv 0123456789aabbccddeeff
[06/11/21]seed@VM:~/.../Task5$
```

The click bar is at offset 55 [00000036 / 98]



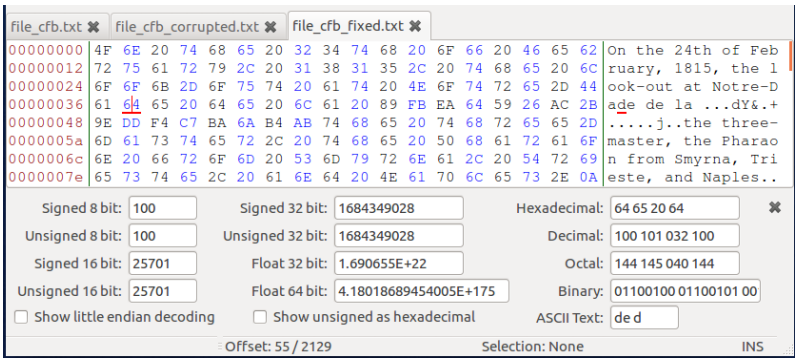
At offset 55 of encrypted file "file_cfb.txt", let's place 1 so that new 55th offset becomes "91" and change name of a new file to be "file_cfb_corrupted.txt"

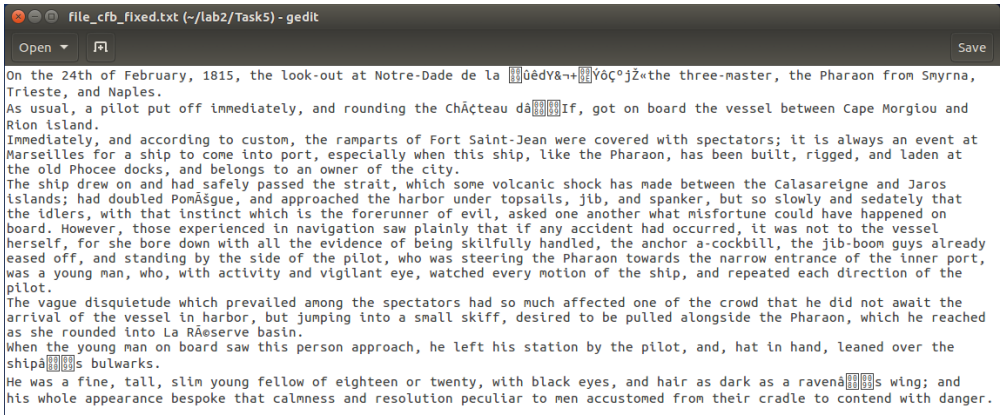


Using the code below, we decrypt "file_cfb_corrupted.txt" into "file_cfb_fixed.txt"

```
[06/11/21]seed@VM:~/.../Task5$ openssl enc -aes-128-cfb -d -in file_cfb_corrupted.txt -out file_cfb_fixed.txt -K 1023456789abcdfe -iv 0123456789aabbccddeeff
[06/11/21]seed@VM:~/.../Task5$
```

This is the content of “file_cfb_fixed.txt”

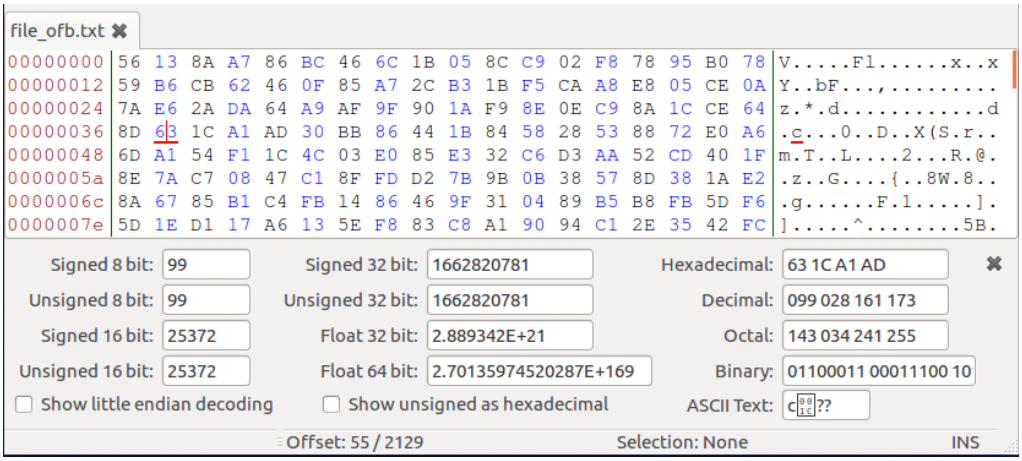




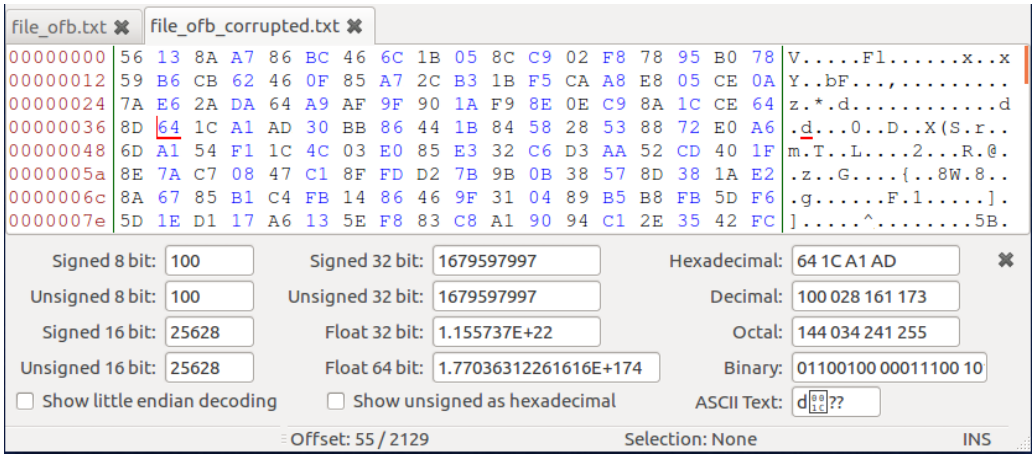
Next, let’s use OFB mode.

```
[06/11/21]seed@VM:~/.../Task5$ openssl enc -aes-128-ofb -e -in file.txt -out file_ofb.txt -K 192345678abcdfe -iv 1234506789abcdefabcdef
[06/11/21]seed@VM:~/.../Task5$
```

The click bar is at offset 55 [00000036 / 63]



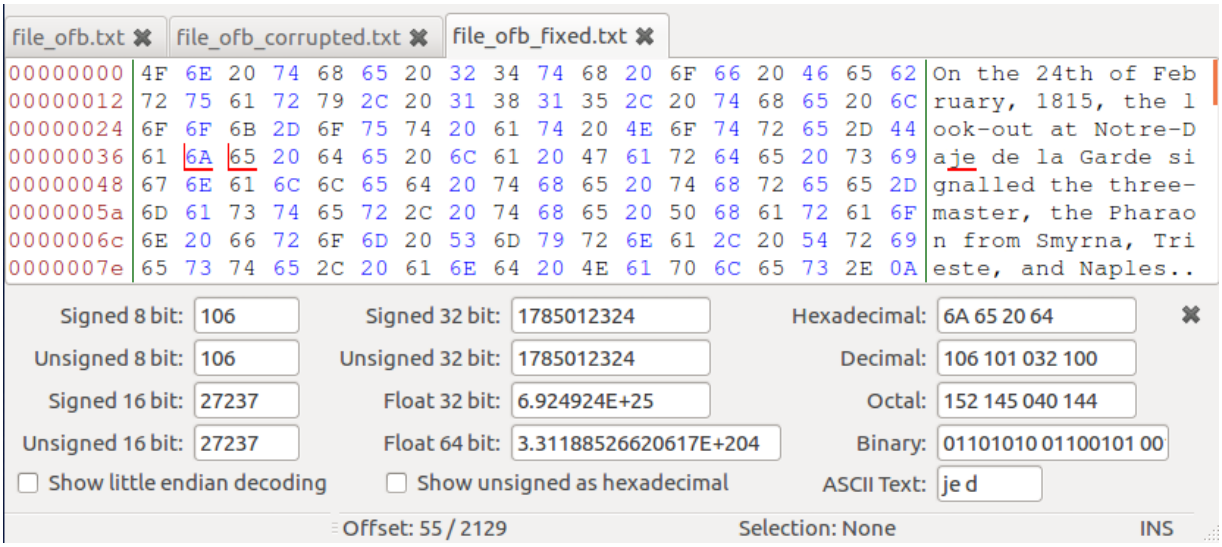
At offset 55 of encrypted file “file_ofb.txt”, let’s place 1 so that new 55th offset becomes “91” and change name of a new file to be “file_ofb_corrupted.txt”

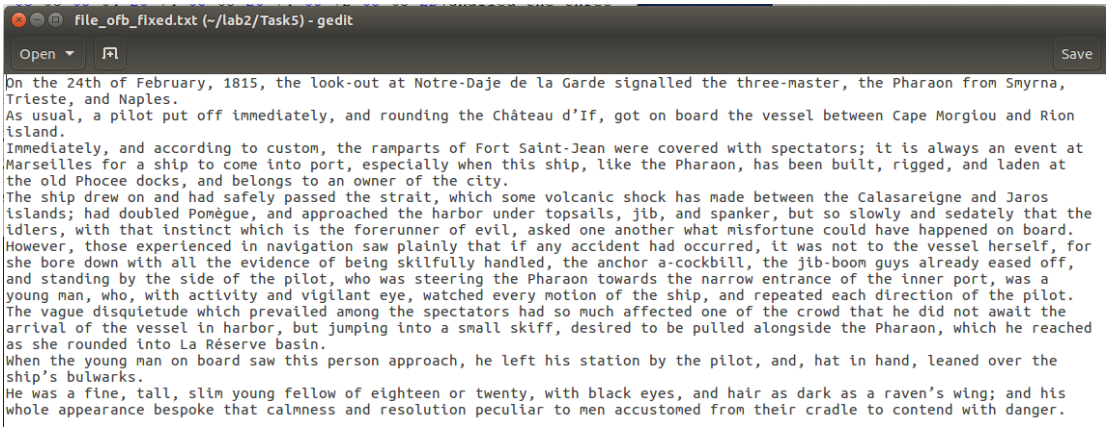


Using the code below, we decrypt “file_ofb_corrupted.txt” into “file_ofb_fixed.txt”

```
[06/11/21]seed@VM:~/.../Task5$ openssl enc -aes-128-ofb -d -in file_ofb_corrupted.txt -out file_ofb_fixed.txt -K 192345678abcdfe -iv 1234506789abcdefabcdef
[06/11/21]seed@VM:~/.../Task5$
```

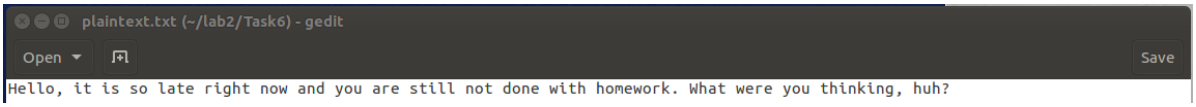
This is the content of “file_ofb_fixed.txt”





Task 6: Initial Vector

1) Let's create a plaintext with the following content



Then, we encrypt the plaintext into "ciphertext_cbc_one.txt" using cipher -aes-128-cbc, key 00112233445566778899aabbccddeeff, and IV 0123456789abcdef. Using "bless", we can see the hex content of "ciphertext_cbc_one.txt"

```
[06/10/21]seed@VM:~/.../Task6$ openssl enc -aes-128-cbc -e -in plaintext.txt -out ciphertext_cbc_one.txt -K 00112233445566778899aabbccddeeff -iv 0123456789abcdef
[06/10/21]seed@VM:~/.../Task6$
```

ciphertext_cbc_one.txt

```
00000000  48 A3 49 13 5A E7 0A 5F 28 54 B9 EA 92 00 AE DA D2 5B  H.I.Z.._(T.......[
00000012  E2 49 28 EC F1 8A 82 D7 D7 D4 5F B0 FA 99 3C 9B 4A A3  .I(......._...<.J.
00000024  BC 28 82 EA 95 41 27 E2 F8 68 70 21 05 DB 35 3A F7 C0  .(...A'..hp!..5:..
00000036  F3 26 1E 9F 5B 04 93 39 75 73 62 B5 61 65 85 D2 68 54  .&..[..9usb.ae..hT
00000048  54 29 4F 13 0D BC AD CB 8F 22 07 89 7A 76 A8 35 3E 95  T)O......"..zv.5>.
0000005a  D4 E0 71 43 D3 F8 A7 59 AF C9 A3 7C F4 2B B7 EB EC 1F  ..qC...Y...|.+....
0000006c  C7 5A 42 DF                                           .ZB.
```

Let's see what happens if we encrypt the plaintext into another ciphertext called "ciphertext_cbc_two.txt" using the same key and IV as before.

```
[06/10/21]seed@VM:~/.../Task6$ openssl enc -aes-128-cbc -e -in plaintext.txt -out ciphertext_cbc_two.txt -K 00112233445566778899aabbccddeeff -iv 0123456789abcdef
[06/10/21]seed@VM:~/.../Task6$
```

ciphertext_cbc_one.txt | ciphertext_cbc_two.txt

```
00000000  48 A3 49 13 5A E7 0A 5F 28 54 B9 EA 92 00 AE DA D2 5B  H.I.Z.._(T.......[
00000012  E2 49 28 EC F1 8A 82 D7 D7 D4 5F B0 FA 99 3C 9B 4A A3  .I(......._...<.J.
00000024  BC 28 82 EA 95 41 27 E2 F8 68 70 21 05 DB 35 3A F7 C0  .(...A'..hp!..5:..
00000036  F3 26 1E 9F 5B 04 93 39 75 73 62 B5 61 65 85 D2 68 54  .&..[..9usb.ae..hT
00000048  54 29 4F 13 0D BC AD CB 8F 22 07 89 7A 76 A8 35 3E 95  T)O......"..zv.5>.
0000005a  D4 E0 71 43 D3 F8 A7 59 AF C9 A3 7C F4 2B B7 EB EC 1F  ..qC...Y...|.+....
0000006c  C7 5A 42 DF                                           .ZB.
```

As can be seen from above, there is no change if we use the same initial vector. Now, we perform what if the initial vectors are different. Let's see the change:

```
[06/10/21]seed@VM:~/.../Task6$ openssl enc -aes-128-cbc -e -in plaintext.txt -out ciphertext_cbc_three.txt -K 00112233445566778899aabbccddeeff -iv 0123456789abcdeff
[06/10/21]seed@VM:~/.../Task6$
```

ciphertext_cbc_one.txt | ciphertext_cbc_two.txt | ciphertext_cbc_three.txt

```
00000000  6C E0 CD E5 7F 9C 4A FB 20 0E F1 C7 42 22 04 DF CE 19  l.....J. ...B"....
00000012  BB 01 18 96 C3 A7 BD D2 FC 31 6E 7E 22 56 AA 54 E7 39  .........1n~"V.T.9
00000024  B5 6F 58 8C E9 91 E1 30 16 FA 56 FA 40 2F F9 CB 00 EB  .oX....0..V.@/....
00000036  6B CA 3D B8 29 61 1A 0A D1 0B 51 55 EE EC FE 70 5A 6E  k.=.)a....QU...pZn
00000048  2B F0 C1 BD FB AE 94 E7 83 FA 66 EC CD B3 1D 1E 56 DD  +.........f.....V.
0000005a  D4 6C 9D 17 F2 B7 DA EF 42 AF 41 4C F3 B0 BC 62 71 62  .l......B.AL...bqb
0000006c  D2 BB 38 61                                           ..8a
```

The third ciphertext looks different from the first one. Based upon this observation, the IV must be unique because an attacker, who is given two ciphertext might have a difficult time to find if the two decrypt to the same plaintext or not. The uniqueness of IVs increases the security of plaintext.

2) Let's convert the plaintext, P1, to hex using Ascii table

P1 = This is a known message! => 54 68 69 73 20 69 73 20 61 20 6b 6e 6f 77 6e 20 6d 65 73 73 61 67 65 21 0a

C1 is already in hex, so we can find K, using formula K = P1 xor C1 (because we encrypt the message)

K =
54 68 69 73 20 69 73 20 61 20 6b 6e 6f 77 6e 20 6d 65 73 73 61 67 65 21 0a
xor
a4 69 b1 c5 02 c1 ca b9 66 96 5e 50 42 54 38 e1 bb 1b 5f 90 37 a4 c1 59 13
=
f0 01 d8 b6 22 a8 b9 99 07 b6 35 3e 2d 23 56 c1 d6 7e 2c e3 56 c3 a4 78 19

Since the usage of IV was the same, we assume that P2 = K xor C2 (because we decrypt the message)

P2 =
f0 01 d8 b6 22 a8 b9 99 07 b6 35 3e 2d 23 56 c1 d6 7e 2c e3 56 c3 a4 78 19
xor
bf 73 bc d3 50 92 99 d5 66 c3 5b 5d 45 03 37 e1 bb 17 5f 90 3f af c1 59 13
=
4f 72 64 65 72 3a 20 4c 61 75 6e 63 68 20 61 20 6d 69 73 73 69 6c 65 21 0a

Hex to Ascii in P2 gives result:

"Order: Launch a missile!"

If OFB is replaced with CFB, then only the first block can be revealed from the plaintext OFB. If the IV is reused, the whole key stream will be reproduced again, since the keystream is done by successive encryption of the IV: K_0=IV, K_1=E_k(C_0), K_2= E_k(C_1), ...

3) Assume that P1 is "Yes". According to Ascii table, "Yes" turns to be 59 65 73 in hex