Task 1: Encryption using different ciphers and modes

I created a plaintext3.1.txt file which consists of “Hello Computer Security” string.

I used the following ciphers.

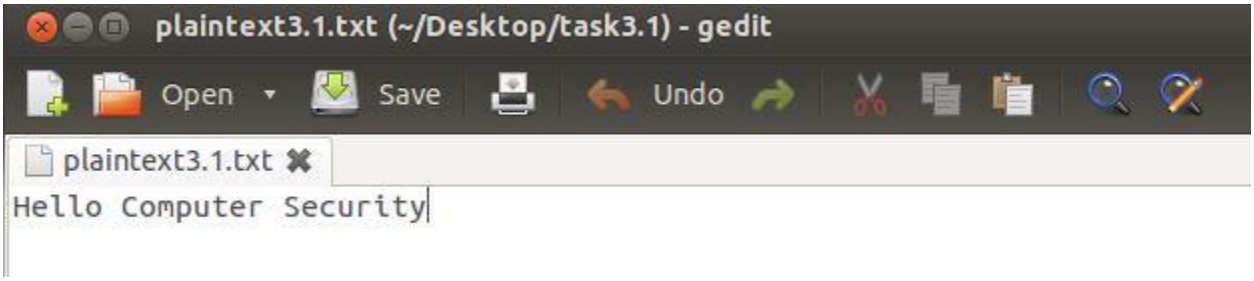
1) AES-128
2) AES-256
3) BF

I used the following encryption modes.

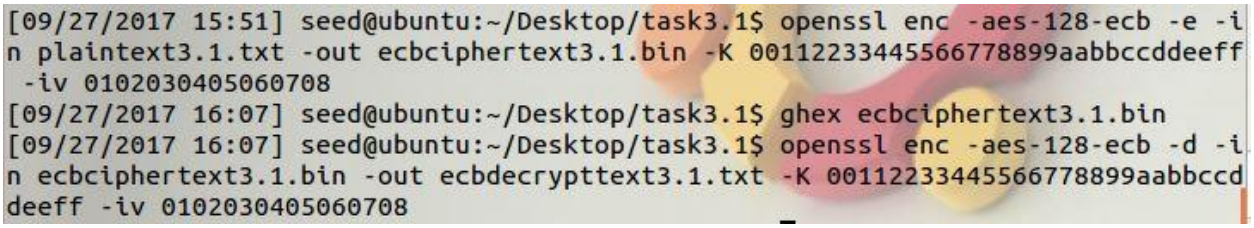
1) ECB
2) CBC
3) OFB
4) CFB

Cipher and Mode: AES-128 and CBC

- The plaintext file:

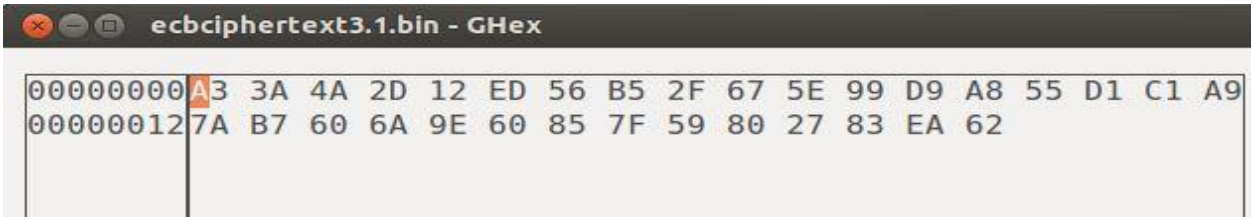


- Encryption and Decryption code

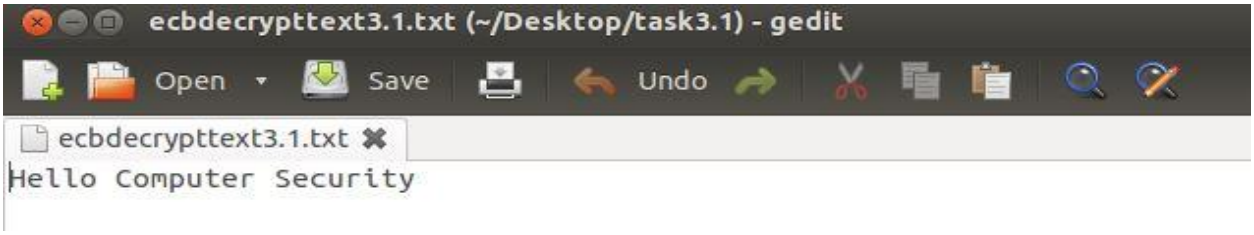


I tried opening the encrypted ecbciphertext3.1.bin file but it is giving an error because it is a binary file. So, I opened the bin file in a hexadecimal format by using GHEX editor.

- Encrypted file in GHEX editor

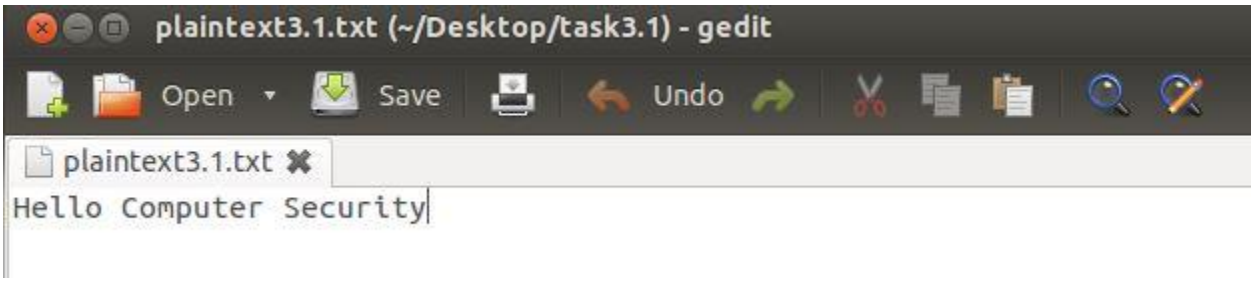


- Decrypted file



Cipher and Mode: AES-128 and CBC

- The plaintext file:

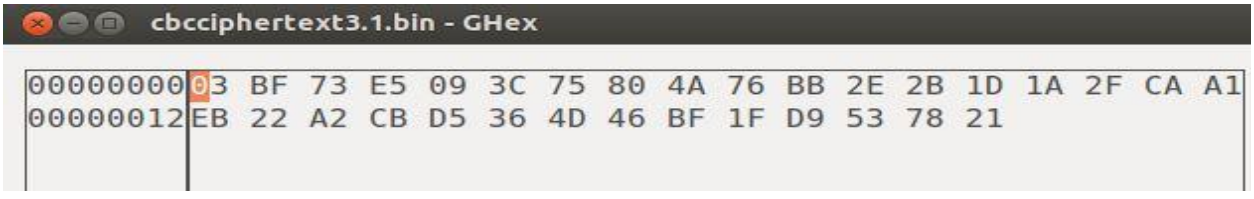


- Encryption and Decryption code

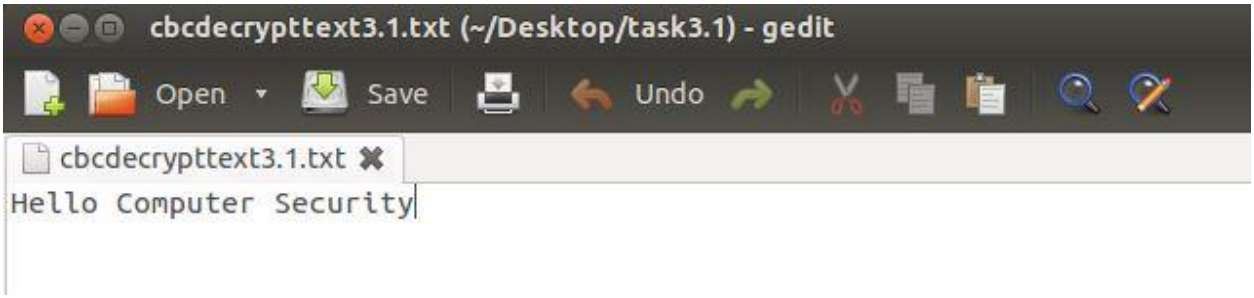
```
[09/27/2017 15:51] seed@ubuntu:~/Desktop/task3.1$ openssl enc -aes-128-cbc -e -i
n plaintext3.1.txt -out cbcciphertext3.1.bin -K 00112233445566778899aabbccddeeff
 -iv 0102030405060708
[09/27/2017 15:51] seed@ubuntu:~/Desktop/task3.1$ ghex cbcciphertext3.1.bin
[09/27/2017 15:51] seed@ubuntu:~/Desktop/task3.1$ openssl enc -aes-128-cbc -d -i
n cbcciphertext3.1.bin -out cbcdecrypttext3.1.txt -K 00112233445566778899aabbccd
deeff -iv 0102030405060708
```

I tried opening the encrypted cbcciphertext3.1.bin file but it is giving an error because it is a binary file. So, I opened the bin file in a hexadecimal format by using GHEX editor.

- Encrypted file in GHEX editor

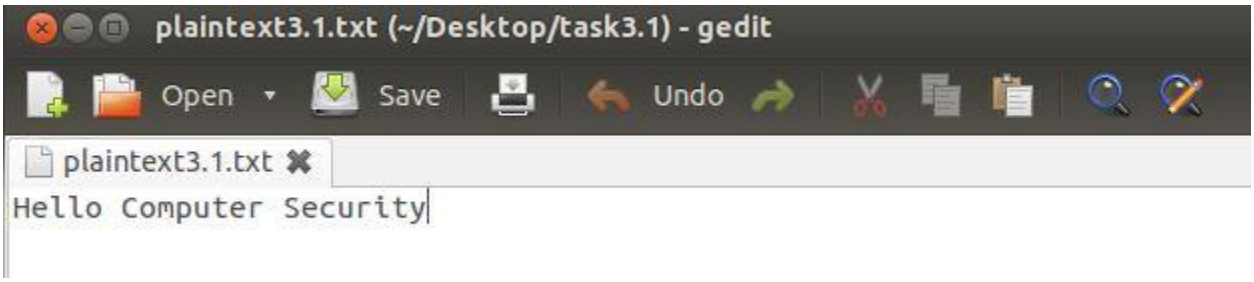


- Decrypted file



Cipher and Mode: AES-128 and OFB

- The plaintext file:

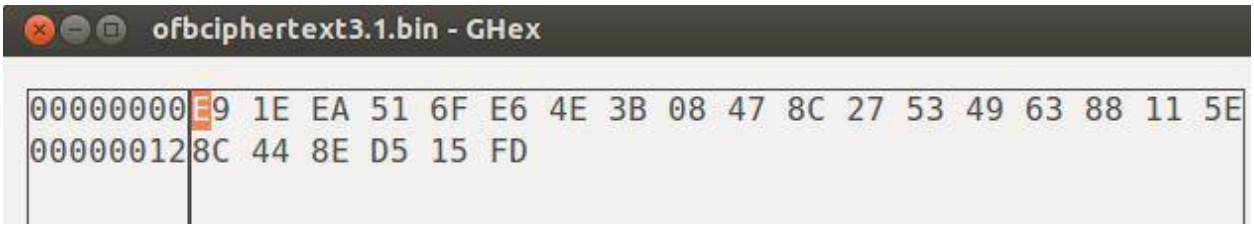


- Encryption and Decryption code

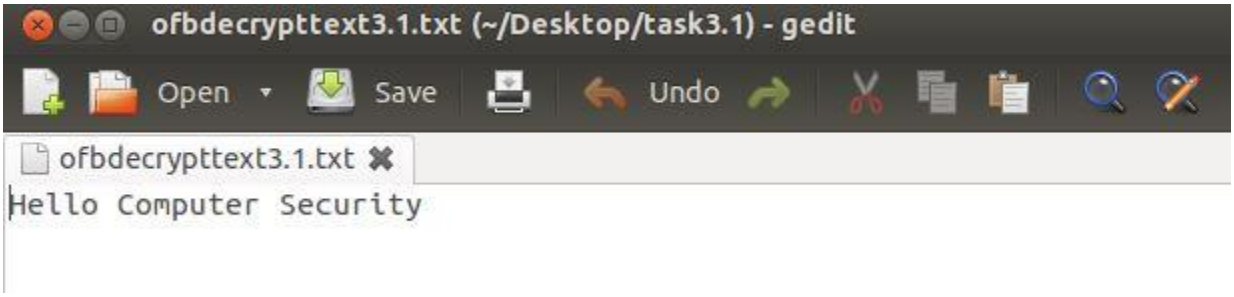
```
[09/27/2017 16:08] seed@ubuntu:~/Desktop/task3.1$ openssl enc -aes-128-ofb -e -i
n plaintext3.1.txt -out ofbciphertext3.1.bin -K 00112233445566778899aabbccddeeff
 -iv 0102030405060708
[09/27/2017 16:13] seed@ubuntu:~/Desktop/task3.1$ ghex ofbciphertext3.1.bin
[09/27/2017 16:14] seed@ubuntu:~/Desktop/task3.1$ openssl enc -aes-128-ofb -d -i
n ofbciphertext3.1.bin -out ofbdecrypttext3.1.txt -K 00112233445566778899aabbccd
deeff -iv 0102030405060708
```

I tried opening the encrypted ofbciphertext3.1.bin file but it is giving an error because it is a binary file. So, I opened the bin file in a hexadecimal format by using GHEX editor.

- Encrypted file in GHEX editor

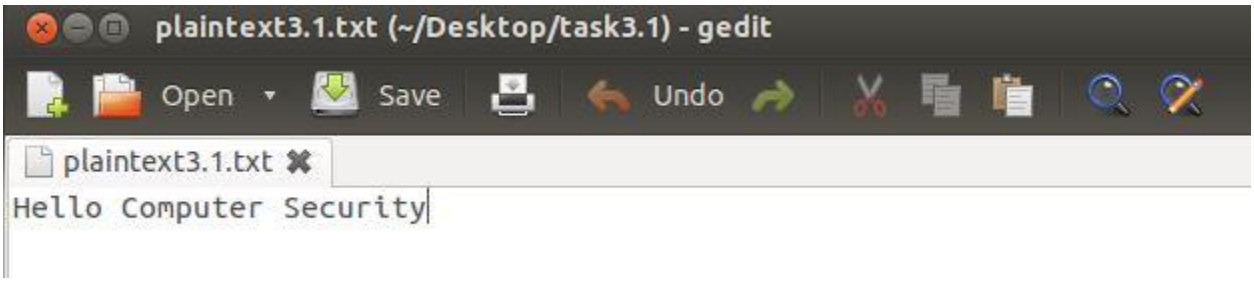


- Decrypted file

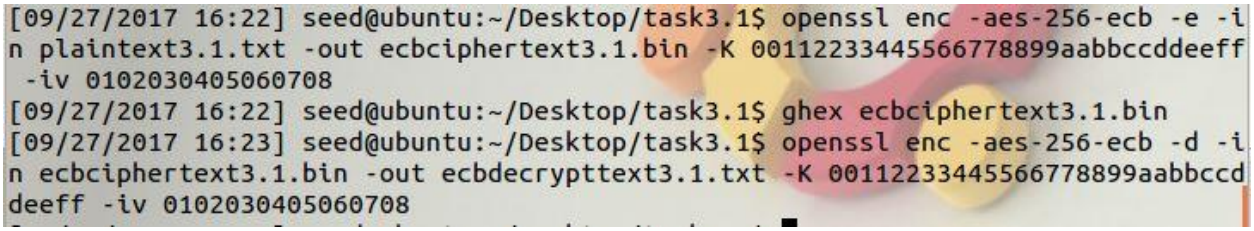


Cipher and Mode: AES-256 and ECB

- The plaintext file:

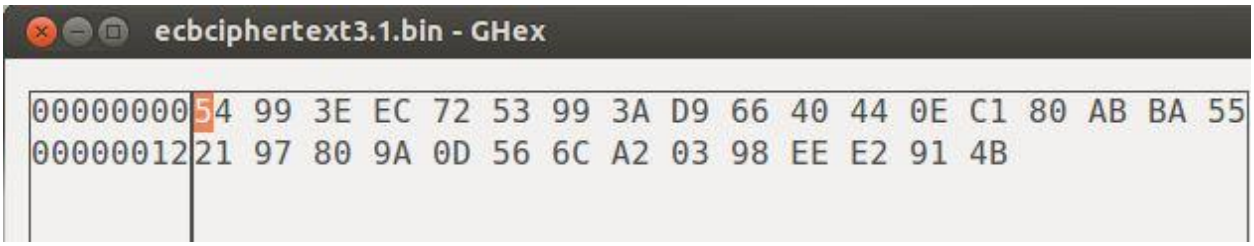


- Encryption and Decryption code

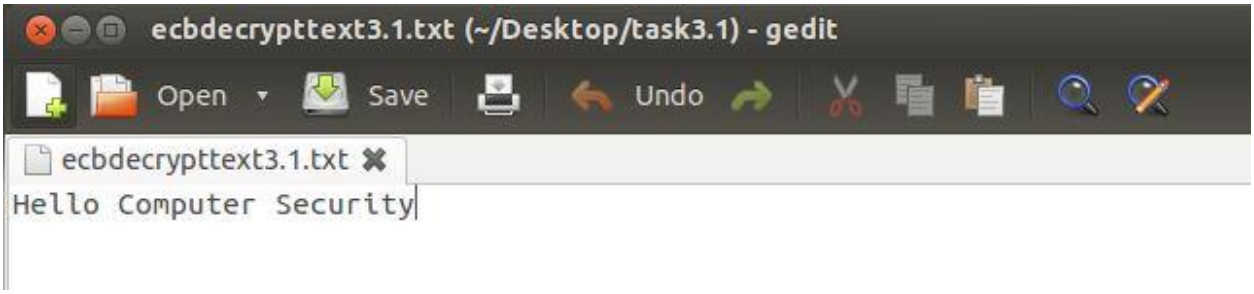


I tried opening the encrypted ecbciphertext3.1.bin file but it is giving an error because it is a binary file. So, I opened the bin file in a hexadecimal format by using GHEX editor.

- Encrypted file in GHEX editor

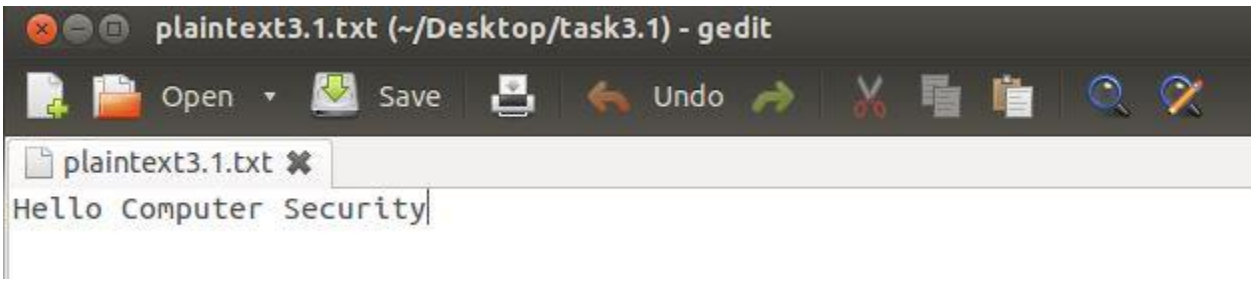


- Decrypted file



Cipher and Mode: AES-256 and CBC

- The plaintext file:

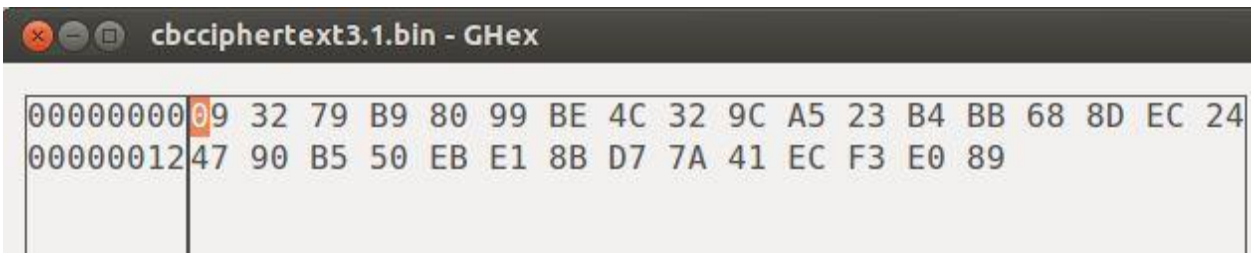


- Encryption and Decryption code

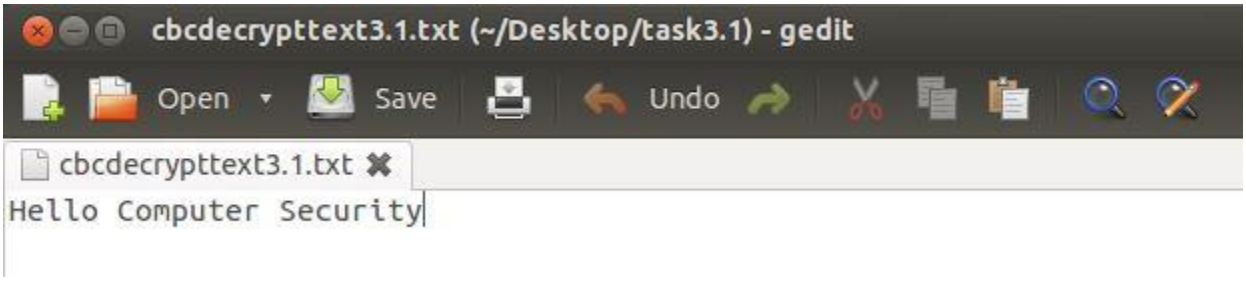
```
[09/27/2017 16:23] seed@ubuntu:~/Desktop/task3.1$ openssl enc -aes-256-cbc -e -i
n plaintext3.1.txt -out cbcciphertext3.1.bin -K 00112233445566778899aabbccddeeff
 -iv 0102030405060708
[09/27/2017 16:32] seed@ubuntu:~/Desktop/task3.1$ ghex cbcciphertext3.1.bin
[09/27/2017 16:33] seed@ubuntu:~/Desktop/task3.1$ openssl enc -aes-256-cbc -d -i
n cbcciphertext3.1.bin -out cbcdecrypttext3.1.txt -K 00112233445566778899aabbccd
deeff -iv 0102030405060708
```

I tried opening the encrypted cbcciphertext3.1.bin file but it is giving an error because it is a binary file. So, I opened the bin file in a hexadecimal format by using GHEX editor.

- Encrypted file in GHEX editor

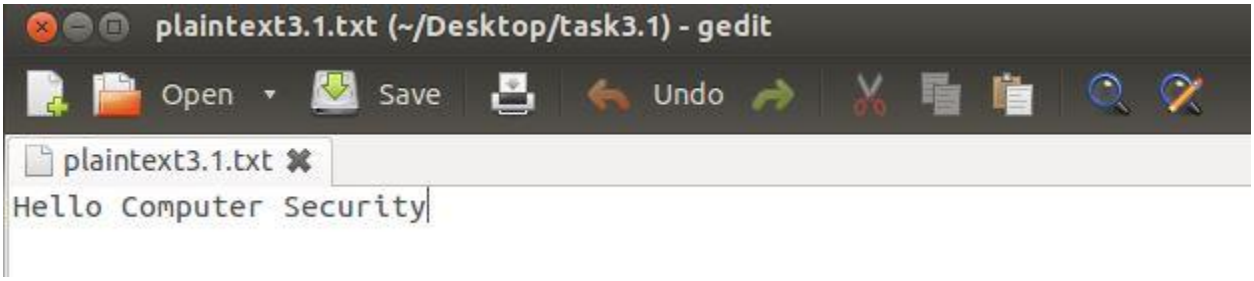


- Decrypted file



Cipher and Mode: AES-256 and CFB

- The plaintext file:

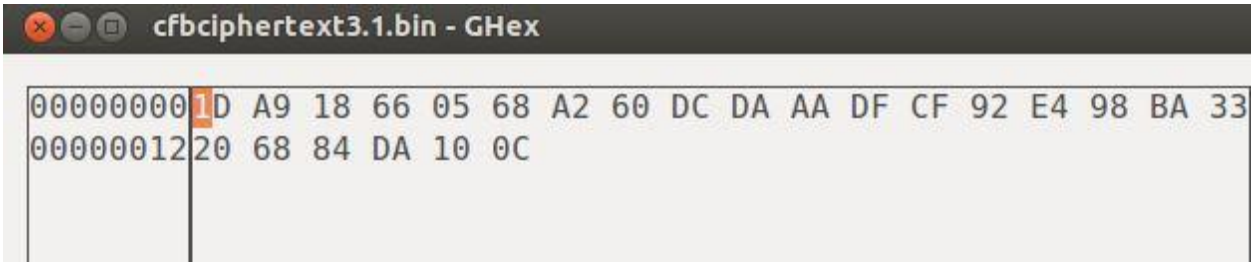


- Encryption and Decryption code

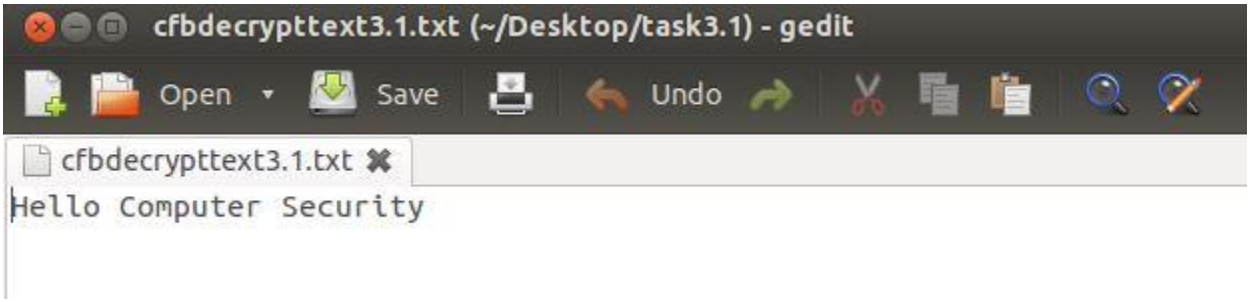
```
[09/27/2017 16:34] seed@ubuntu:~/Desktop/task3.1$ openssl enc -aes-256-cfb -e -i
n plaintext3.1.txt -out cfbciphertext3.1.bin -K 00112233445566778899aabbccddeeff
 -iv 0102030405060708
[09/27/2017 16:37] seed@ubuntu:~/Desktop/task3.1$ ghex cfbciphertext3.1.bin
[09/27/2017 16:38] seed@ubuntu:~/Desktop/task3.1$ openssl enc -aes-256-cfb -d -i
n cfbciphertext3.1.bin -out cfbdecrypttext3.1.txt -K 00112233445566778899aabbccd
deeff -iv 0102030405060708
```

I tried opening the encrypted cfbciphertext3.1.bin file but it is giving an error because it is a binary file. So, I opened the bin file in a hexadecimal format by using GHEX editor.

- Encrypted file in GHEX editor

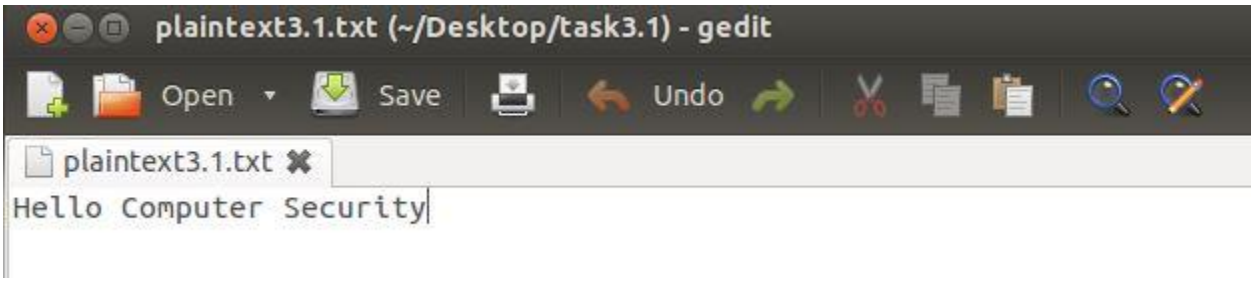


- Decrypted file

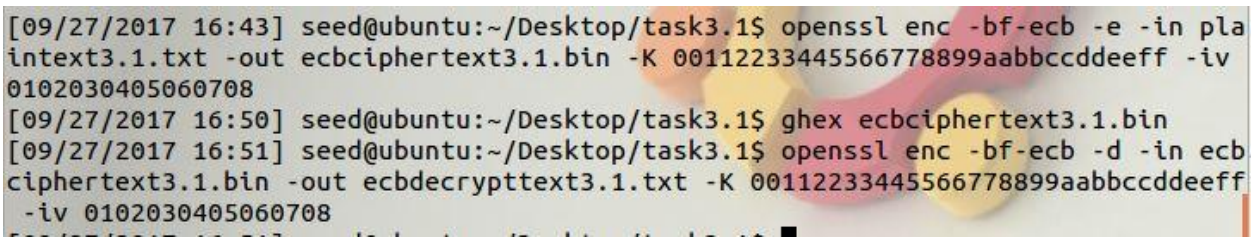


Cipher and Mode: BF and ECB

- The plaintext file:

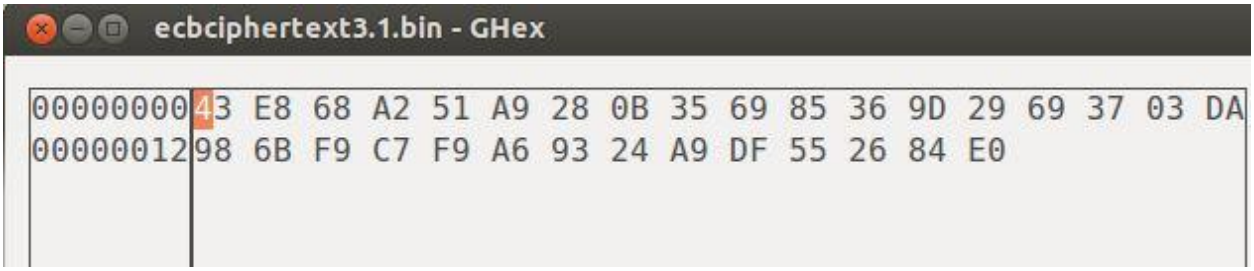


- Encryption and Decryption code

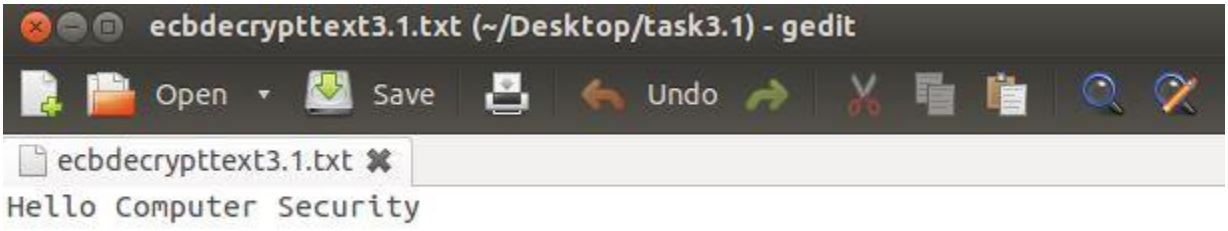


I tried opening the encrypted ecbciphertext3.1.bin file but it is giving an error because it is a binary file. So, I opened the bin file in a hexadecimal format by using GHEX editor.

- Encrypted file in GHEX editor

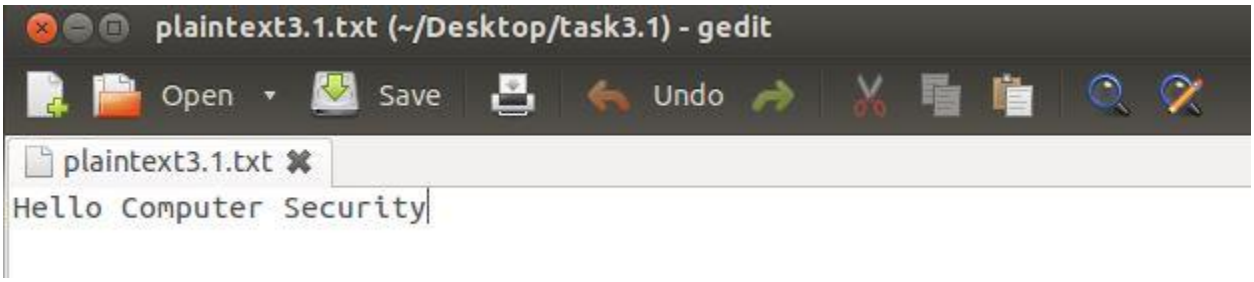


- Decrypted file

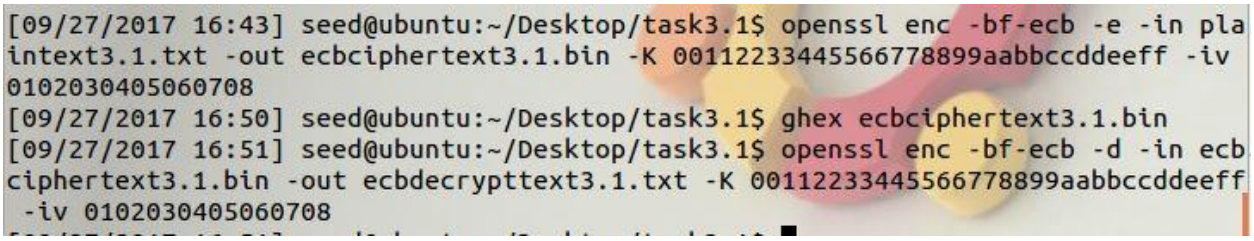


Cipher and Mode: BF and CBC

- The plaintext file:

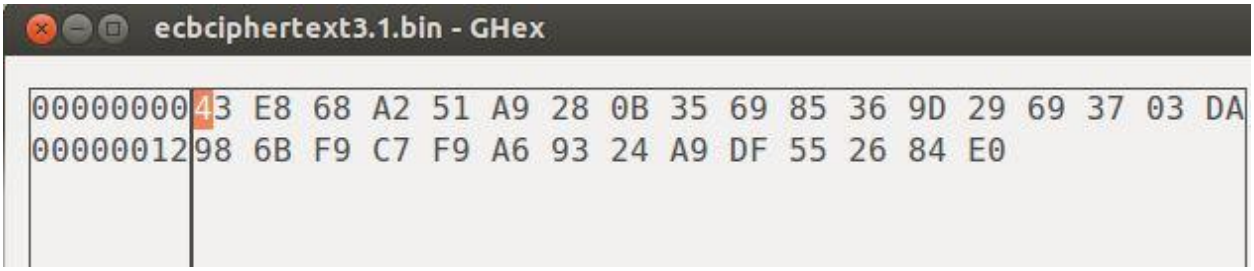


- Encryption and Decryption code

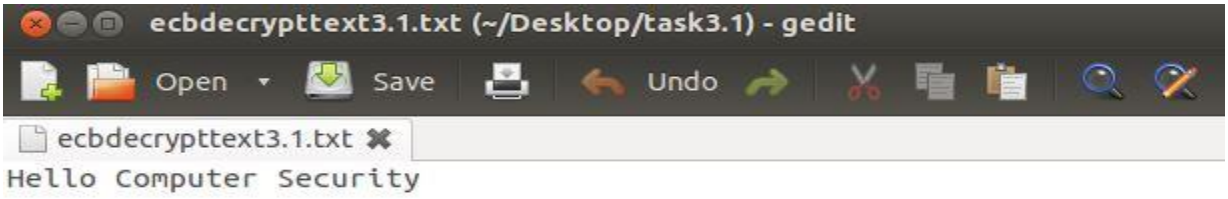


I tried opening the encrypted cbcciphertext3.1.bin file but it is giving an error because it is a binary file. So, I opened the bin file in a hexadecimal format by using GHEX editor.

- Encrypted file in GHEX editor

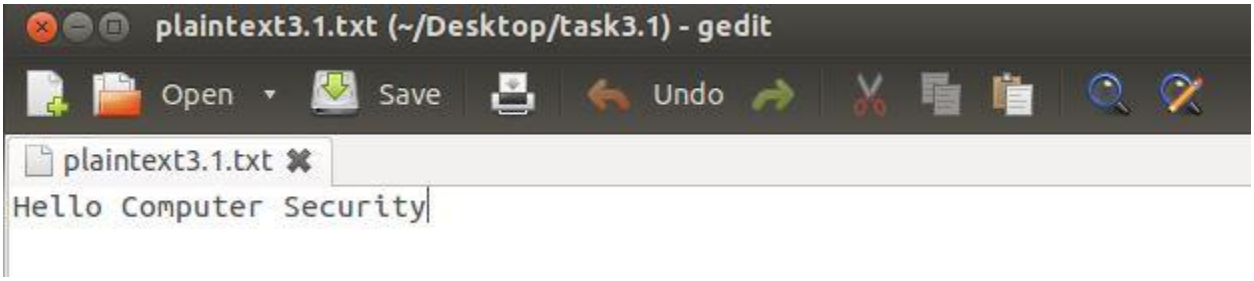


- Decrypted file



Cipher and Mode: BF and OFB

- The plaintext file:

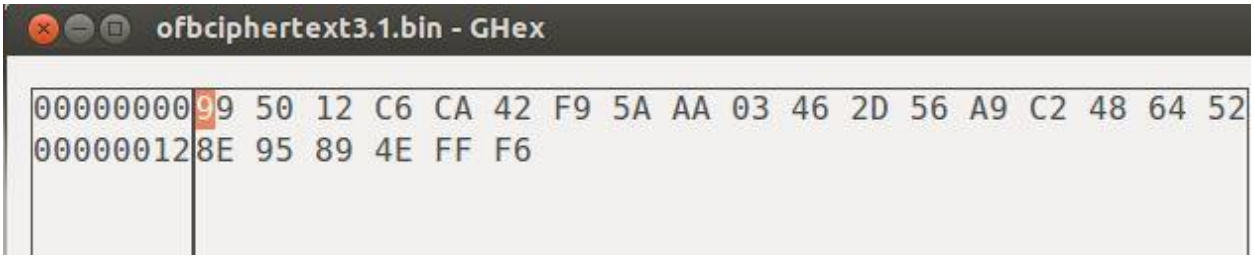


- Encryption and Decryption code

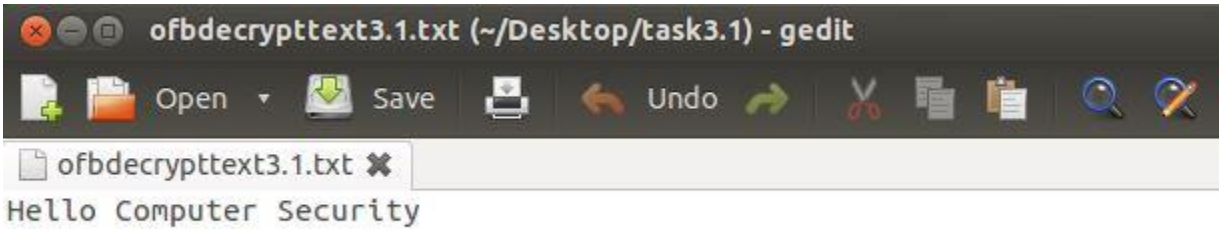
```
[09/27/2017 16:55] seed@ubuntu:~/Desktop/task3.1$ openssl enc -bf-ofb -e -in pla
intext3.1.txt -out ofbciphertext3.1.bin -K 00112233445566778899aabbccddeeff -iv
0102030405060708
[09/27/2017 16:57] seed@ubuntu:~/Desktop/task3.1$ ghex ofbciphertext3.1.bin
[09/27/2017 16:58] seed@ubuntu:~/Desktop/task3.1$ openssl enc -bf-ofb -d -in ofb
ciphertext3.1.bin -out ofbdecrypttext3.1.txt -K 00112233445566778899aabbccddeeff
 -iv 0102030405060708
```

I tried opening the encrypted ofbciphertext3.1.bin file but it is giving an error because it is a binary file. So, I opened the bin file in a hexadecimal format by using GHEX editor.

- Encrypted file in GHEX editor

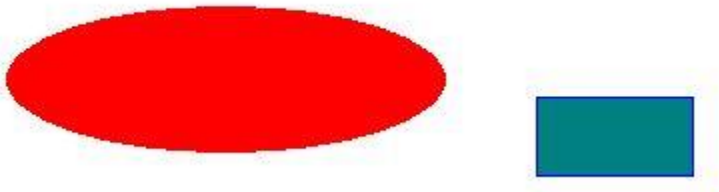


- Decrypted file



Task 2: Encryption mode – ECB vs CBC

Simple.bmp image:



CBC: Cipher Block Chaining

Code:

```
[09/26/2017 01:08] seed@ubuntu:~/Desktop/ecbvscbc$ head -c 54 simple.bmp > cbc.bmp
[09/26/2017 01:09] seed@ubuntu:~/Desktop/ecbvscbc$ openssl enc -aes-128-cbc -e -in simple.bmp -out temp1.bin -K 00112233445566778899 -iv 0011223
3445566778899
[09/26/2017 01:09] seed@ubuntu:~/Desktop/ecbvscbc$ ls -ltr
total 740
-rw-rw-r-- 1 seed seed 184974 Sep 25 15:03 simple.bmp
-rw-rw-r-- 1 seed seed 184976 Sep 26 01:05 temp.bin
-rw-rw-r-- 1 seed seed 184976 Sep 26 01:08 ecb.bmp
-rw-rw-r-- 1 seed seed     54 Sep 26 01:09 cbc.bmp
-rw-rw-r-- 1 seed seed 184976 Sep 26 01:09 temp1.bin
[09/26/2017 01:09] seed@ubuntu:~/Desktop/ecbvscbc$ tail -c 184922 temp1.bin >> ecb.bmp
[09/26/2017 01:10] seed@ubuntu:~/Desktop/ecbvscbc$ tail -c 184922 temp1.bin >> cbc.bmp
```

First, I used “head -c 54 Simple.bmp > cbc.bmp“ to transfer 54 bytes header information to cbc.bmp file. Then I opened the cbc.bmp file which shows the black color image. Next, the Simple.bmp is taken as an input with a key and initialization vector and encrypted into a temp1.bin file. In the above image, we can see that the cbc.bmp consists of only 54 bytes of information. Next, I used the “tail -c (184976-54) temp1.bin >> cbc.bmp” to embed the encrypted file into the cbc.bmp image which consists of 54 bytes initially.

Output: cbc.bmp

Observation: The whole image is blurred which provides higher confidentiality with a more secure mechanism because it used the XOR operation before encryption.

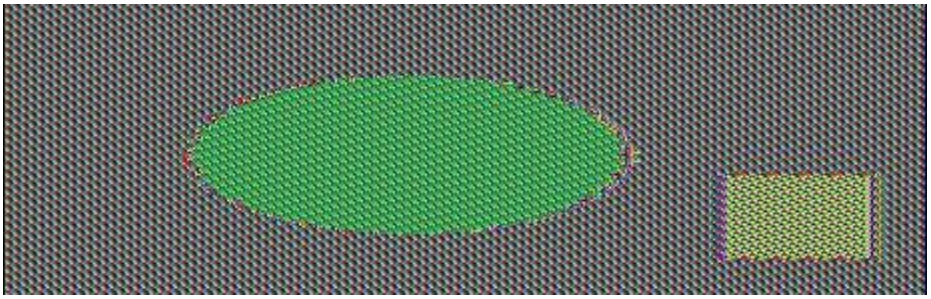
ECB: Electronic Code Book

Code:

```
[09/26/2017 01:04] seed@ubuntu:~/Desktop/ecbvscbc$ head -c 54 simple.bmp > ecb.bmp
[09/26/2017 01:04] seed@ubuntu:~/Desktop/ecbvscbc$ openssl enc -aes-128-ecb -e -in simple.bmp -out temp.bin -K 00112233445566778899 -iv 00112233
445566778899
[09/26/2017 01:05] seed@ubuntu:~/Desktop/ecbvscbc$ ls -ltr
total 372
-rw-rw-r-- 1 seed seed 184974 Sep 25 15:03 simple.bmp
-rw-rw-r-- 1 seed seed     54 Sep 26 01:04 ecb.bmp
-rw-rw-r-- 1 seed seed 184976 Sep 26 01:05 temp.bin
[09/26/2017 01:05] seed@ubuntu:~/Desktop/ecbvscbc$ tail -c 184922 temp.bin >> ecb.bmp
```

First, I used “head -c 54 Simple.bmp > ecb.bmp“ to transfer 54 bytes header information to ecb.bmp file. Then I opened the ecb.bmp file which shows the black color image. Next, the Simple.bmp is taken as an input with a key and initialization vector and encrypted into a temp.bin file. In the above image, one can see that the ecb.bmp consists of only 54 bytes of information. Next, I used the “tail -c (184976-54) temp.bin >> ecb.bmp” to embed the encrypted file into the ecb.bmp image which consists of 54 bytes initially.

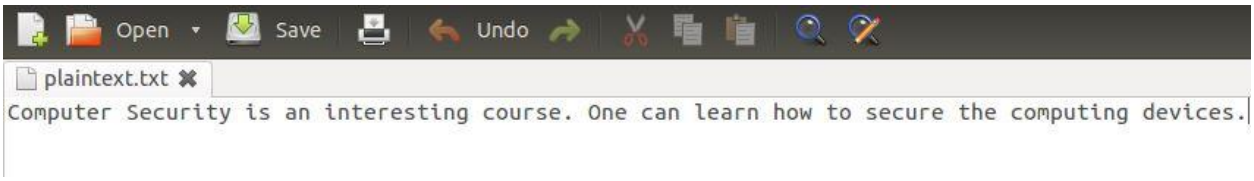
Output: cbc.bmp



Observation: In ECB encryption mode, when the identical plaintext blocks are encrypted into the identical cipher text blocks, the data patterns are not well hidden. So the is no confidentiality provided. We can see the image in the above picture. So, **CBC encryption mode is more secure than ECB.**

Task 3: Encryption Mode – Corrupted Cipher Text

I have created a plaintext.txt file whose size is 96bytes. This is the data that have written in the plaintext.txt.



I encrypted this file using aes-128 cipher with four encryption modes. The four encryption modes are ECB, CBC, CFB and, OFB. Then I changed the single bit of the 30th byte in the encrypted file to make it corrupted. I achieved this by using the GHEX editor. After that I decrypted the corrupted files using the same key and IV. The results explained below.

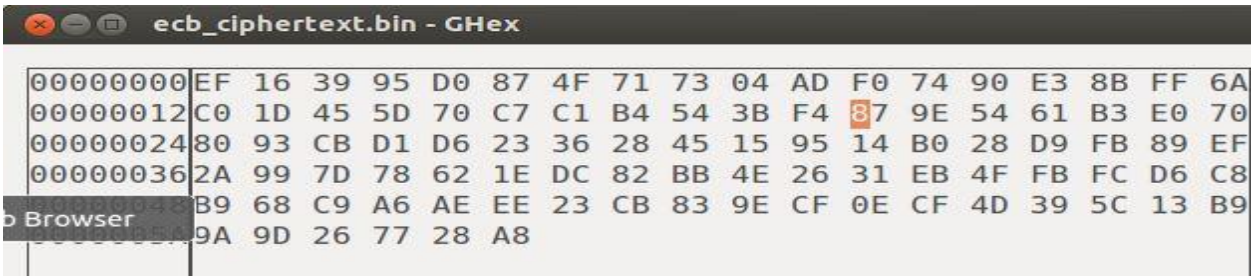
ECB Encryption mode:

I encrypted the plaintext.txt file into ecb_ciphertext.bin. Then I changed the 30th byte in the ecb_ciphertext.bin by using the GHEX editor and decrypted it into ecb_decrypt.txt.

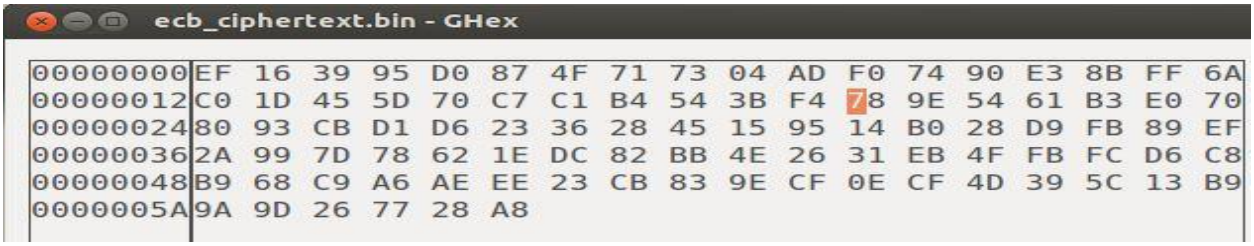
- Encryption and Decryption

```
[09/27/2017 11:46] seed@ubuntu:~/Desktop/enc modes$ openssl enc -aes-128-ecb -e
-in plaintext.txt -out ecb_ciphertext.bin -K 00112233445566778899 -iv 0011223344
5566778899
[09/27/2017 11:48] seed@ubuntu:~/Desktop/enc modes$ openssl enc -aes-128-ecb -d
-in ecb_ciphertext.bin -out ecb_decrypt -K 00112233445566778899 -iv 001122334455
66778899
```

- Original Encrypted file (ecb_ciphertext.bin)



- Corrupted Encrypted file (ecb_ciphertext.bin)



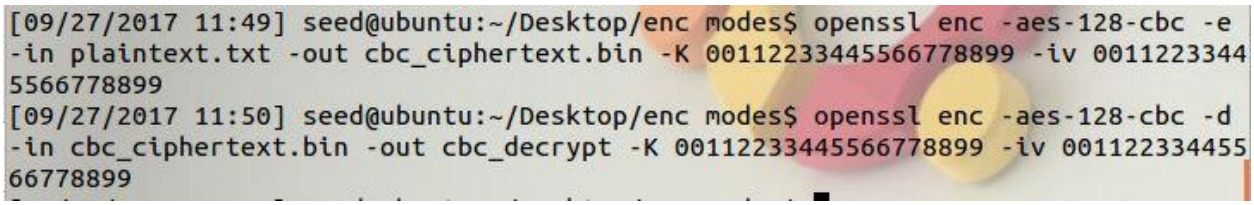
In the original file, the 30th bit is 87 and I changed it to 78. Then I saved the file and it made into a corrupted file.

Then I decrypted the ecb_ciphertext.bin by using -aes-128-ecb into ecb_decrypt. The difference between the plaintext and corrupted text can be seen below.

CBC Encryption mode:

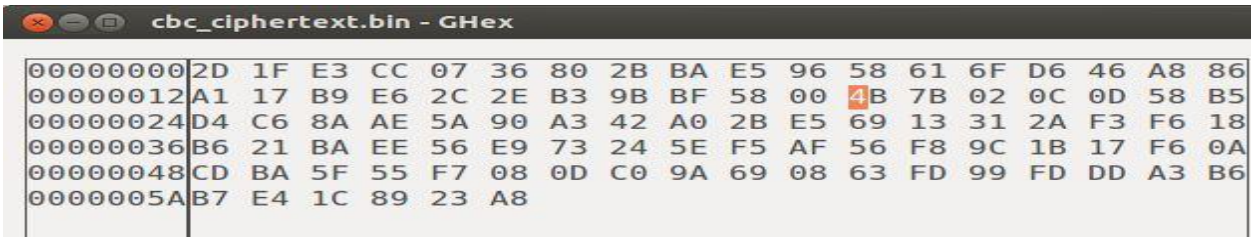
I encrypted the plaintext.txt file into cbc_ciphertext.bin. Then I changed the 30th byte in the cbc_ciphertext.bin by using the GHEX editor and decrypted it into cbc_decrypt.txt.

- Encryption and Decryption



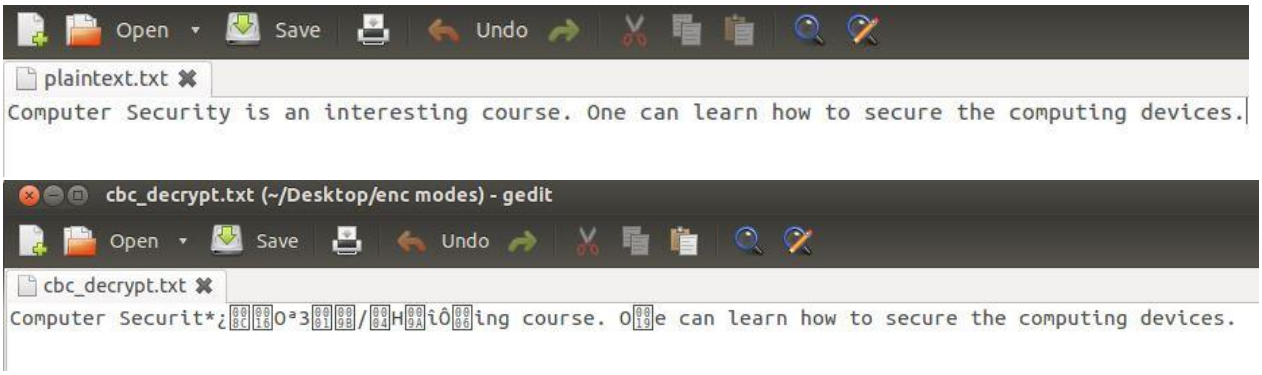
- Original Encrypted file (cbc_ciphertext.bin)

- Corrupted Encrypted file (cbc_ciphertext.bin)



In the original file, the 30th bit is 3C and I changed it to 4B. Then I saved the file and it made into a corrupted file.

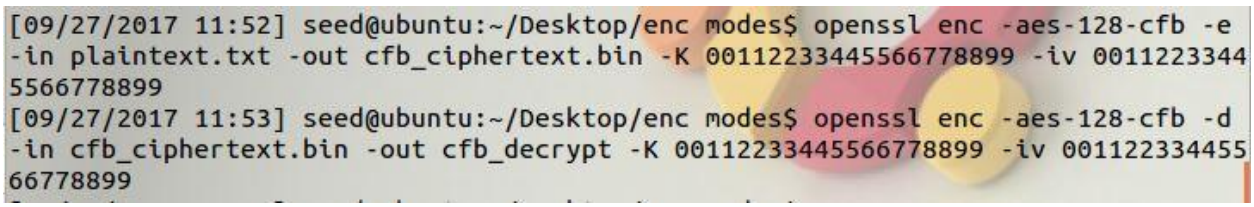
Then I decrypted the cbc_ciphertext.bin by using -aes-128-cbc into cbc_decrypt. The difference between the plaintext and corrupted text can be seen below.



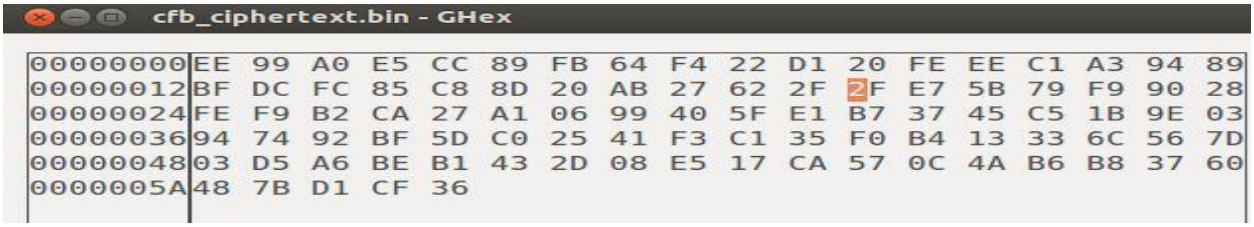
CFB Encryption mode:

I encrypted the plaintext.txt file into cfb_ciphertext.bin. Then I changed the 30th byte in the cfb_ciphertext.bin by using the GHEX editor and decrypted it into cfb_decrypt.txt.

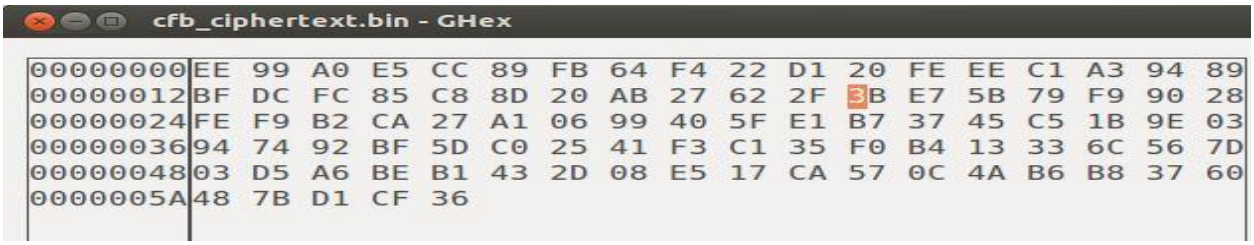
- Encryption and Decryption



- Original Encrypted file (cfb_ciphertext.bin)

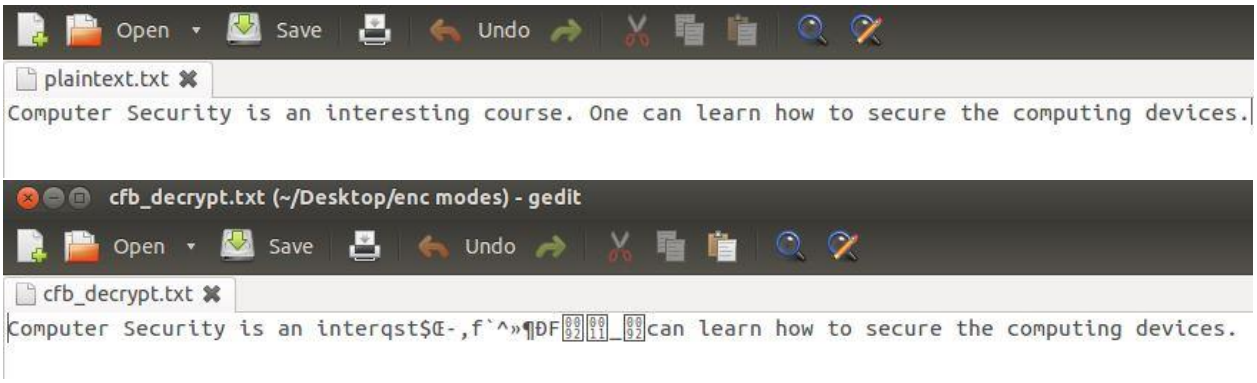


- Corrupted Encrypted file (cfb_ciphertext.bin)



In the original file, the 30th bit is 2F and I changed it to 3B. Then I saved the file and it made into a corrupted file.

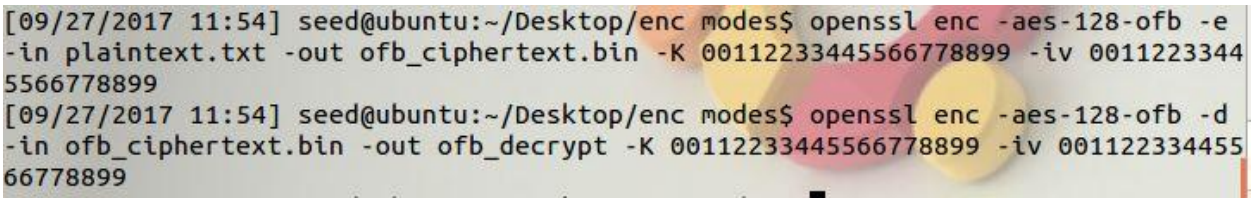
Then I decrypted the cfb_ciphertext.bin by using -aes-128-cfb into cfb_decrypt. The difference between the plaintext and corrupted text can be seen below.



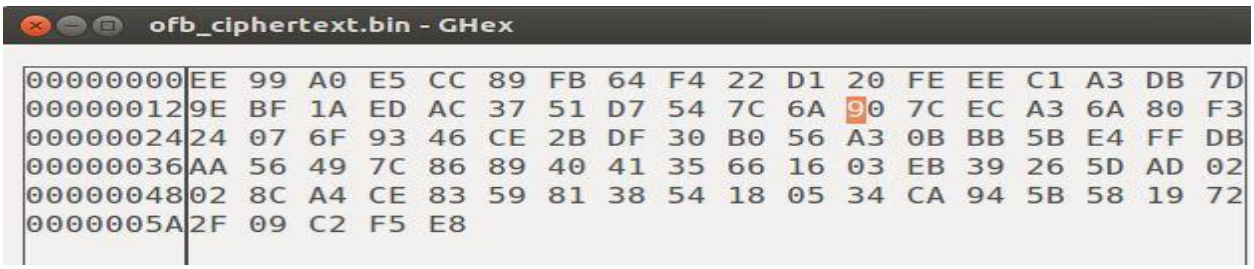
OFB Encryption mode:

I encrypted the plaintext.txt file into ofb_ciphertext.bin. Then I changed the 30th byte in the ofb_ciphertext.bin by using the GHEX editor and decrypted it into ofb_decrypt.txt.

- Encryption and Decryption



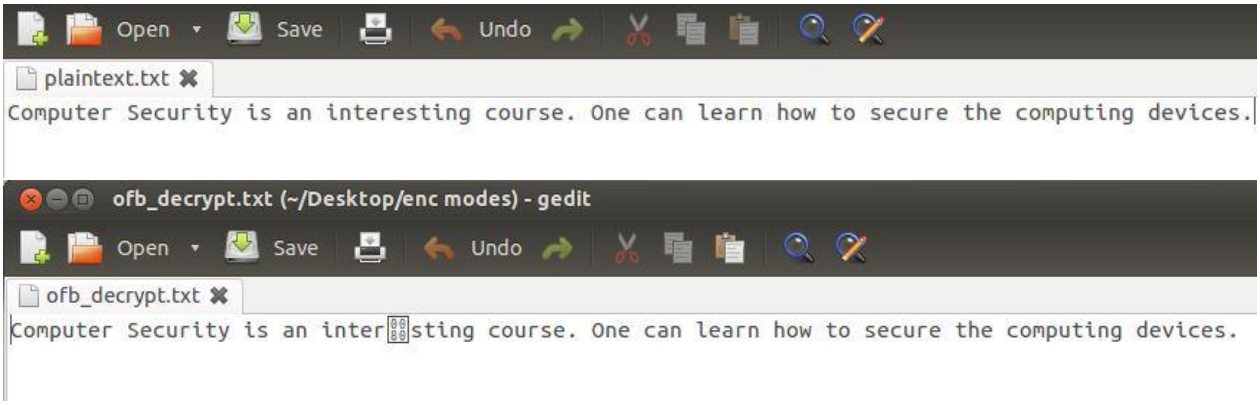
- Original Encrypted file (ofb_ciphertext.bin)



- Corrupted Encrypted file (ofb_ciphertext.bin)

In the original file, the 30th bit is 90 and I changed it to 75. Then I saved the file and it made into a corrupted file.

Then I decrypted the ofb_ciphertext.bin by using -aes-128-ofb into ofb_decrypt. The difference between the plaintext and corrupted text can be seen below.



Observation:

ECB: Only one block is affected but it is spread to the neighbor characters too.

CFB: Only one block is affected but it is spread to the neighbor characters too.

CBC: Almost two blocks are affected which is very inefficient.

OFB: Only one character is corrupted. So only **OFB is giving the most promising result** among all the other modes.

3.2.3) Image encrypted by using rc2-cbc cipher type (smiley.bmp).

Task 4: Padding

The Block cipher algorithms like AES and 3DES in some encryption modes like ECB and CBC require their input to be an exact multiple of block size. If the plaintext to be encrypted is not an exact multiple of the block size, it will add padding to the file before encrypting. While decrypting, the receiving party needs to be known about how to decrypt the padding string from the encrypted file.

Many of the block ciphers use PKCS5 padding. I created two files: plaintext20.txt which is of 20 bytes and plaintext32.txt which is of 32 bytes. These files are encrypted by using aes-128-cbc. So, the block size of 128 bits is 128/8 = 16 bytes.

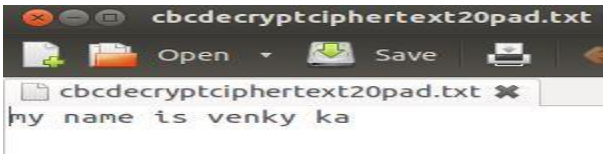
Now I decrypted the encrypted file with default and -nopad option. So normally when I encrypt the data the padding is automatically added to the input file and the encrypted file will be generated. When I decrypt the data, the padding is automatically removed from the encrypted file and a normal file is generated. But, when I use the -nopad option, the padding won't be removed from the encrypted file.

Padding 20 bytes:

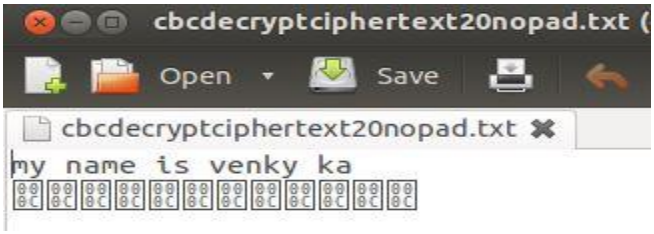
The below picture clearly depicts the commands needed to encrypt and decrypt the data by default and by using nopad option.

- plaintext20.txt

- cbcdecryptciphertext20pad.txt



- cbcdecryptciphertext20nopad.txt



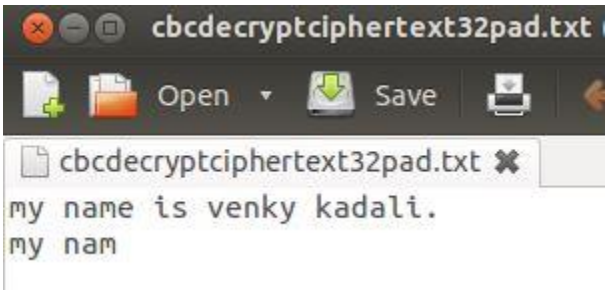
The above file clearly shows that there is a padding which is of 12bytes. The plaintext is of 20 bytes. First block consists of 16 bytes and second block consists of 4 bytes. So, 12 more bytes are needed to become an exact multiple of the block. So, 12 bytes of padding is added to the input file after encryption.

Padding 32 bytes:

The below picture clearly shows the commands needed to encrypt and decrypt the data by default and by using nopad option.

- plaintext32.txt

- cbcdecryptciphertext20pad.txt



- cbcdecryptciphertext20nopad.txt

The above file clearly shows that there is a padding which is of 16 bytes. The plaintext is of 32 bytes. First block consists of 16 bytes and second block consists of 16 bytes. Nevertheless, the padding is needed and to become an exact multiple of the block, 16 more bytes are needed. So, 16 bytes of padding is added to the input file after encryption.

Different Encryption modes:

Padding in 20 bytes:

Padding in ECB: (size of input file increased. Padding is needed.)

Padding in CBC: (size of input file increased. Padding is needed.)

```
[09/27/2017 12:47] seed@ubuntu:~/Desktop/padding20$ openssl enc -aes-128-cbc -e -in plaintext20.txt -out cbc_ciphertext20.txt -K 001122334455667
78899 -iv 00112233445566778899
[09/27/2017 12:48] seed@ubuntu:~/Desktop/padding20$ ls -l
total 8
-rw-rw-r-- 1 seed seed 32 Sep 27 12:48 cbc_ciphertext20.txt
-rw-rw-r-- 1 seed seed 20 Sep 25 22:35 plaintext20.txt
```

Padding in CFB: (size of input file in not changed. Padding is not needed.)

```
[09/27/2017 12:48] seed@ubuntu:~/Desktop/padding20$ openssl enc -aes-128-cfb -e -in plaintext20.txt -out cfb_ciphertext20.txt -K 001122334455667
78899 -iv 00112233445566778899
[09/27/2017 12:49] seed@ubuntu:~/Desktop/padding20$ ls -l
total 16
-rw-rw-r-- 1 seed seed 32 Sep 27 12:48 cbc_ciphertext20.txt
-rw-rw-r-- 1 seed seed 20 Sep 27 12:49 cfb_ciphertext20.txt
-rw-rw-r-- 1 seed seed 32 Sep 27 12:48 ecb_ciphertext20.txt
-rw-rw-r-- 1 seed seed 20 Sep 25 22:35 plaintext20.txt
```

Padding in OFB: (size of input file in not changed. Padding is not needed.)

```
[09/27/2017 12:49] seed@ubuntu:~/Desktop/padding20$ openssl enc -aes-128-ofb -e -in plaintext20.txt -out ofb_ciphertext20.txt -K 001122334455667
78899 -iv 00112233445566778899
[09/27/2017 12:49] seed@ubuntu:~/Desktop/padding20$ ls -l
total 20
-rw-rw-r-- 1 seed seed 32 Sep 27 12:48 cbc_ciphertext20.txt
-rw-rw-r-- 1 seed seed 20 Sep 27 12:49 cfb_ciphertext20.txt
-rw-rw-r-- 1 seed seed 32 Sep 27 12:48 ecb_ciphertext20.txt
-rw-rw-r-- 1 seed seed 20 Sep 27 12:49 ofb_ciphertext20.txt
-rw-rw-r-- 1 seed seed 20 Sep 25 22:35 plaintext20.txt
```

Result:

```
-rw-rw-r-- 1 seed seed 32 Sep 27 12:48 cbc_ciphertext20.txt
-rw-rw-r-- 1 seed seed 20 Sep 27 12:49 cfb_ciphertext20.txt
-rw-rw-r-- 1 seed seed 32 Sep 27 12:48 ecb_ciphertext20.txt
-rw-rw-r-- 1 seed seed 20 Sep 27 12:49 ofb_ciphertext20.txt
-rw-rw-r-- 1 seed seed 20 Sep 25 22:35 plaintext20.txt
```

Observation:

From the above result, padding is needed for ECB and CBC encryption modes. Because the size of ecb_ciphertext20.txt and cbc_ciphertext20.txt are increased after encryption. The reason the inputs contain number of blocks which are not fixed. **The input should be the exact multiple of the block size.** This block size depends on algorithm. AES uses 16-byte blocks and 3DES uses 8-byte blocks.

The size of cfb_ciphertext20.txt and ofb_ciphertext20.txt are not changed after encryption. So, the padding is not added for CFB and OFB encryption modes. The reason is that **they are stream ciphers, in which the size of the block is fixed.**

Padding in 32 bytes:

Padding in ECB: (size of input file increased. Padding is needed.)

```
[09/27/2017 13:07] seed@ubuntu:~/Desktop/padding32$ openssl enc -aes-128-ecb -e -in plaintext32.txt -out ecb_ciphertext32.txt -K 001122334455667
78899 -iv 00112233445566778899
[09/27/2017 13:07] seed@ubuntu:~/Desktop/padding32$ ls -l
total 12
-rw-rw-r-- 1 seed seed 48 Sep 27 13:07 cbc_ciphertext32.txt
-rw-rw-r-- 1 seed seed 48 Sep 27 13:07 ecb_ciphertext32.txt
-rw-rw-r-- 1 seed seed 32 Sep 25 22:36 plaintext32.txt
```

Padding in CBC: (size of input file increased. Padding is needed.)

```
[09/27/2017 13:06] seed@ubuntu:~/Desktop/padding32$ openssl enc -aes-128-cbc -e -in plaintext32.txt -out cbc_ciphertext32.txt -K 001122334455667
78899 -iv 00112233445566778899
[09/27/2017 13:07] seed@ubuntu:~/Desktop/padding32$ ls -l
total 8
-rw-rw-r-- 1 seed seed 48 Sep 27 13:07 cbc_ciphertext32.txt
-rw-rw-r-- 1 seed seed 32 Sep 25 22:36 plaintext32.txt
```

Padding in CFB: (size of input file in not changed. Padding is not needed.)

```
[09/27/2017 13:07] seed@ubuntu:~/Desktop/padding32$ openssl enc -aes-128-cfb -e -in plaintext32.txt -out cfb_ciphertext32.txt -K 001122334455667
78899 -iv 00112233445566778899
[09/27/2017 13:07] seed@ubuntu:~/Desktop/padding32$ ls -l
total 16
-rw-rw-r-- 1 seed seed 48 Sep 27 13:07 cbc_ciphertext32.txt
-rw-rw-r-- 1 seed seed 32 Sep 27 13:07 cfb_ciphertext32.txt
-rw-rw-r-- 1 seed seed 48 Sep 27 13:07 ecb_ciphertext32.txt
-rw-rw-r-- 1 seed seed 32 Sep 25 22:36 plaintext32.txt
```

Padding in OFB: (size of input file in not changed. Padding is not needed.)

```
[09/27/2017 13:07] seed@ubuntu:~/Desktop/padding32$ openssl enc -aes-128-ofb -e -in plaintext32.txt -out ofb_ciphertext32.txt -K 001122334455667
78899 -iv 00112233445566778899
[09/27/2017 13:08] seed@ubuntu:~/Desktop/padding32$ ls -l
total 20
-rw-rw-r-- 1 seed seed 48 Sep 27 13:07 cbc_ciphertext32.txt
-rw-rw-r-- 1 seed seed 32 Sep 27 13:07 cfb_ciphertext32.txt
-rw-rw-r-- 1 seed seed 48 Sep 27 13:07 ecb_ciphertext32.txt
-rw-rw-r-- 1 seed seed 32 Sep 27 13:08 ofb_ciphertext32.txt
-rw-rw-r-- 1 seed seed 32 Sep 25 22:36 plaintext32.txt
```

Result:

```
-rw-rw-r-- 1 seed seed 48 Sep 27 13:07 cbc_ciphertext32.txt
-rw-rw-r-- 1 seed seed 32 Sep 27 13:07 cfb_ciphertext32.txt
-rw-rw-r-- 1 seed seed 48 Sep 27 13:07 ecb_ciphertext32.txt
-rw-rw-r-- 1 seed seed 32 Sep 27 13:08 ofb_ciphertext32.txt
-rw-rw-r-- 1 seed seed 32 Sep 25 22:36 plaintext32.txt
```

Observation:

From the above result, padding is added for ECB and CBC encryption modes. As we can see that the size of ecb_ciphertext32.txt and cbc_ciphertext32.txt are increased after encryption. The reason is that the inputs contain number of blocks which are not fixed. **The input should be the**

exact multiple of the block size. This block size depends on algorithm. AES uses 16-byte blocks and 3DES uses 8-byte blocks.

The size of cfb_ciphertext32.txt and ofb_ciphertext32.txt are not changed after encryption. So, the padding is not added for CFB and OFB encryption modes. The reason is that **they are stream ciphers, in which the size of the block is fixed.**

Task 5: Programming using the Crypto Library

I took the given input plaintext and cipher text. I wrote a program which finds the key encrypted with the plaintext to make it a cipher test.

KEY: median

The program scans all the words in the dictionary which is given as an input and add padding to every word by using the padding method and checks whether it is present in the cipher text or not. Finally, I found that median is the key that is encrypted with the given plaintext.

Task 6.A:

I run this command: “cat /proc/sys/kernel/random/entropy_avail” for several times. Some random values are generating when I run that command without blocking. But when I type something on the terminal or click the mouse on the terminal or interrupting the hardware by running a program and run that command, I found the larger entropy value. The below image clearly depicts the observation.

Observation:

So, hence proved that the Linux gains the randomness from the physical resources like keyboard, mouse, interrupts and blkdev.

Task 6.B:

I run this command: “head -c 16 /dev/random | hexdump” for several times (nearly 10 times). I found that /dev/random is blocking in between. It is waiting for some time and then giving another random number. **But when I start moving the mouse or using keyboard in terminal it is not blocking.** So, by using some physical resources like keyboard, mouse, interrupt and, blkdev we can stop blocking.

Task 6.C: Get Random Numbers from /dev/urandom

I run this command: head -c 1600 /dev/urandom | hexdump for several times (30 to 40 times). I found that /dev/urandom is not blocking unlike /dev/random. Always, 1600 bytes of random numbers are generating.