

SECRET-KEY ENCRYPTION LAB

Table of Contents

| | |
|--|-----------|
| 2.1. ENCRYPTION USING DIFFERENT CIPHERS AND MODES | 1 |
| <i>i. AES 128 CBC Task 1</i> | 1 |
| a. Encryption | 1 |
| Create Directory | 1 |
| A new txt File in the Directory Lab2 | 2 |
| The plaintext File | 2 |
| Command to Encrypt | 3 |
| The Encrypted Text (Cipher) | 3 |
| a. Decryption | 4 |
| AES-128-CFB Task2 | 4 |
| a. Decrypting with CFB command | 5 |
| Executing Decryption Command for CFB command | 6 |
| ii. AES-128-OFB Task3 | 7 |
| a. Encryption | 7 |
| b. Decryption | 9 |
| iii. Blow Fish CBC EncryptionTask3 | 10 |
| a. Encryption | 10 |
| b. Decryption | 12 |
| 2.2. ENCRYPTION MODES ECB vs CBC | 14 |
| a. Image Encryption in ECB | 14 |
| Image to be Encrypted | 14 |
| Image Encryption | 14 |
| Original Image in Bless Editor | 15 |
| Encrypted Image in Bless Editor | 15 |
| Byte Replacement | 16 |
| Encrypted Image After Byte Replacement | 16 |
| b. Image Encryption in CBC | 17 |
| Image for CBC Encryption: | 17 |
| The Code for CBC Encryption | 17 |
| Original Image Byte View in Bless Editor | 18 |
| Encrypted Image Byte View in Bless Edito | 18 |
| Byte Replacement | 19 |
| Image After Byte Replacement | 19 |
| 2.3. ENCRYPTION MODES ON CORRUPTED FILES | 20 |
| a. Encryption Mode: ECB | 20 |
| Plaintext for Encryption in ECB | 20 |
| Encrypted Message in ECB | 20 |
| Encrypted Message in Bless Editor | 21 |
| Byte Replacement | 21 |
| Decryption of the Encrypted Message | 22 |
| b. Encryption Mode: CBC | 23 |
| Plaintext for CBC Encryption | 23 |
| Encryption in CBC | 23 |
| CBC Encrypted File in Bless Editor | 24 |
| Byte Replacement in Bless Editor | 24 |
| Decrypting the Corrupted File | 25 |
| c. Encryption Mode: CFB | 26 |
| The Plaintext | 26 |
| CFB Encryption of the Plaintext | 26 |
| Encrypted File in Bless Editor | 27 |
| Byte Replacement in Bless Editor | 27 |
| Decrypting the Corrupted File | 28 |
| d. Encryption Mode: OFB | 29 |

| | | |
|--|-----------|-----------------------|
| Secret-key Encryption. | ii | M. Sajid Iqbal |
| Plaintext for Encryption in OFB | 29 | |
| Encryption in OFB | 29 | |
| Encrypted File in Bless Editor | 30 | |
| Byte Replacement in Bless Editor | 30 | |
| Decrypting the Corrupted File | 31 | |
| 2.4. PADDING | 32 | |
| a. 20 Byte File Encryption | 32 | |
| The Plaintext of 20 Byte | 32 | |
| Encrypting 20 Byte File in ECB | 32 | |
| Encrypting 20 Byte File in CBC | 33 | |
| Encrypting 20 Byte File in CFB | 33 | |
| Encrypting 20 Byte File in OFB | 34 | |
| Decrypting 20 Byte File in CBC | 34 | |
| b. 32 Byte File Encryption | 35 | |
| The Plaintext of 32 Byte | 35 | |
| Encrypting 32 Byte File in ECB | 35 | |
| Encrypting 32 Byte File in CBC | 36 | |
| Encrypting 32 Byte File in CFB | 36 | |
| Encrypting 32 Byte File in OFB | 37 | |
| 2.5. Pseudo Random Number Generation | 38 | |
| a. Measure the Entropy of Kernel | 38 | |
| b. Pseudo Random Numbers from /dev/random | 38 | |
| c. Get Random Numbers from /dev/urandom | 39 | |
| 2.6. Programming using the Crypto Library | 40 | |
| a. The Available Files | 40 | |
| b. Running the Command | 41 | |
| c. Program Output | 42 | |
| d. Actual Program: | 43 | |

Lab 2

Secret Key Encryption



SECRET-KEY ENCRYPTION LAB

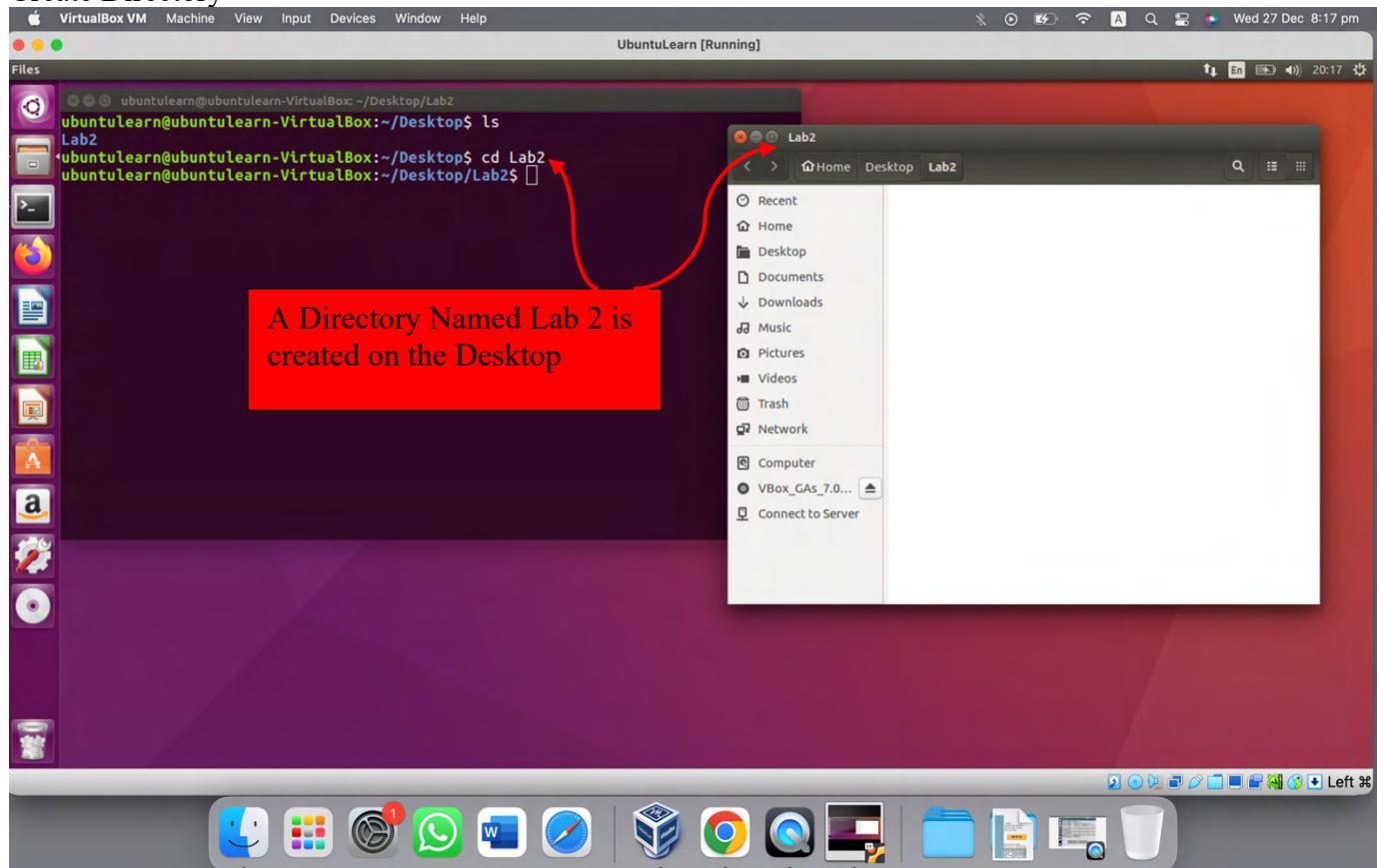
2.1. ENCRYPTION USING DIFFERENT CIPHERS AND MODES

i. AES 128 CBC Task 1

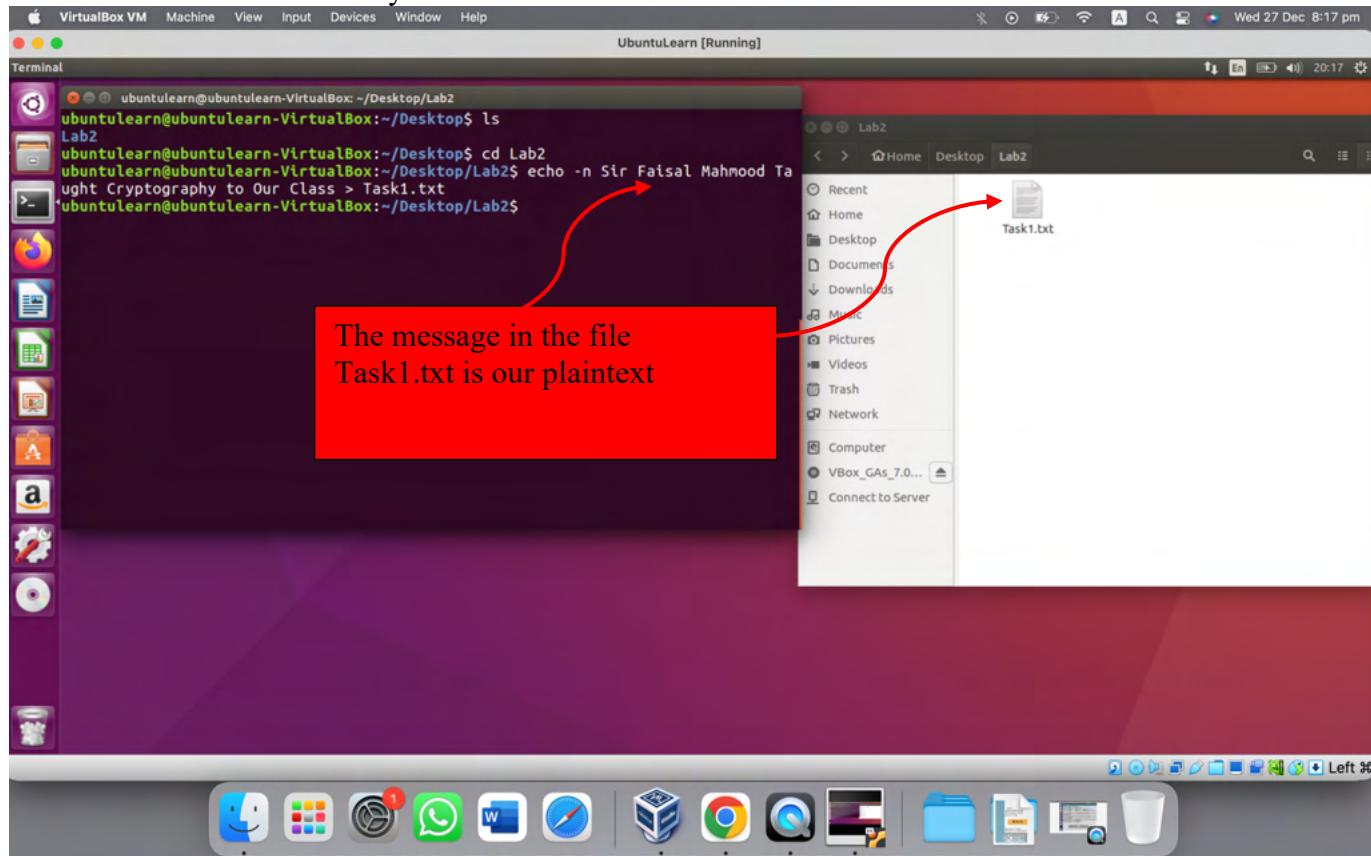
In this mode of encryption, AES Key of 128 bit (16 Bytes) and encryption mode of Cipher Block Chaining has been used. They Key value used for the process is: 123a5678b012c456789f12d4567f9d12 and the initial vector (iv) is: 12e4567c12d45d78.

a. Encryption

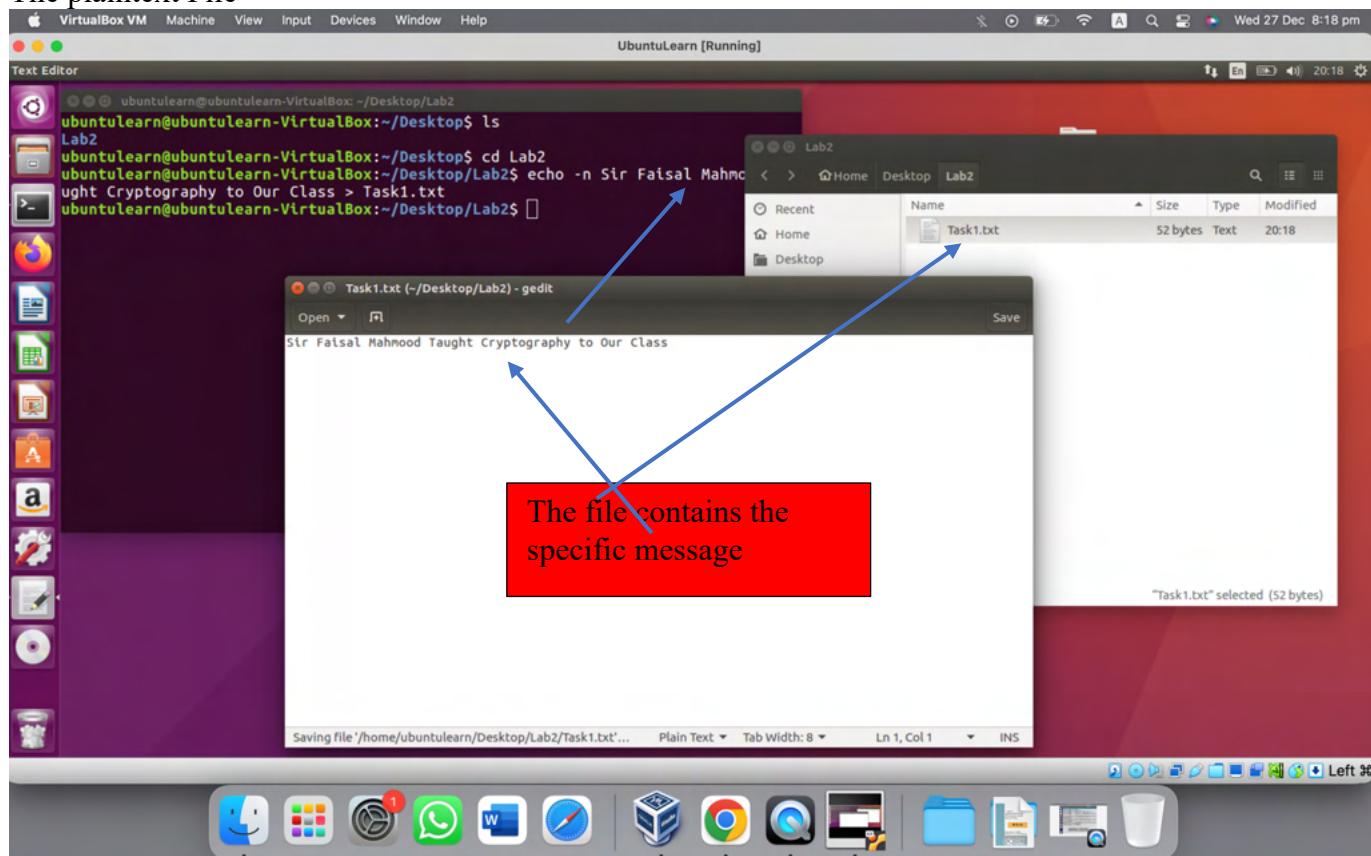
Create Directory



A new txt File in the Directory Lab2



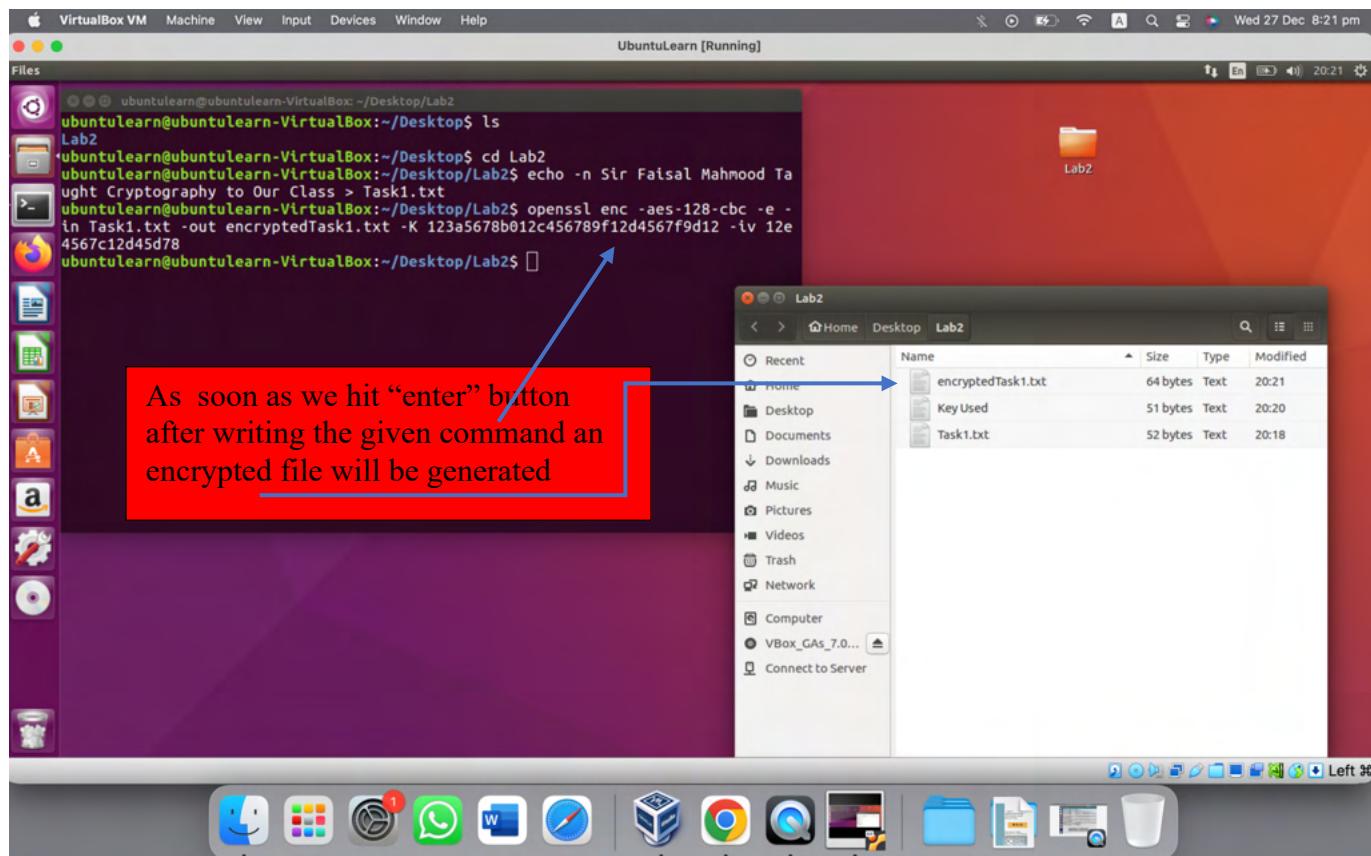
The plaintext File



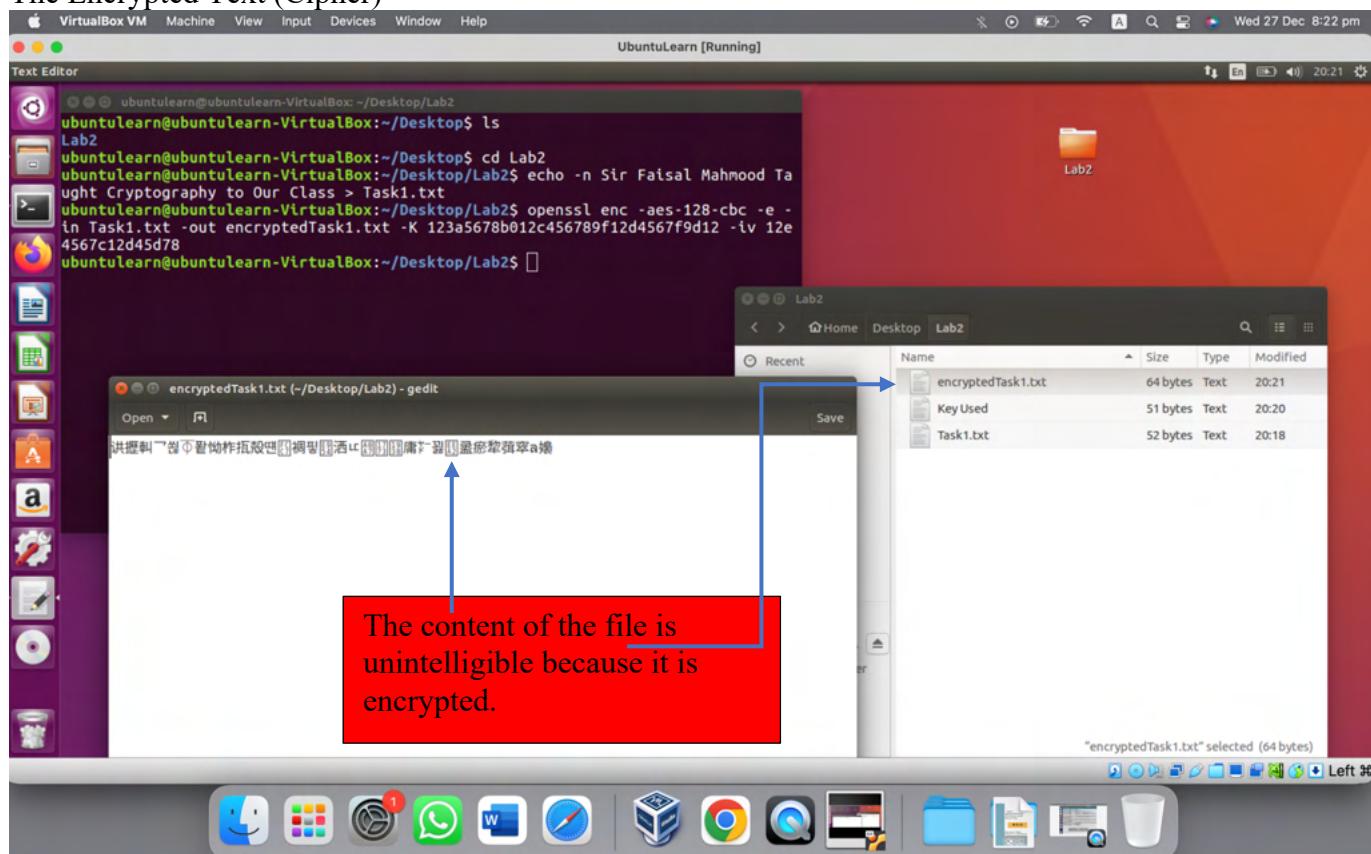
Command to Encrypt

Following command is used to encrypt in CBC mode

```
openssl enc -aes-128-cbc -e -in Task1.txt -out encryptedTask1.txt -K 123a5678b012c456789f12d4567f9d12 -iv 12e4567c12d45d78
```



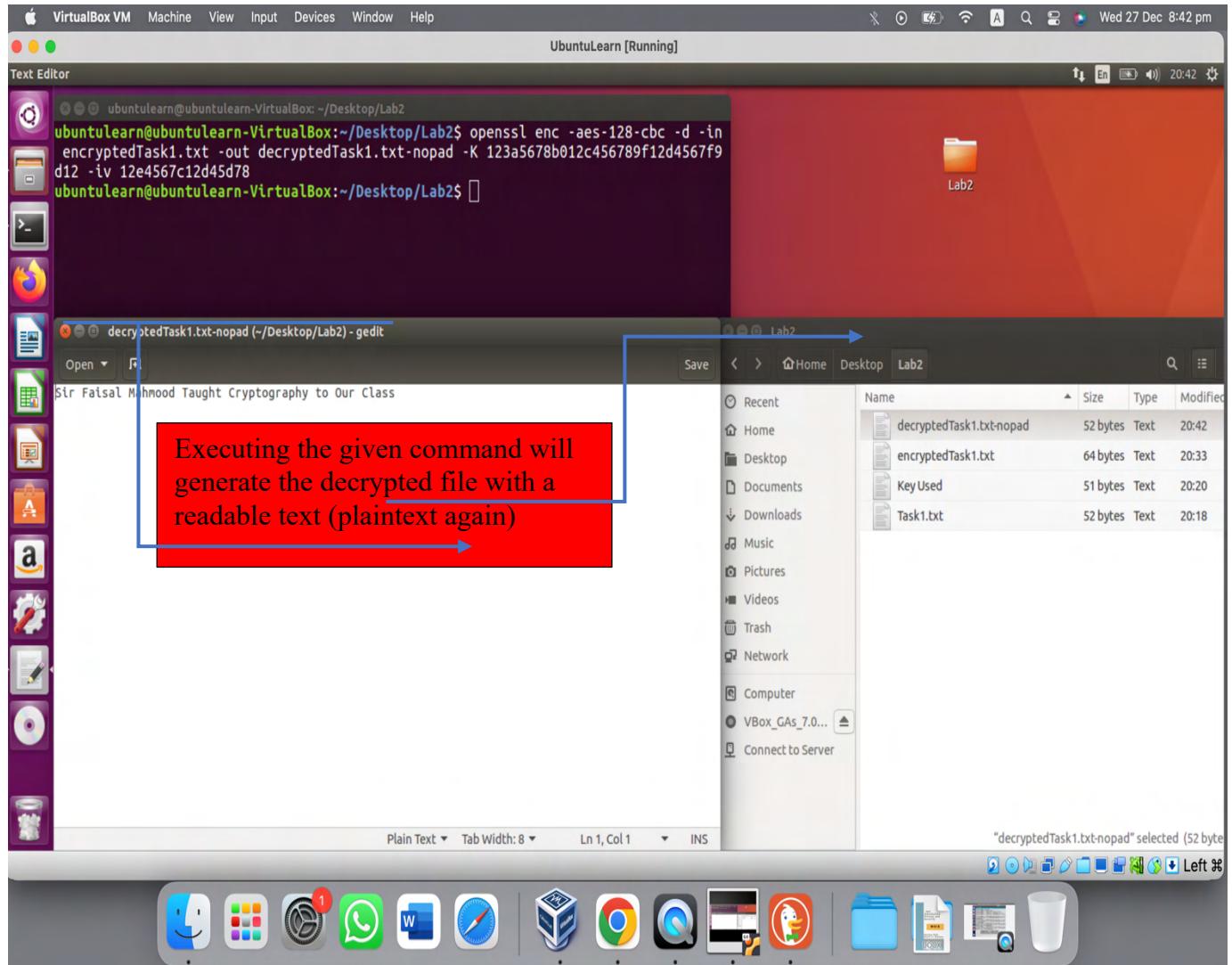
The Encrypted Text (Cipher)



a. Decryption

Following command is used to decrypt in CBC mode

```
openssl enc -aes-128-cbc -d -in encryptedTask1.txt -out decryptedTask1.txt -K 123a5678b012c456789f12d4567f9d12 -iv 12e4567c12d45d78
```



AES-128-CFB Task2

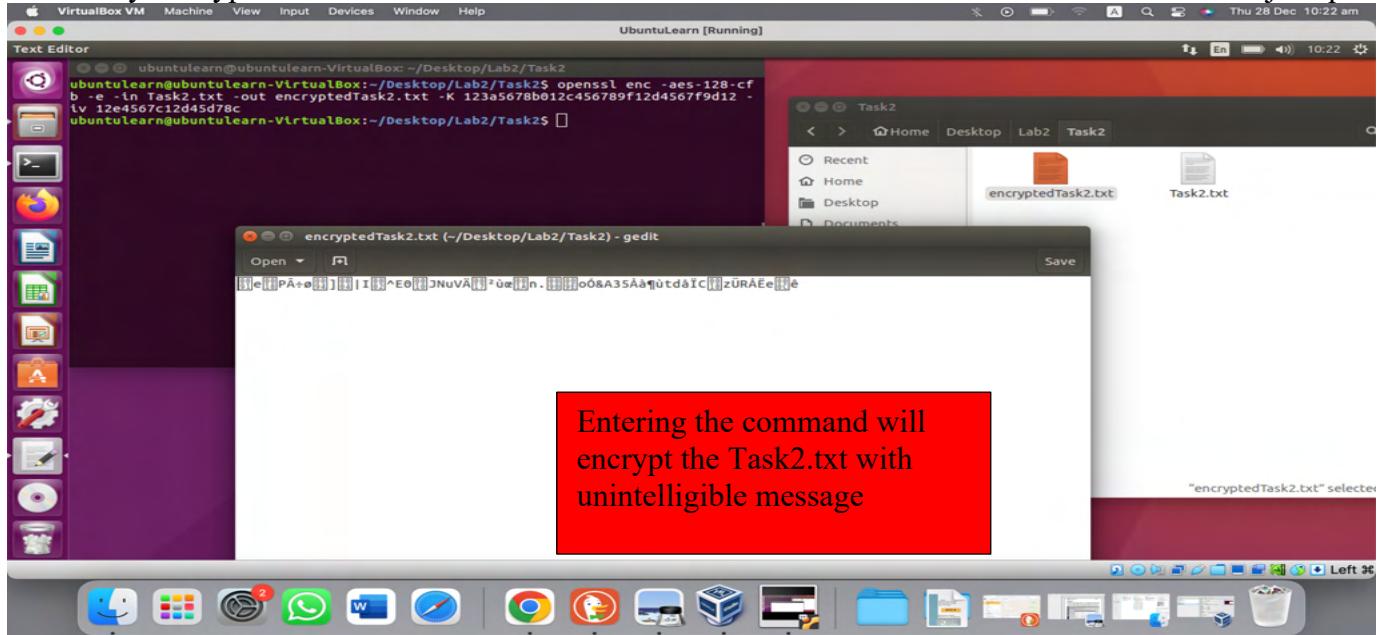
Encryption:

Here we will use Cipher type as AES, Key size = 128 bit (16 bytes), encryption mode as Cipher Feedback (CFB), which is a self-synchronizing stream cipher. Her key value used is

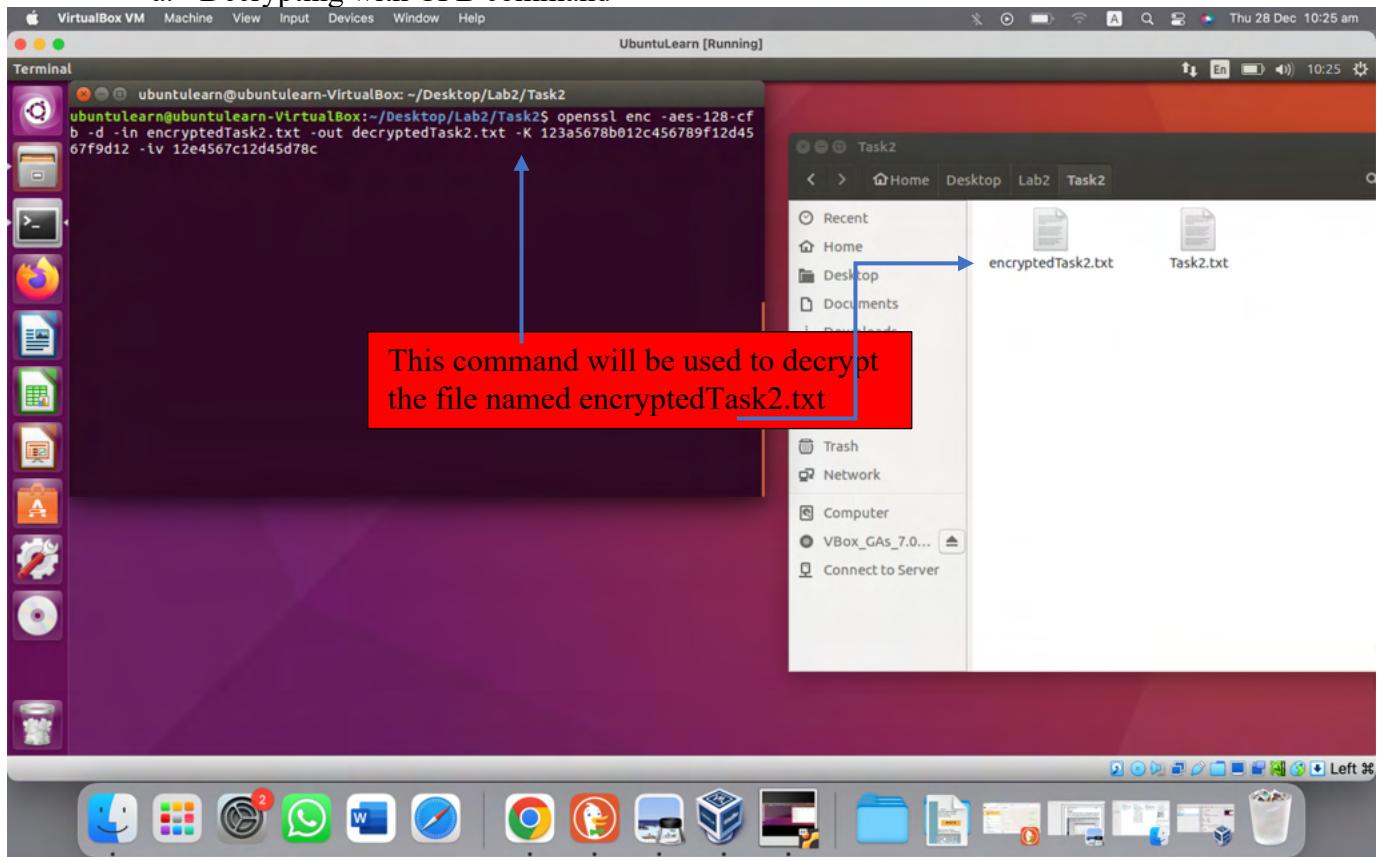
123a5678b012c456789f12d4567f9d12 and initialization vector used is 12e4567c12d45d78. A file Task2.txt with a plaintext will be created in the directory and on running the command for CFB mode we will get the cipher taxt.

Encrypting with CFB command

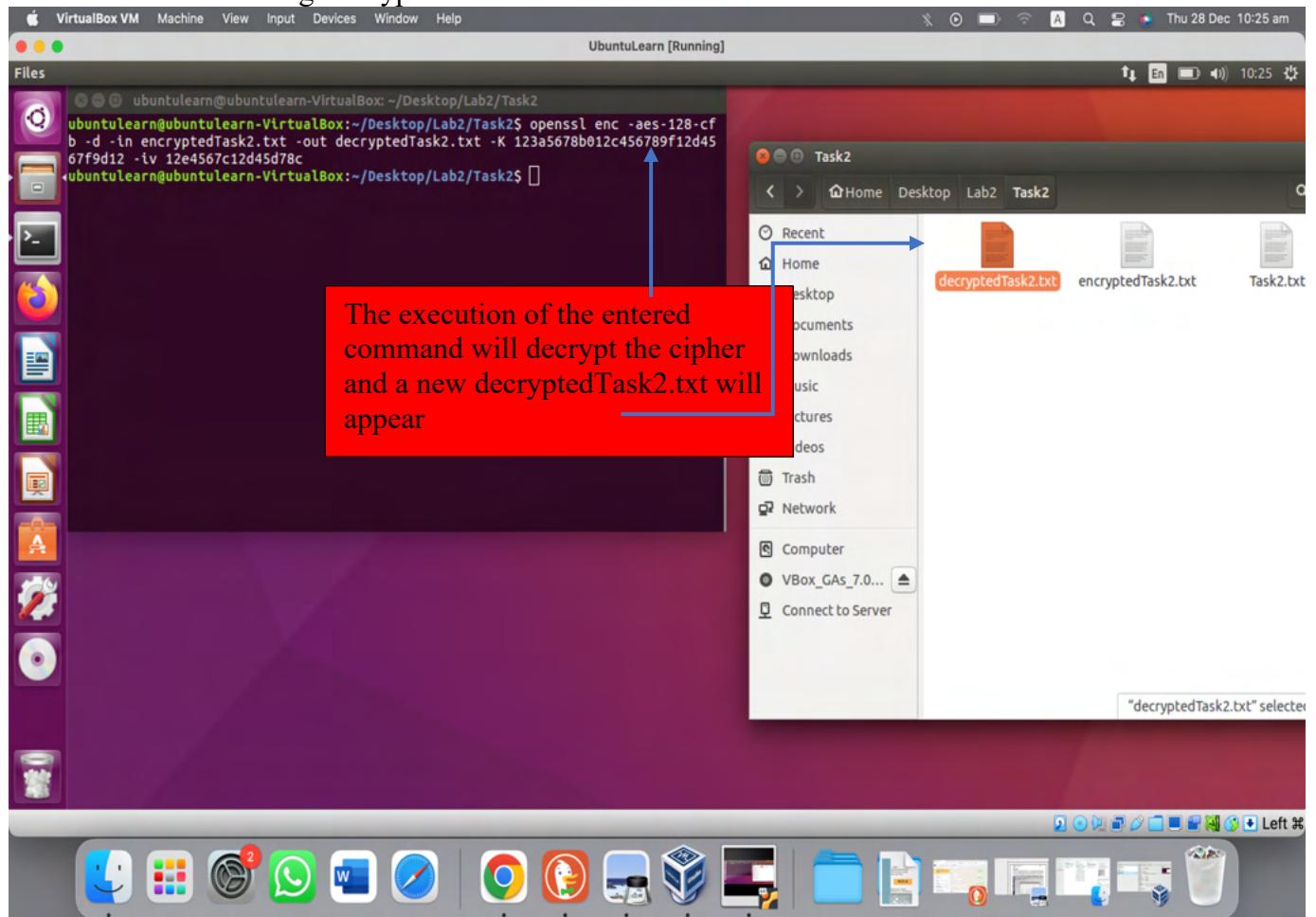
```
openssl enc -aes-128-cfb -e -in encryptedTask2.txt -out decryptedTask2.txt -K 123a5678b012c456789f12d4567f9d12 -iv 12e4567c12d45d78c
```



a. Decrypting with CFB command



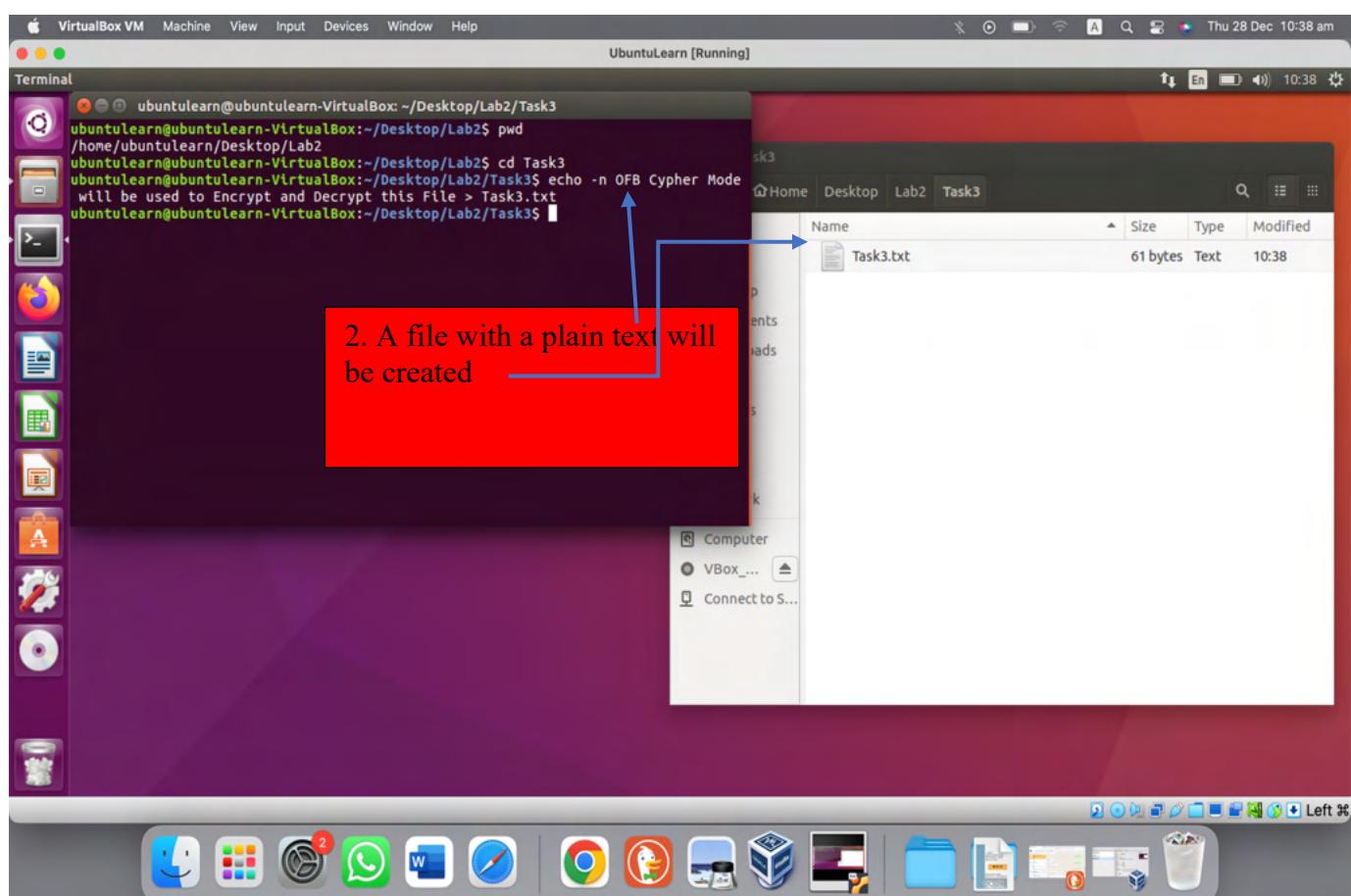
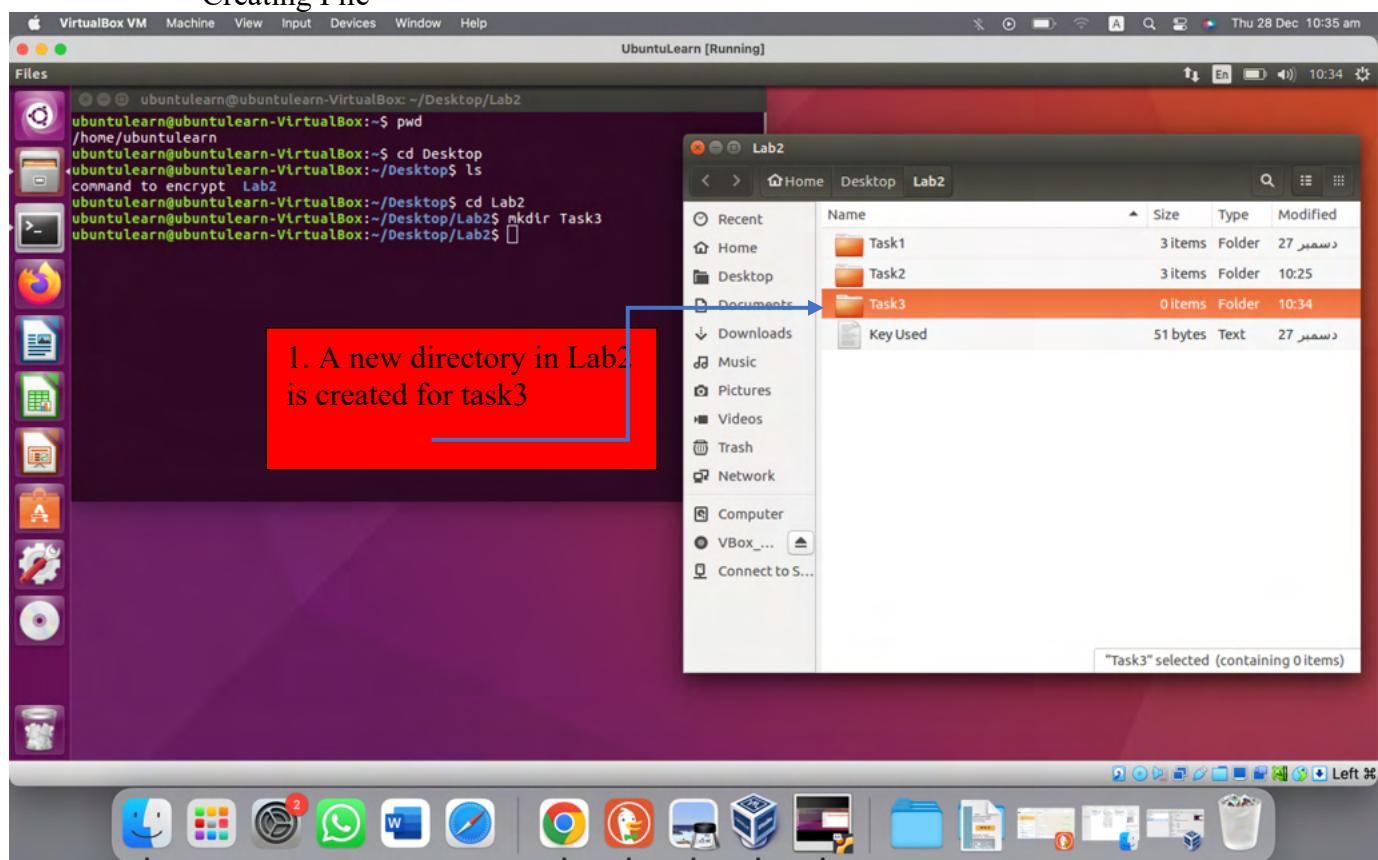
Executing Decryption Command for CFB command



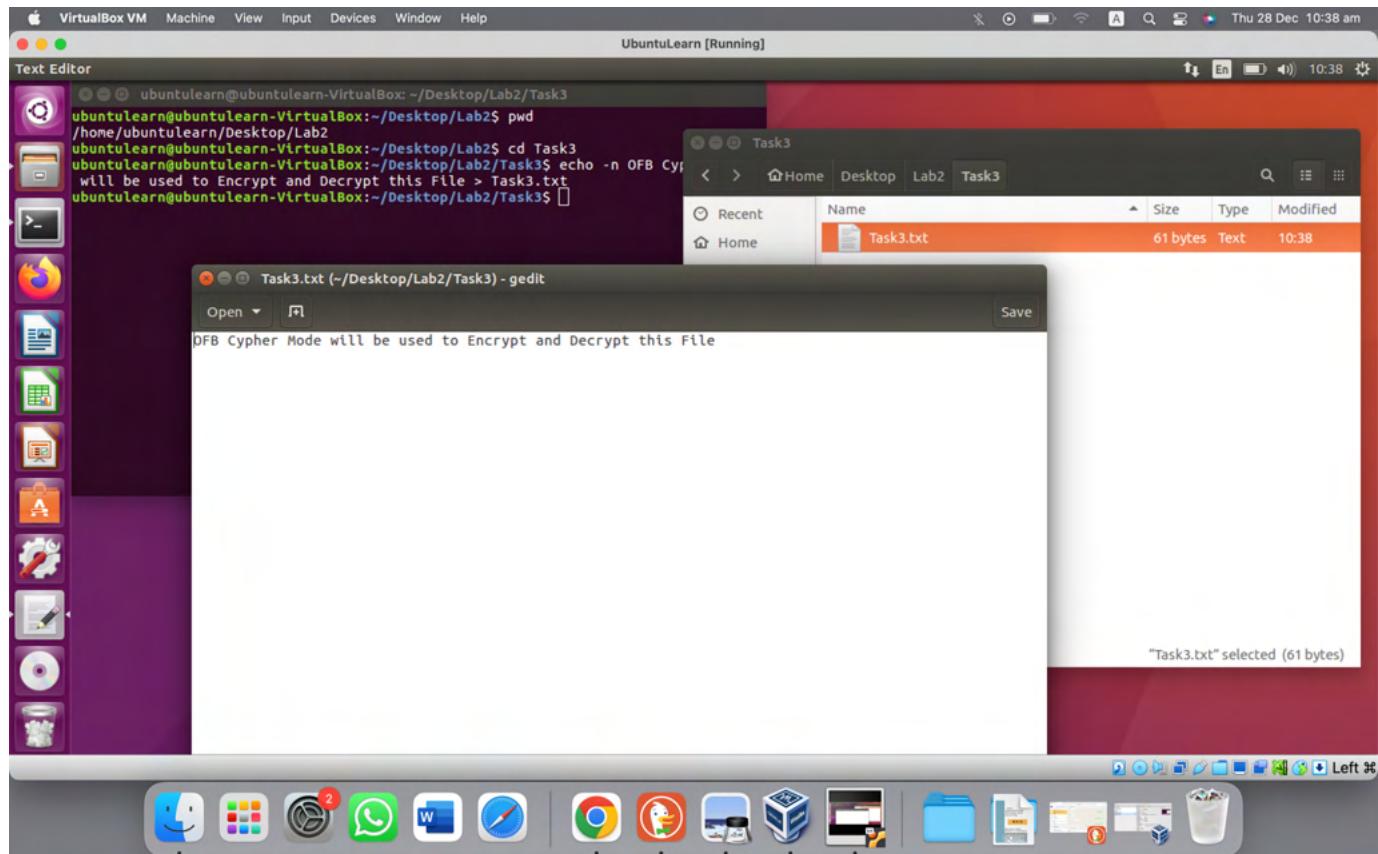
ii. AES-128-OFB Task3

a. Encryption

Creating File

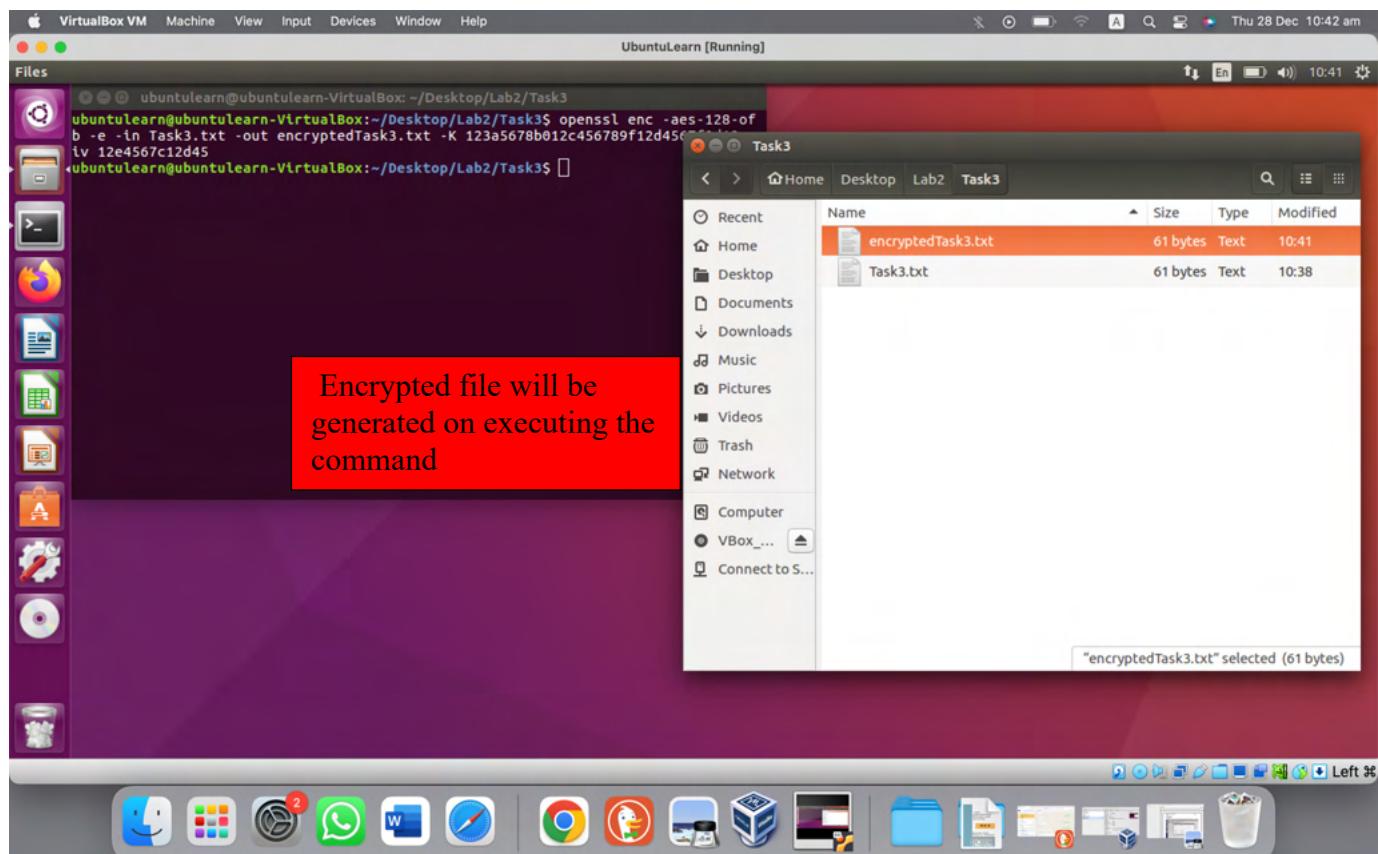


The Plaintext in the .txt File



Encrypting the Plaintext

```
openssl enc -aes-128-cbc -e -in Task3.txt -out encryptedTask3.txt -K 123a5678b012c456789f12d4567f9d12 -iv 12e4567c12d45d78
```



b. Decryption

The cipher Text

Decrypting the Message

The screenshot shows a macOS desktop environment with a purple gradient background. At the top is a dark grey menu bar with the title "UbuntuLearn [Running]" and various system icons. Below the menu bar is a dock with several application icons. On the left, there's a vertical dock of icons for the Dock, Finder, Mail, Safari, and other utilities.

A terminal window titled "Terminal" is open in the foreground, displaying the following command and output:

```
ubuntulearn@ubuntulearn-VirtualBox:~/Desktop/Lab2/Task3$ openssl enc -aes-128-of  
b -e -in Task3.txt -out encryptedTask3.txt -K 123a5678b012c456789f12d45679d12 -  
iv 12e4567c12d45  
ubuntulearn@ubuntulearn-VirtualBox:~/Desktop/Lab2/Task3$ openssl enc -aes-128-of  
b -d -in encryptedTask3.txt -out decryptedTask3.txt -K 123a5678b012c456789f12d45  
67f9d12 -iv 12e4567c12d45  
ubuntulearn@ubuntulearn-VirtualBox:~/Desktop/Lab2/Task3$
```

To the right of the terminal is a file browser window titled "Task3". The sidebar shows "Home", "Desktop", "Lab2", and "Task3". The main pane lists three files:

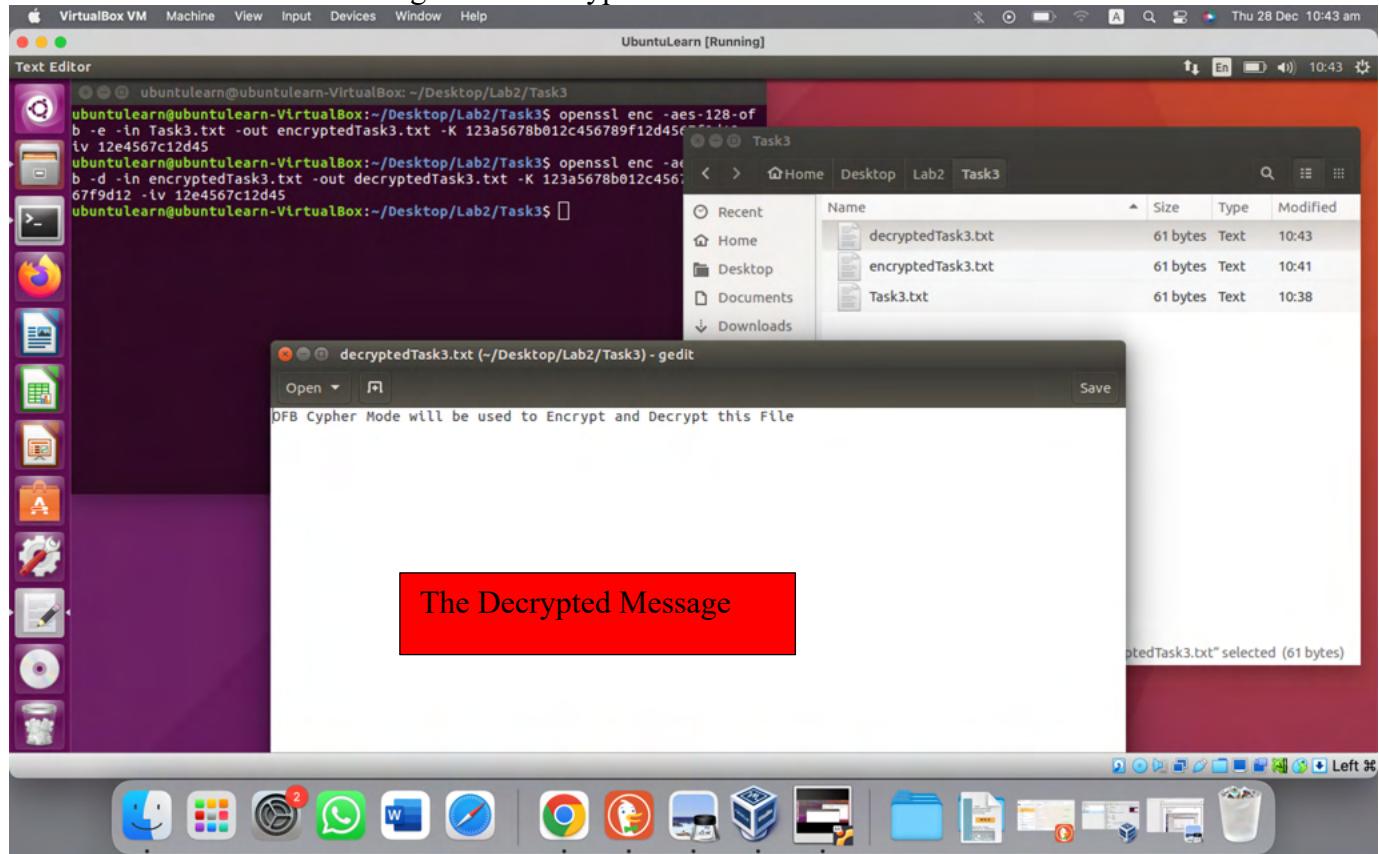
| Name | Size | Type | Modified |
|--------------------|----------|------|----------|
| decryptedTask3.txt | 61 bytes | Text | 10:43 |
| encryptedTask3.txt | 61 bytes | Text | 10:41 |
| Task3.txt | 61 bytes | Text | 10:38 |

A red callout box with a black border and white text points to the "decryptedTask3.txt" file in the list, containing the text:

Entering the command a
decrypted file will appear

At the bottom of the screen, a status bar displays "Left 30" and several small icons.

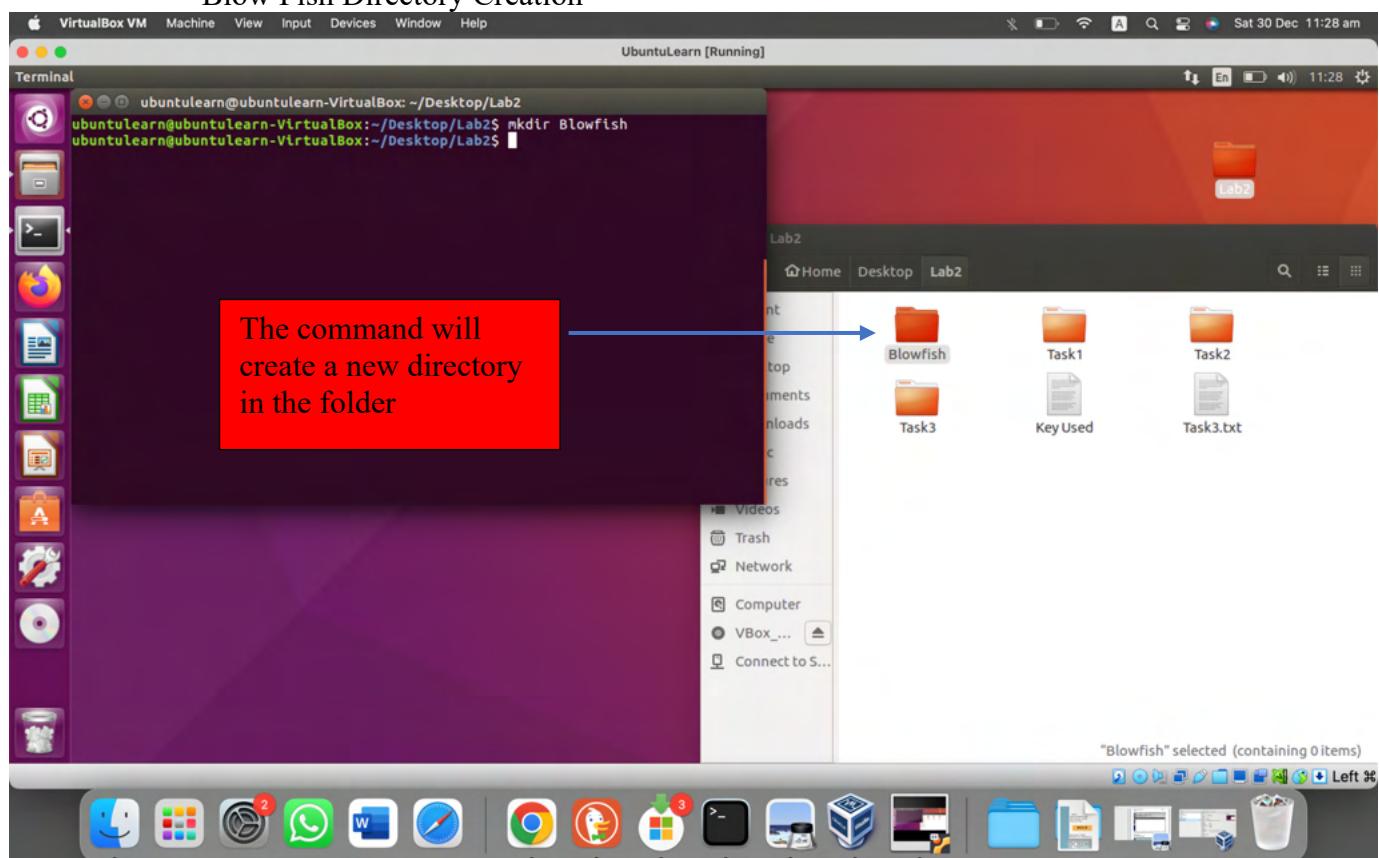
Readable Message After Decryption



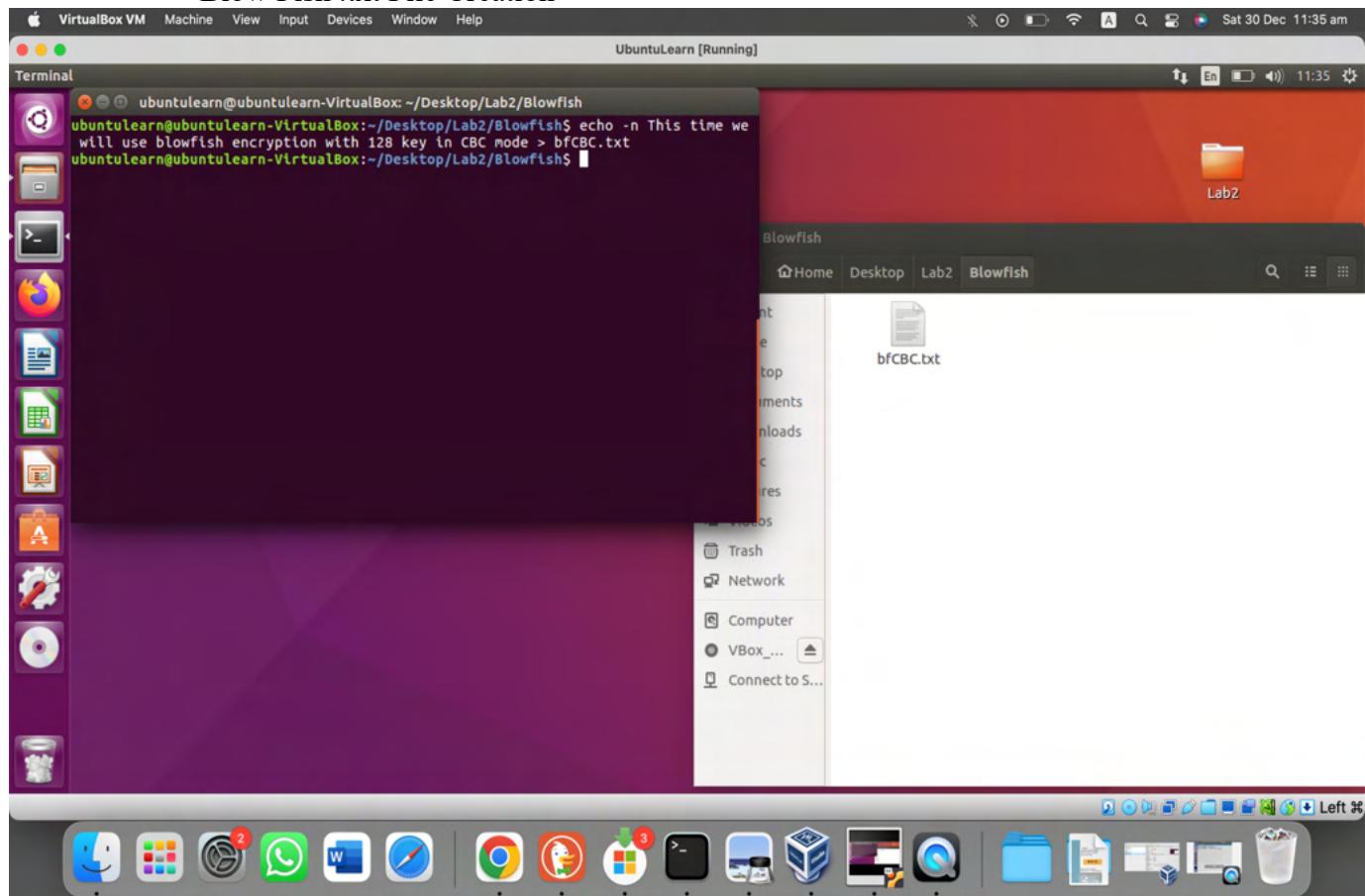
iii. Blow Fish CBC Encryption Task3

a. Encryption

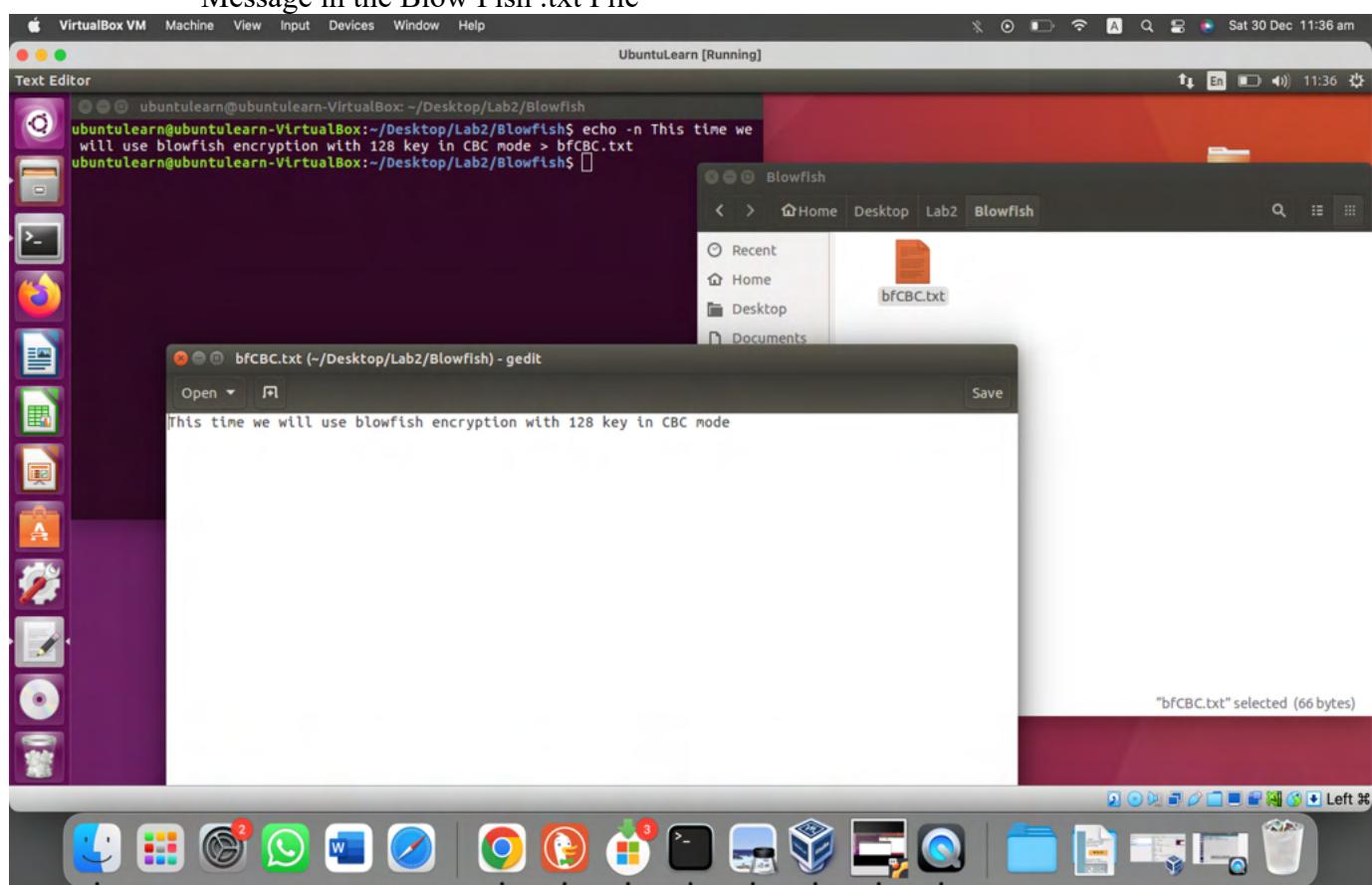
Blow Fish Directory Creation



Blow Fish .txt File Creation



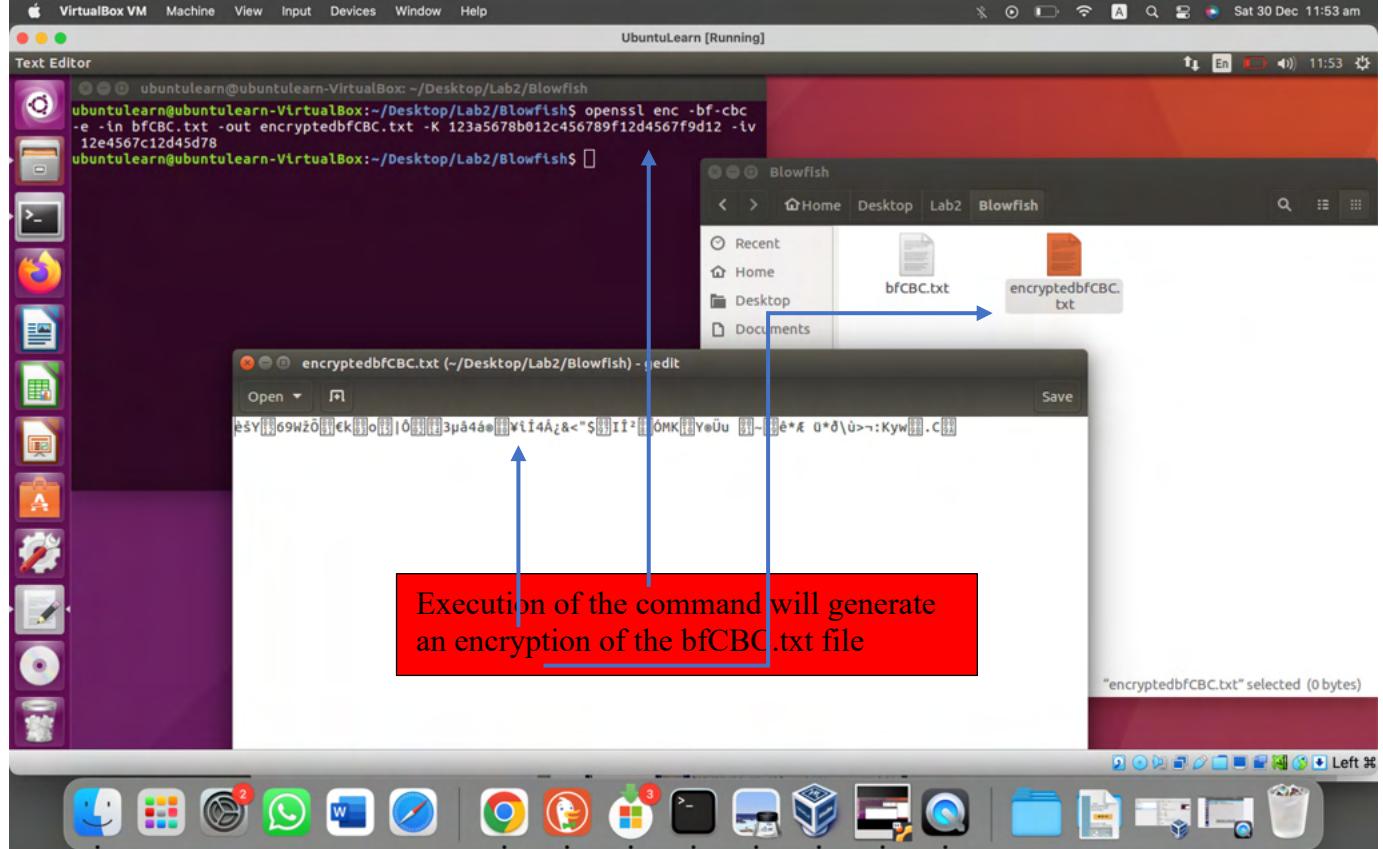
Message in the Blow Fish .txt File



Blow Fish Encryption Command

Following command will be used for Blow Fish encryption:

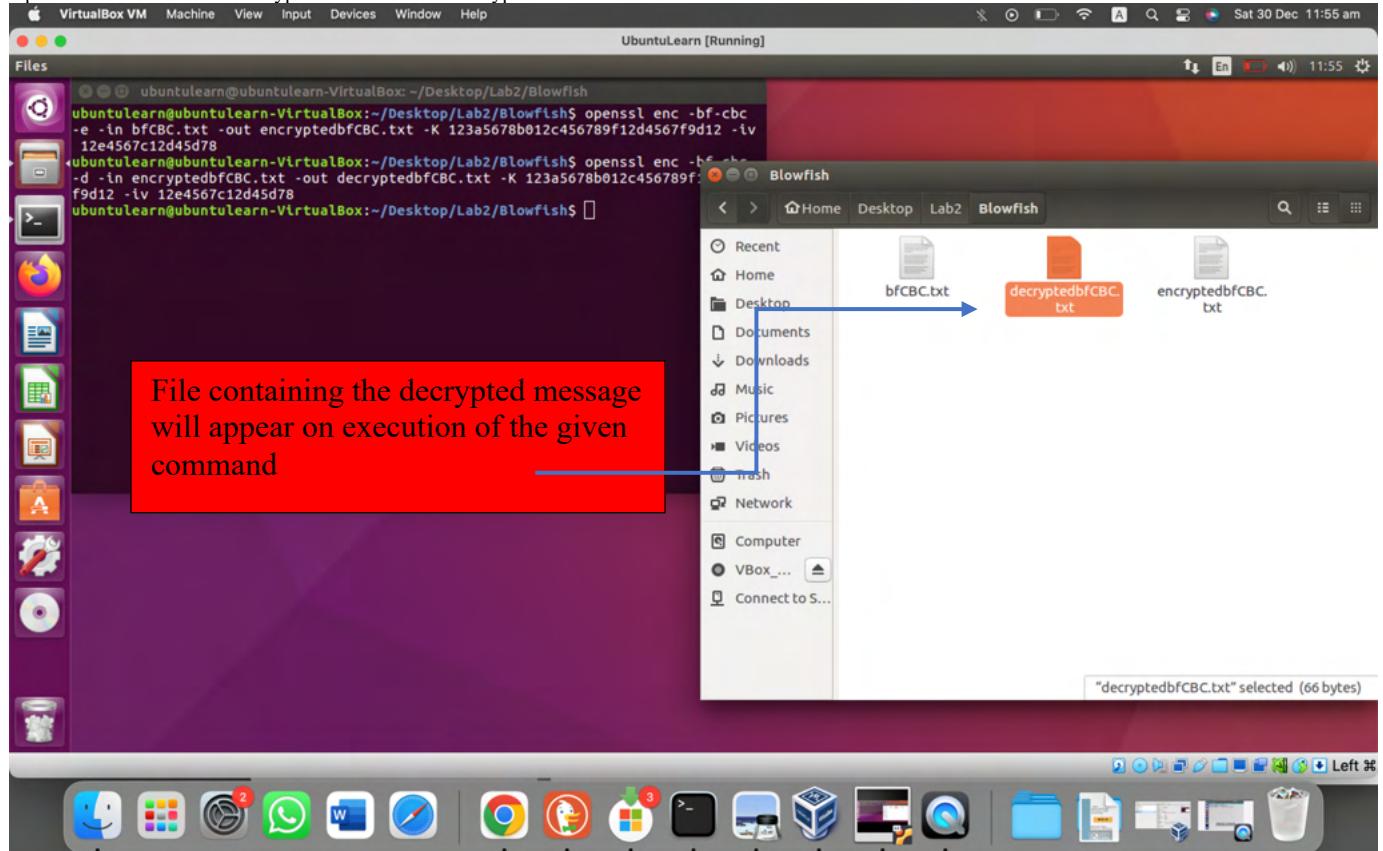
```
Openssl enc -bf-cbc -e -in bfCBC.txt -out encryptedbfCBC.txt -K 123a5678b012c456789f12d4567f9d12 -iv 12e4567c12d45d78
```



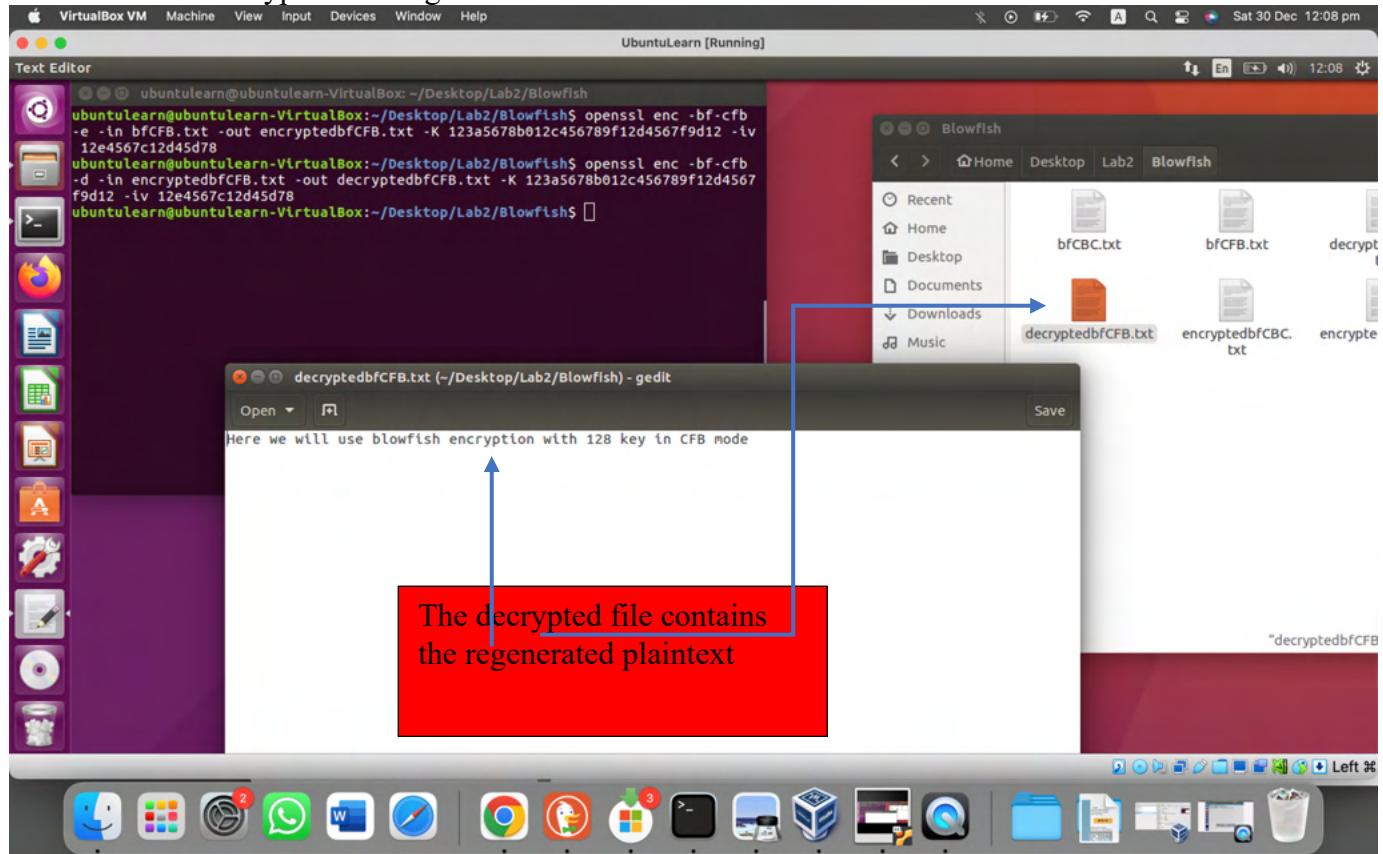
b. Decryption

Command to Decrypt in Blow Fish

```
Openssl enc -bf-cbc -d -in encryptedbfCBC.txt -out decryptedbfCBC.txt -K 123a5678b012c456789f12d4567f9d12 -iv 12e4567c12d45d78
```



Decrypted Message in Blow Fish



2.2. ENCRYPTION MODES ECB vs CBC

a. Image Encryption in ECB

Image to be Encrypted

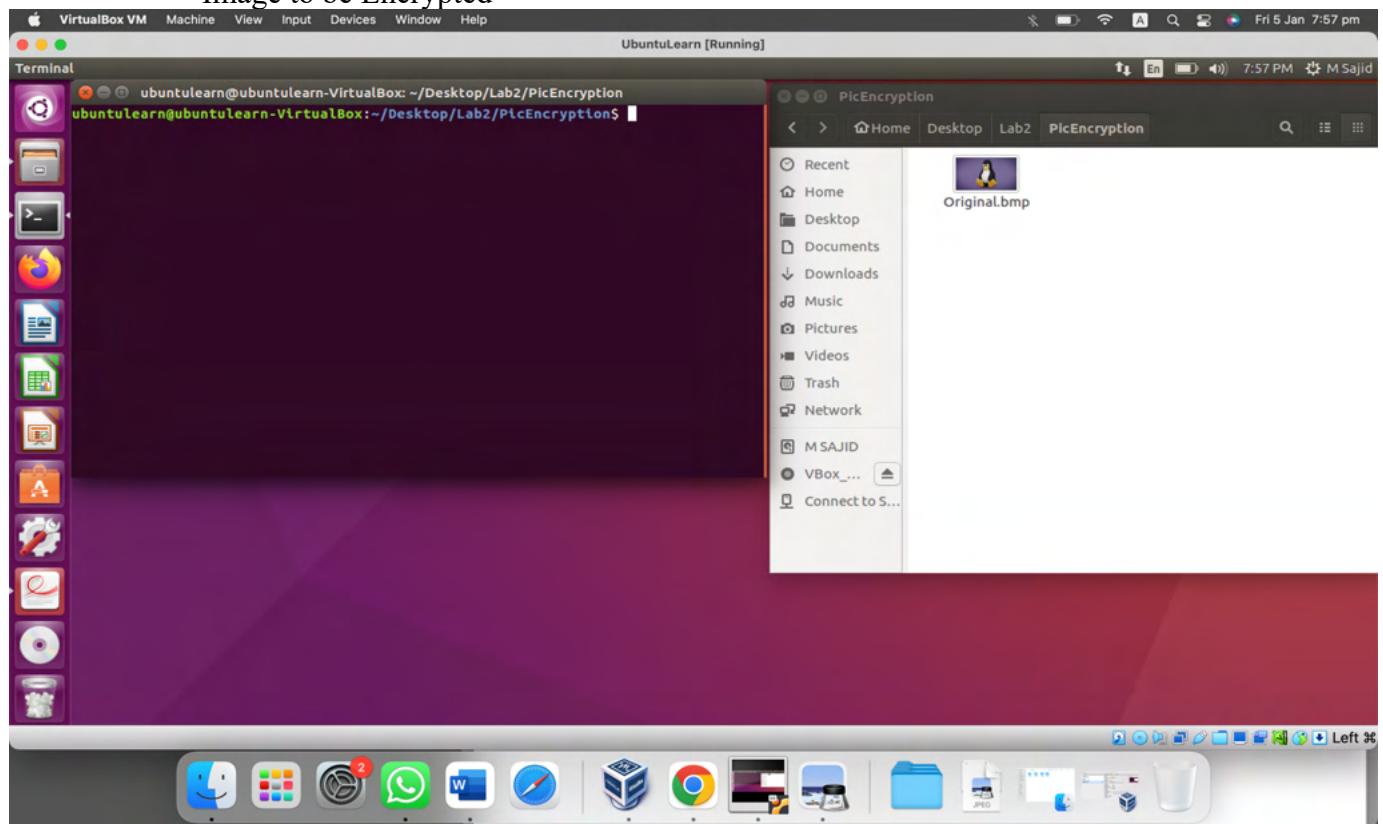
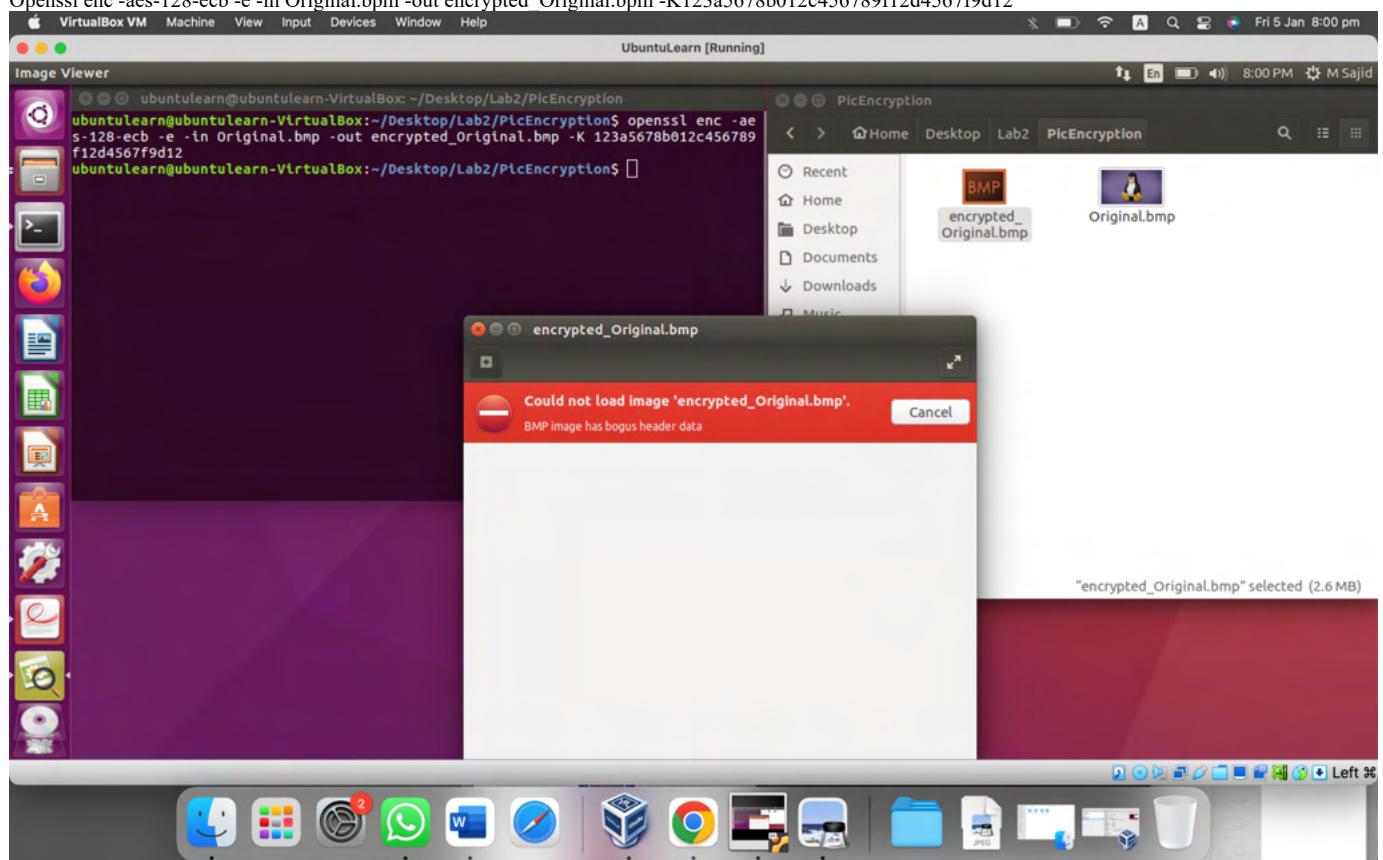


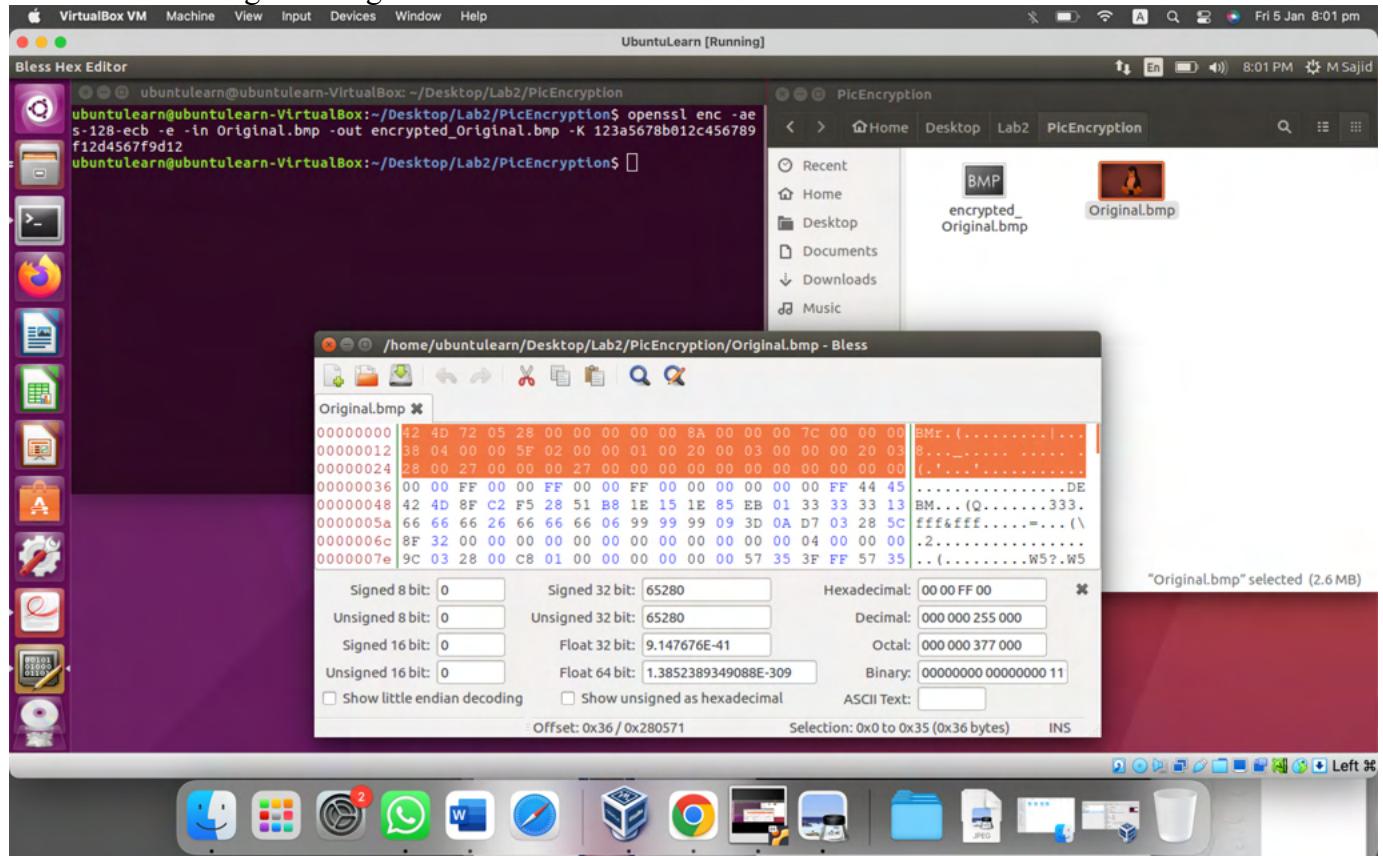
Image Encryption

Following code is used to encrypt the image:

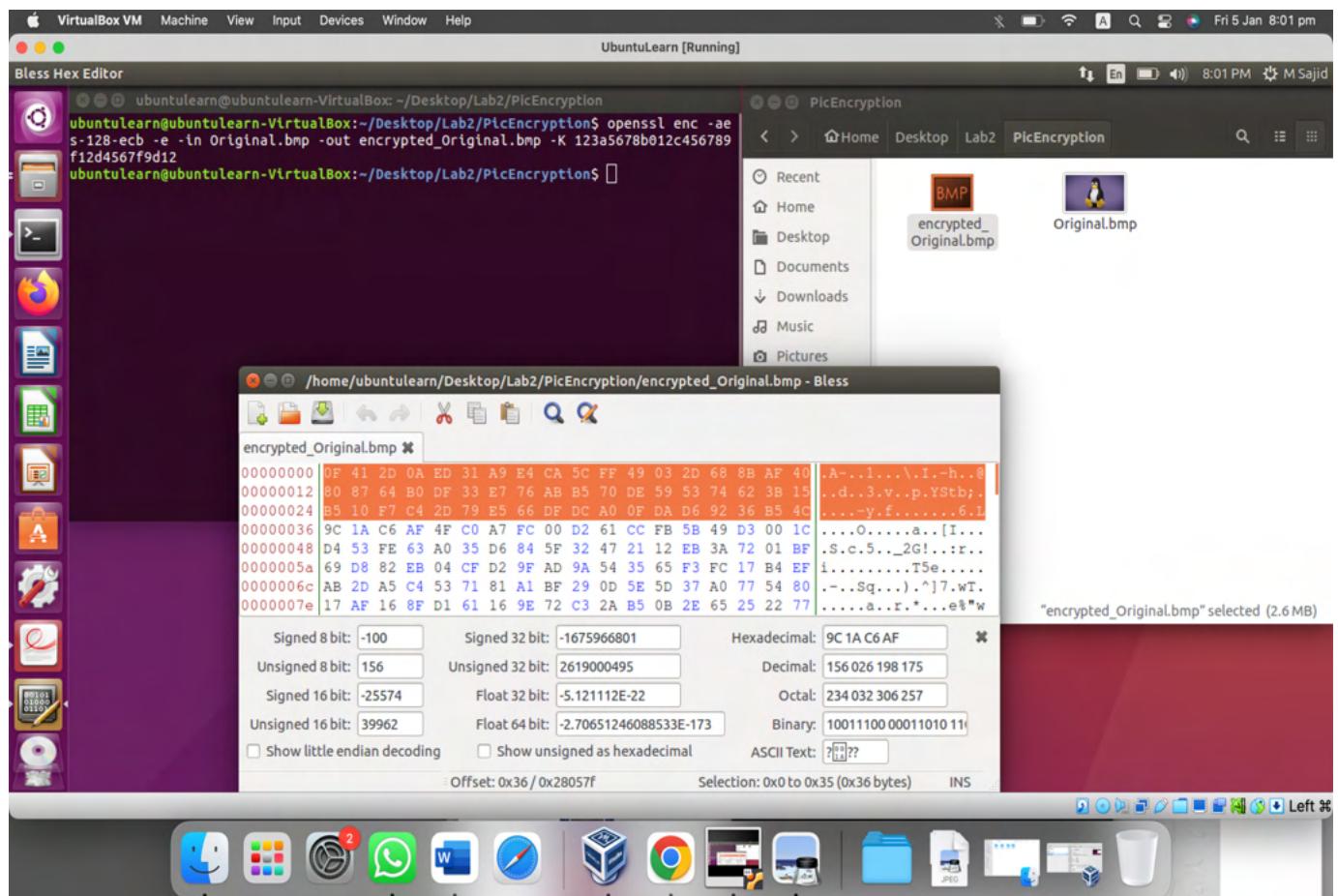
```
Openssl enc -aes-128-ecb -e -in Original.bmp -out encrypted_Original.bmp -K 123a5678b012c456789f12d4567f9d12
```



Original Image in Bless Editor

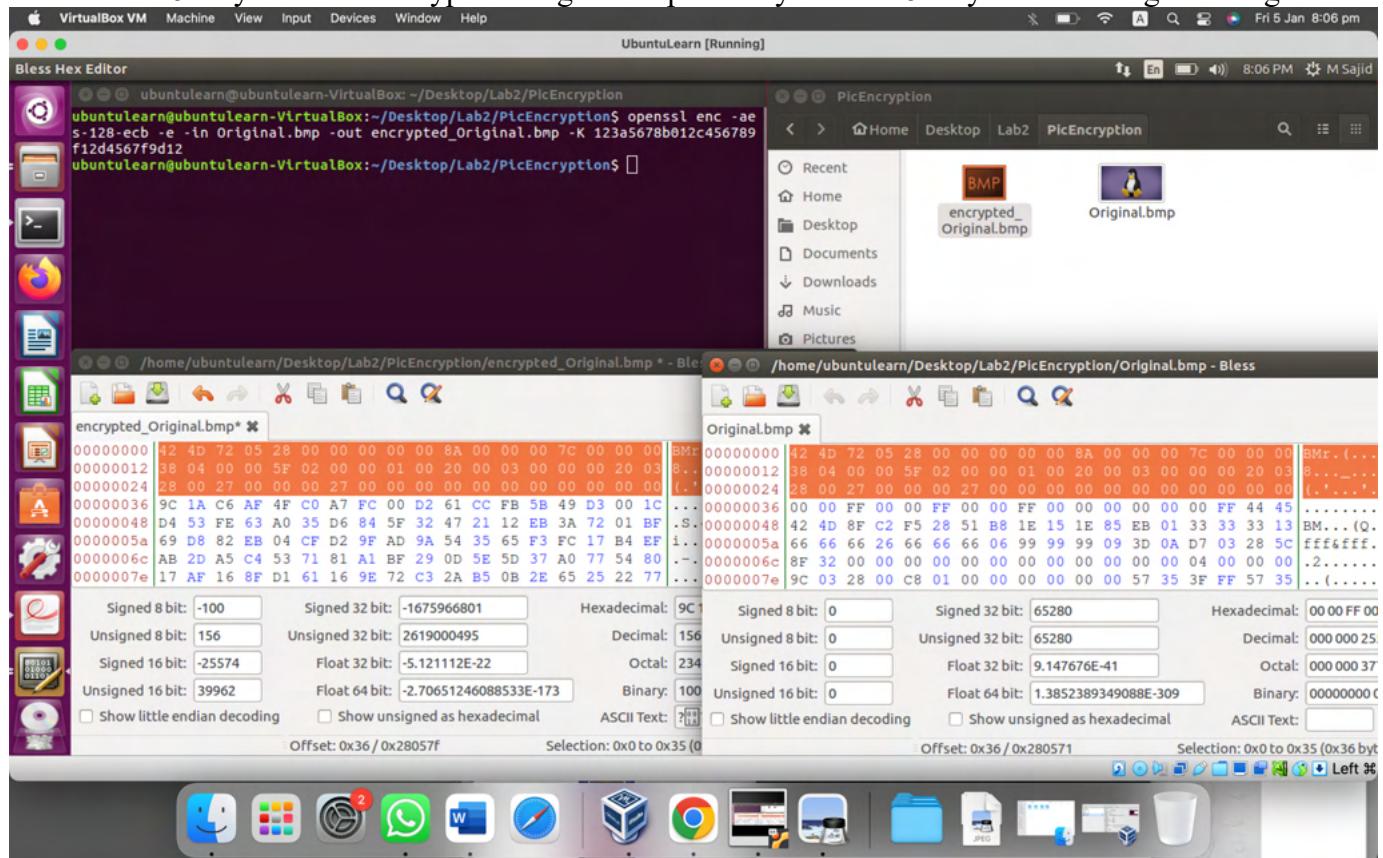


Encrypted Image in Bless Editor

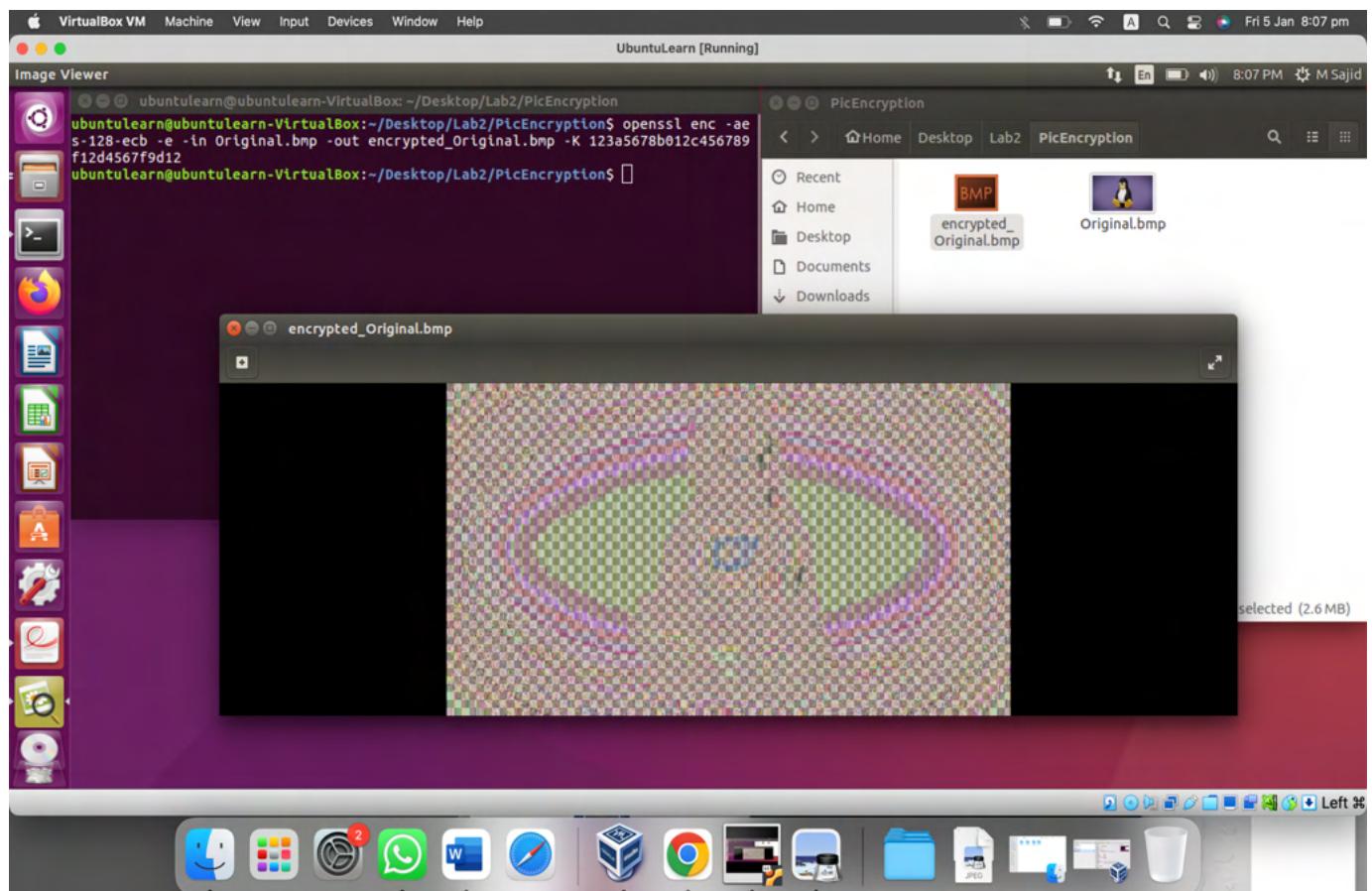


Byte Replacement

First 54 Bytes of the encrypted image are replaced by the first 54 Bytes of the original image

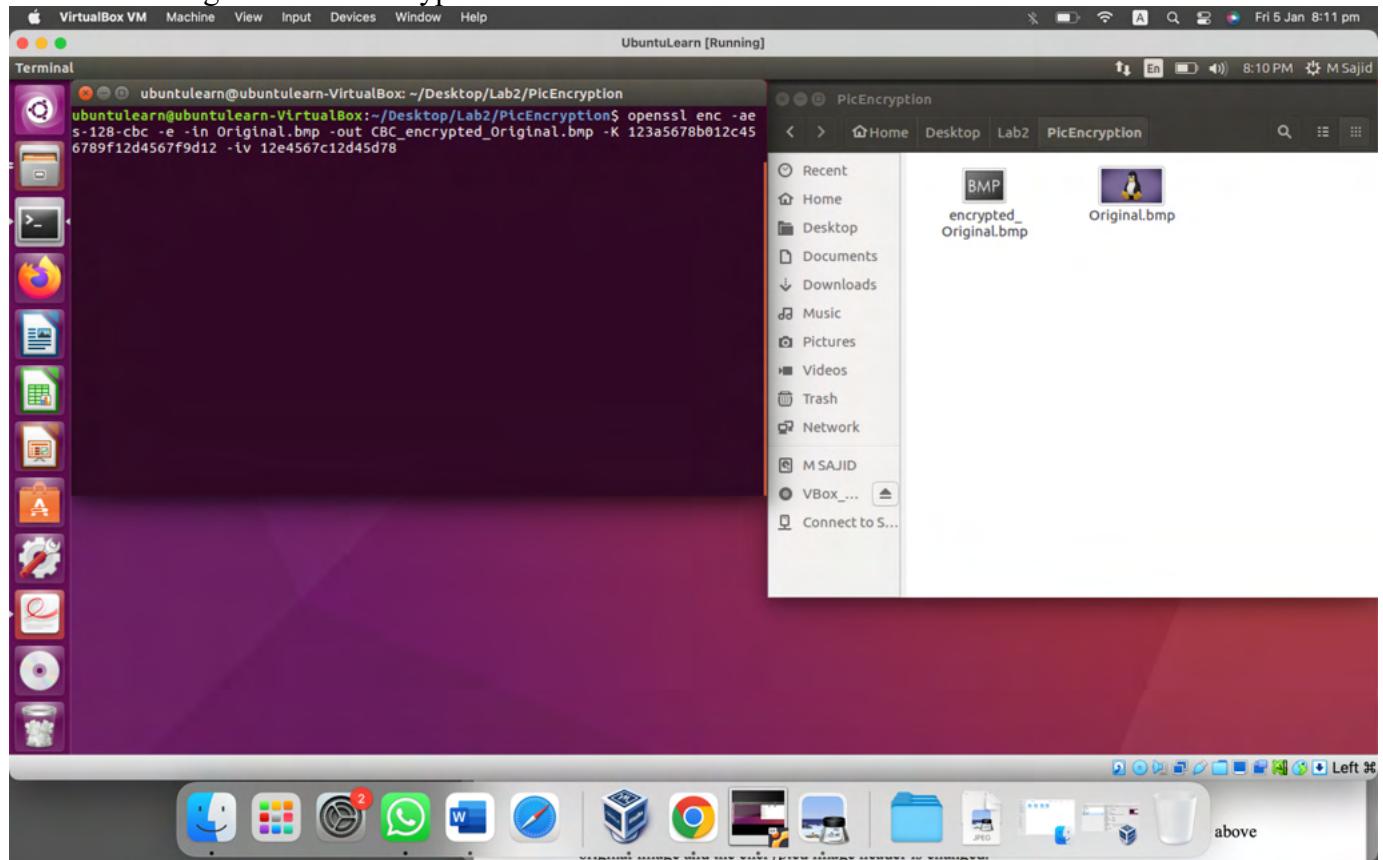


Encrypted Image After Byte Replacement



b. Image Encryption in CBC

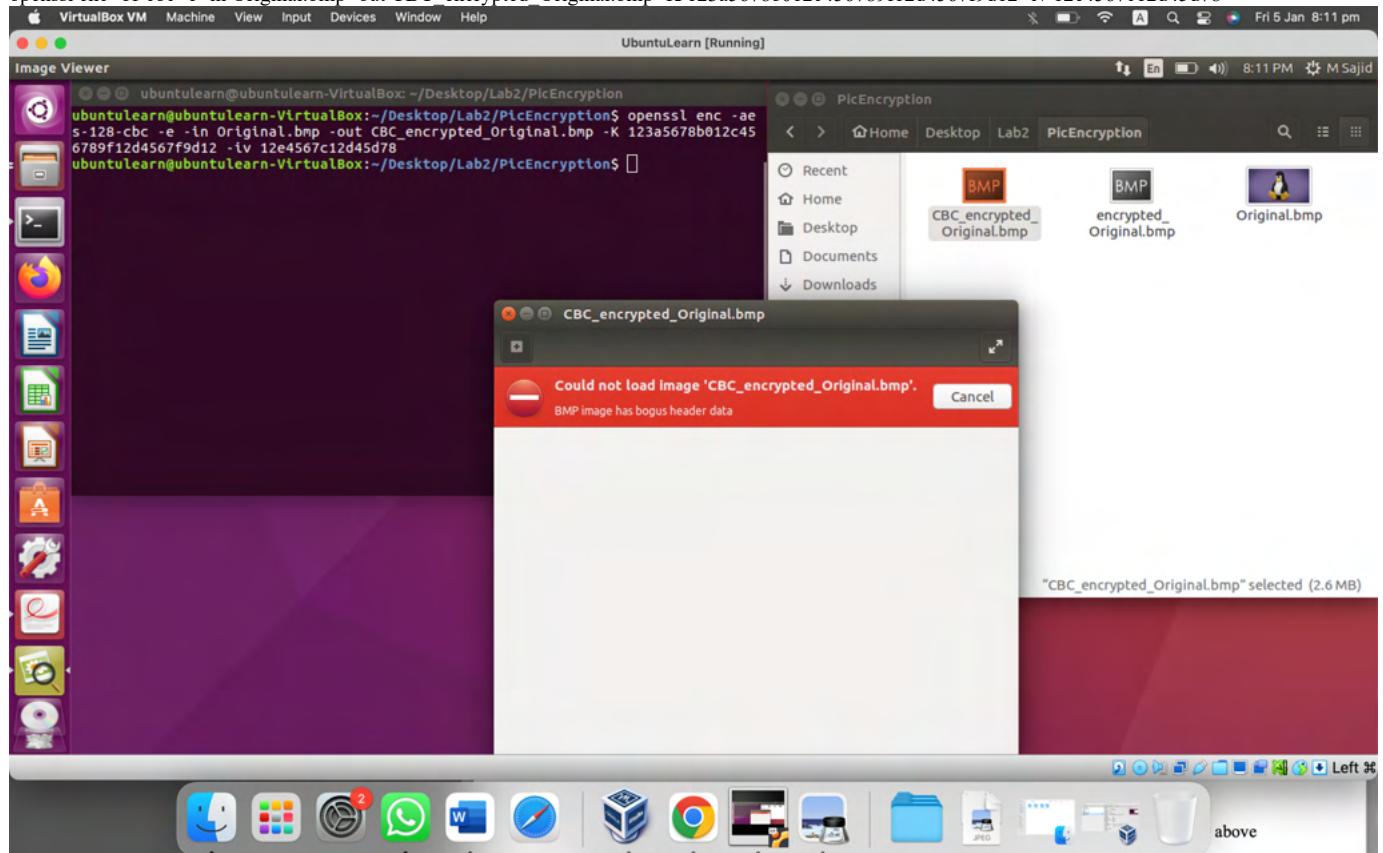
Image for CBC Encryption:



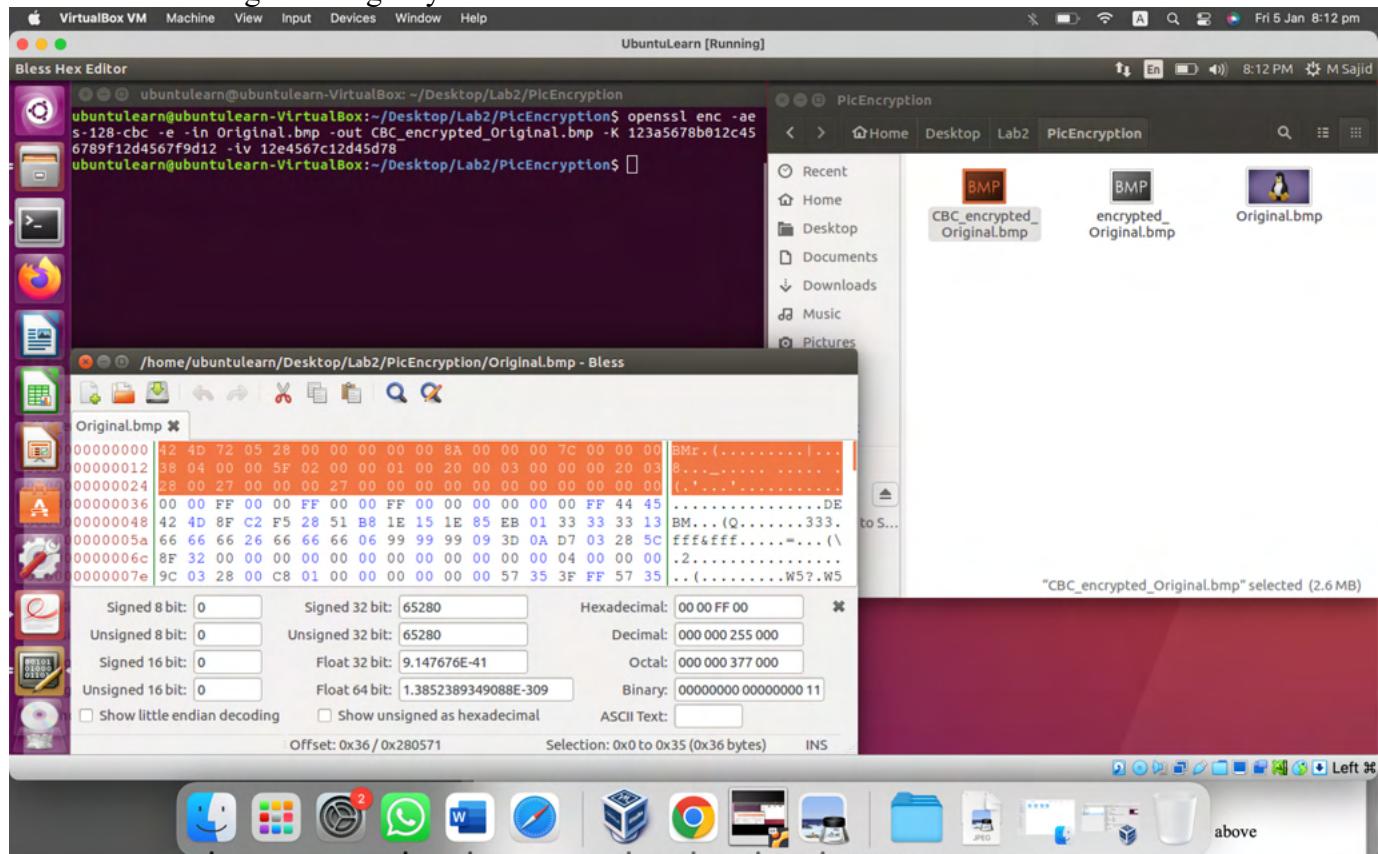
The Code for CBC Encryption

Following code is used for cbc Encryption

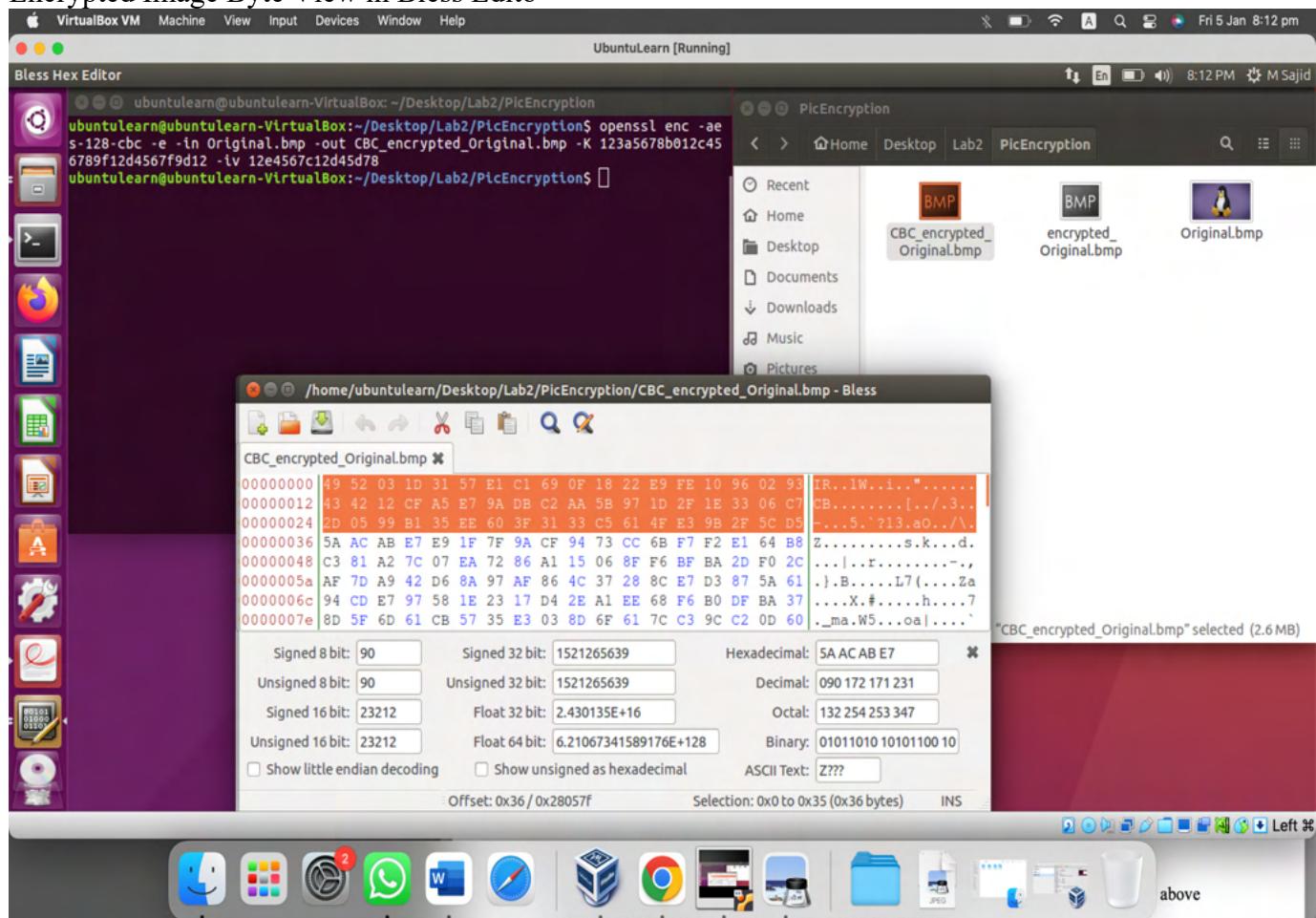
```
openssl enc -bf-cbc -e -in Original.bmp -out CBC_encrypted_Original.bmp -K 123a5678b012c456789f12d4567f9d12 -iv 12e4567c12d45d78
```



Original Image Byte View in Bless Editor



Encrypted Image Byte View in Bless Edito



Byte Replacement

The first 54 byte of the Encrypted Image replaced by the first 54 Bytes of the Original Image

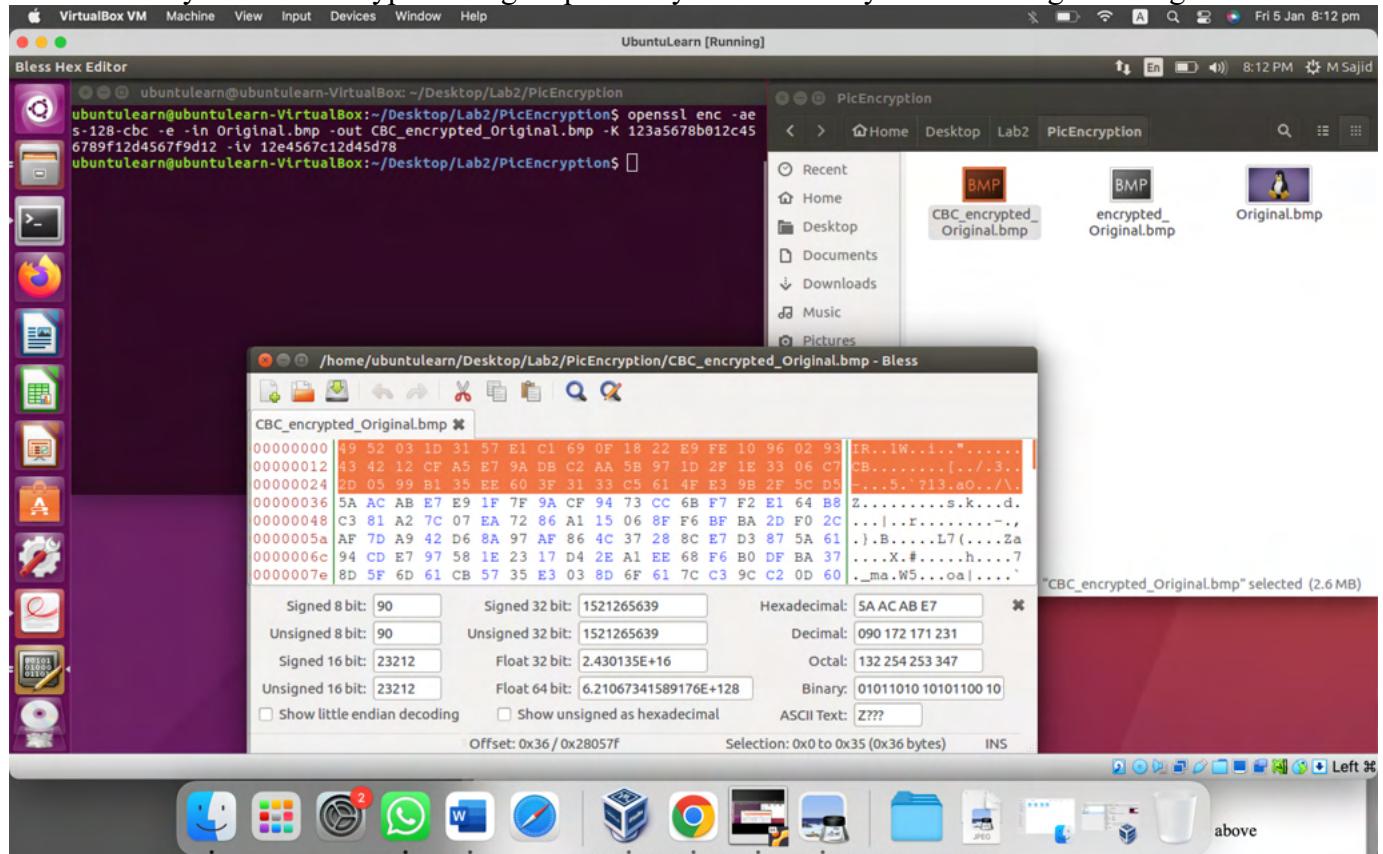
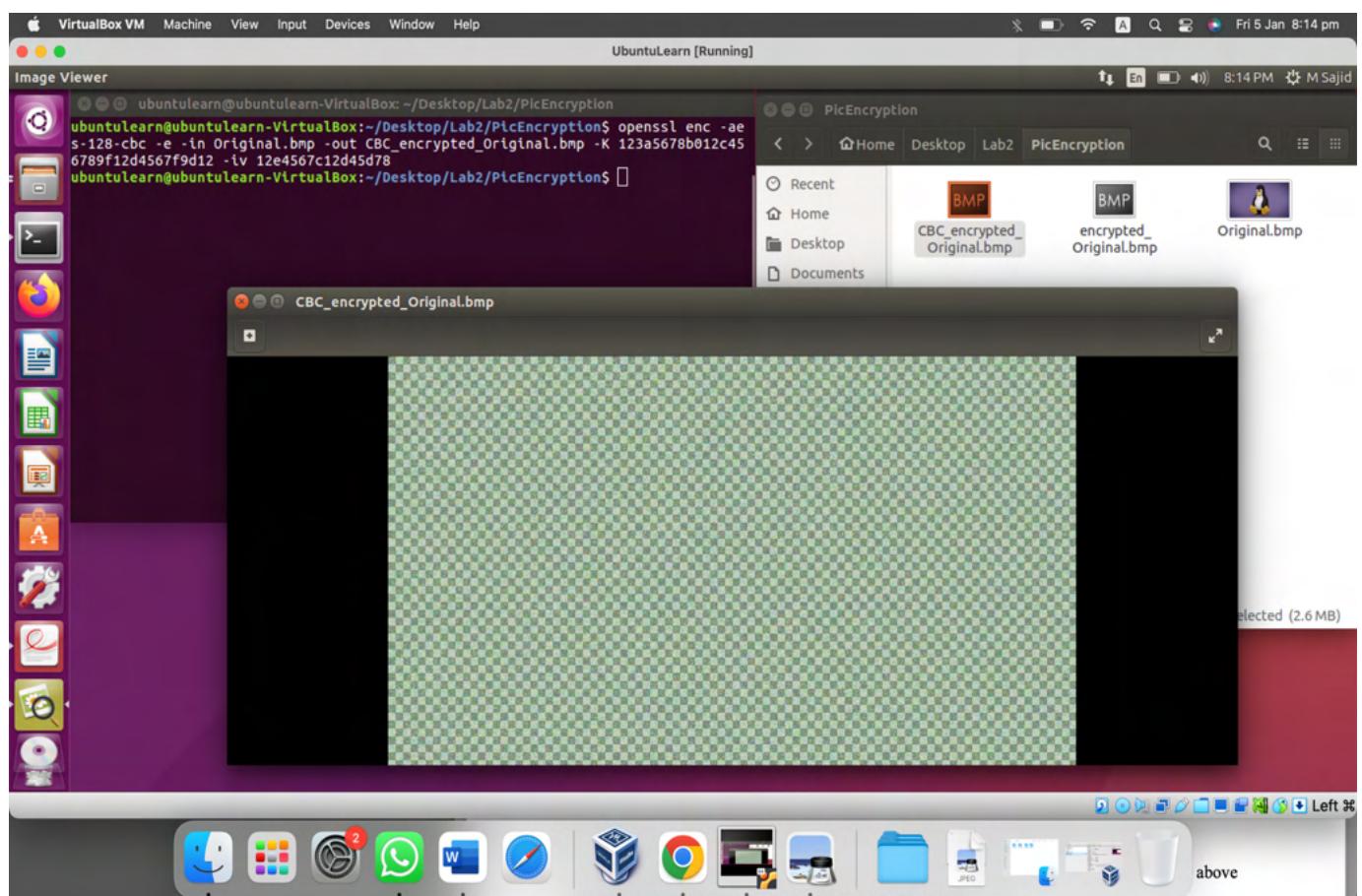


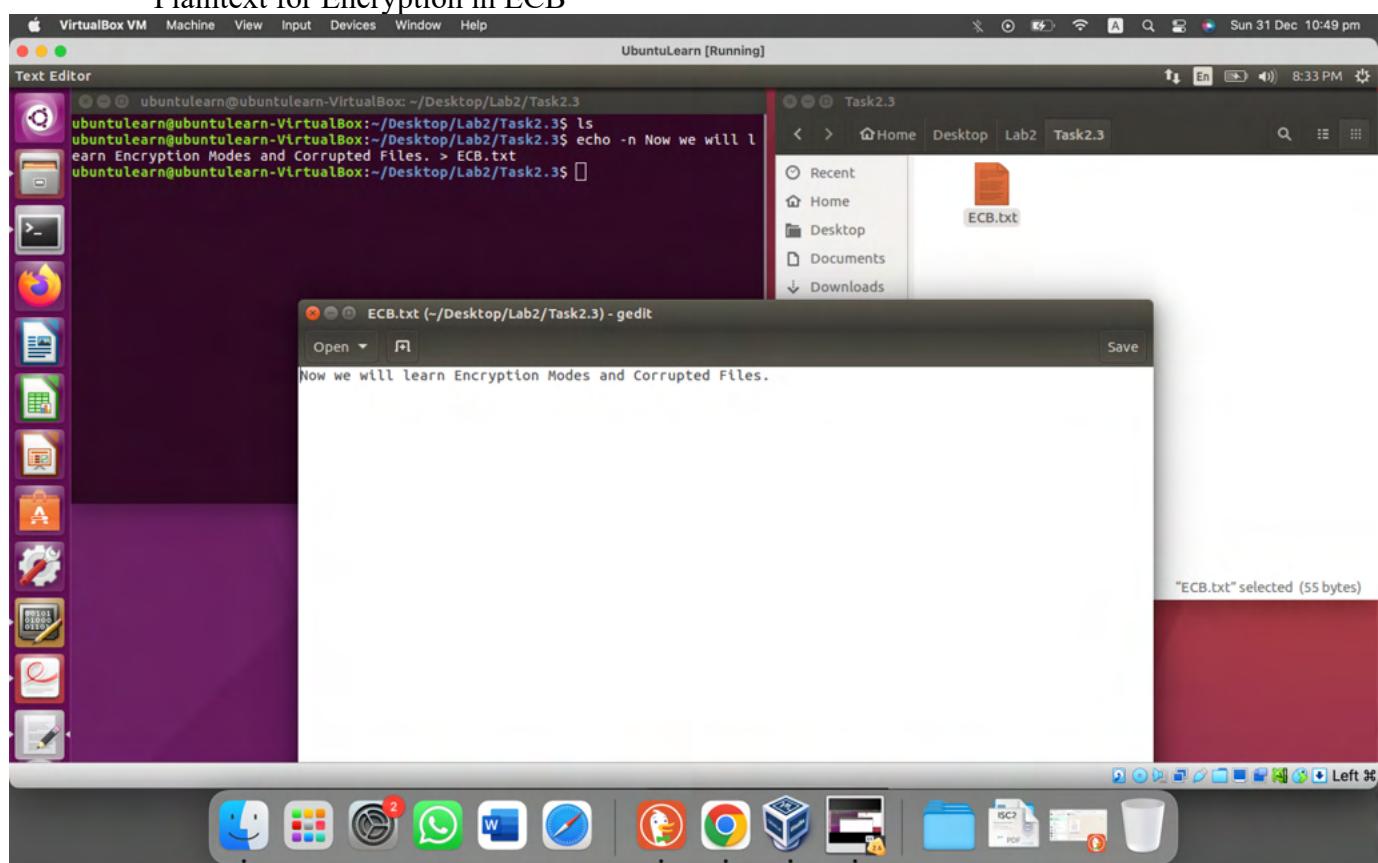
Image After Byte Replacement



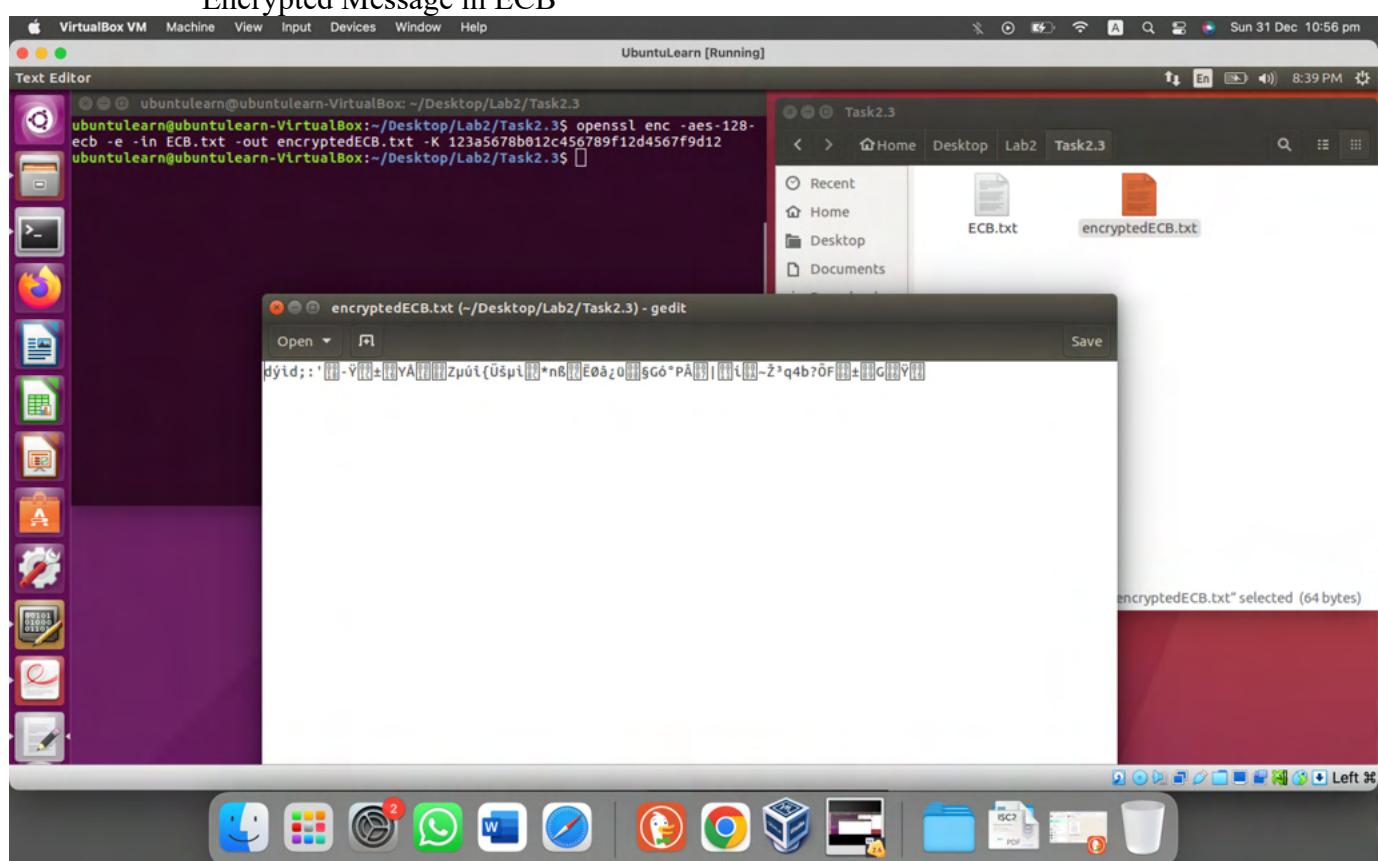
2.3. ENCRYPTION MODES ON CORRUPTED FILES

a. Encryption Mode: ECB

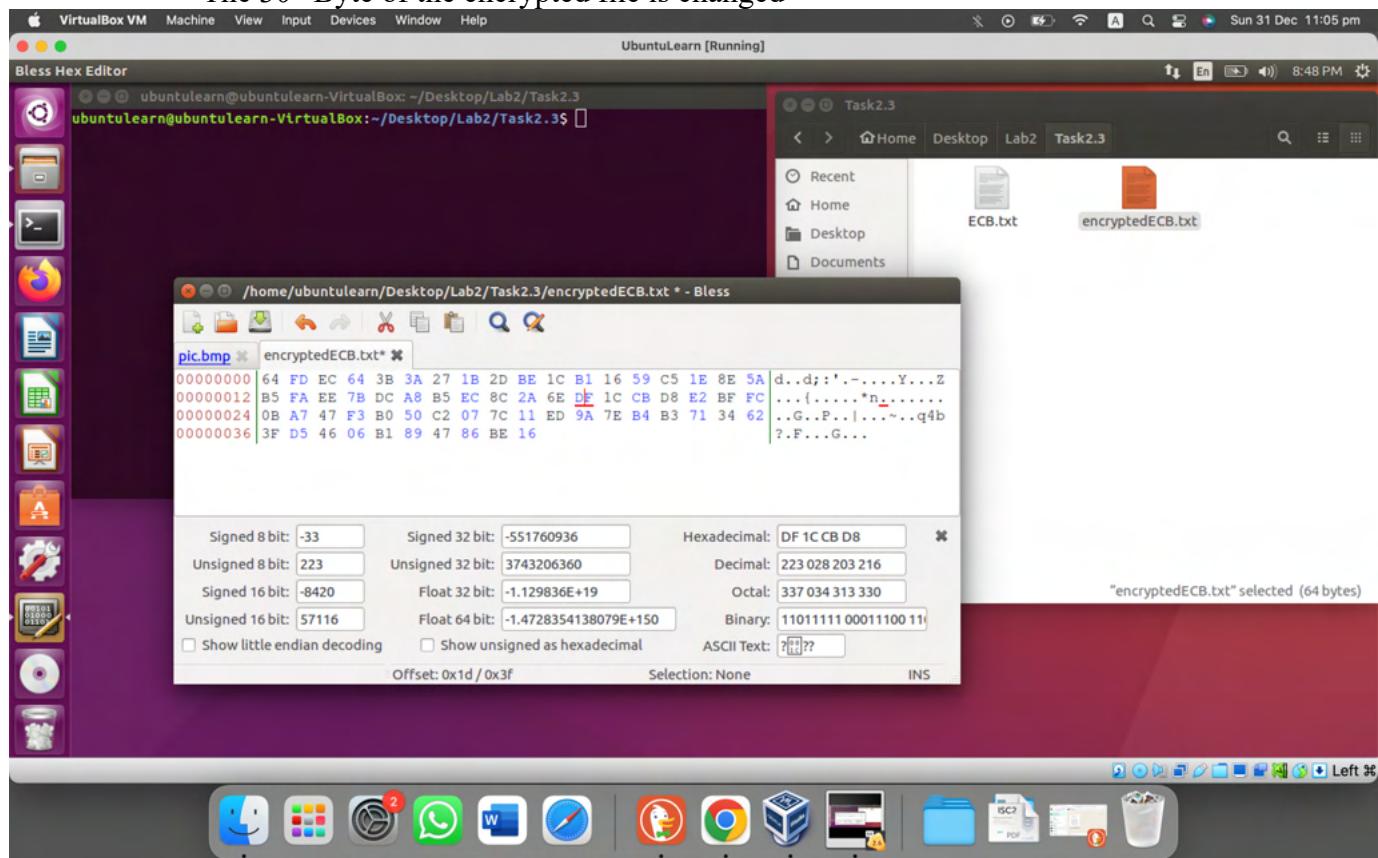
Plaintext for Encryption in ECB



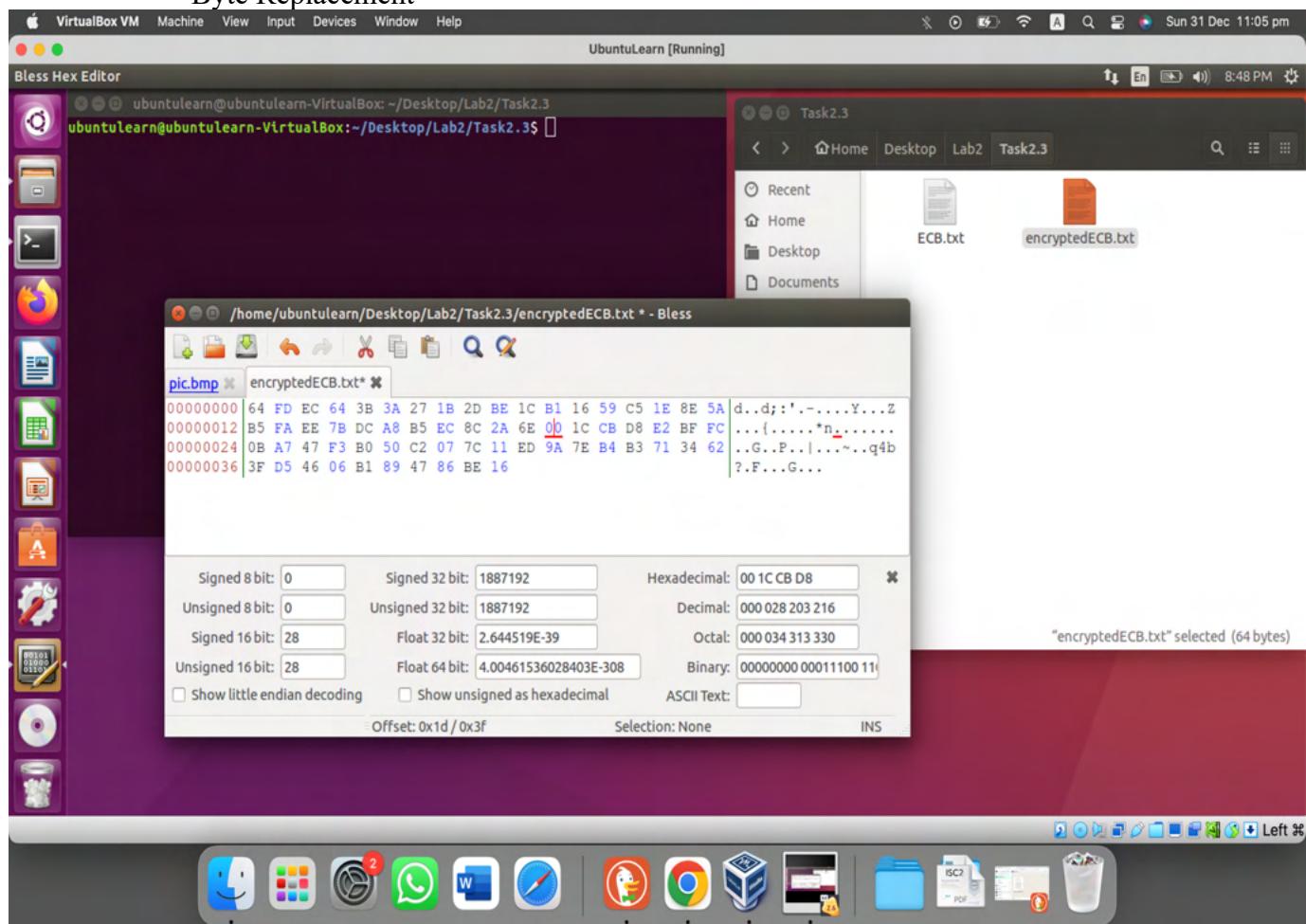
Encrypted Message in ECB



Encrypted Message in Bless Editor

The 30th Byte of the encrypted file is changed

Byte Replacement



Decryption of the Encrypted Message

The screenshot shows a Linux desktop environment with a purple theme. The desktop icons include a terminal window, a file manager window, and a text editor window.

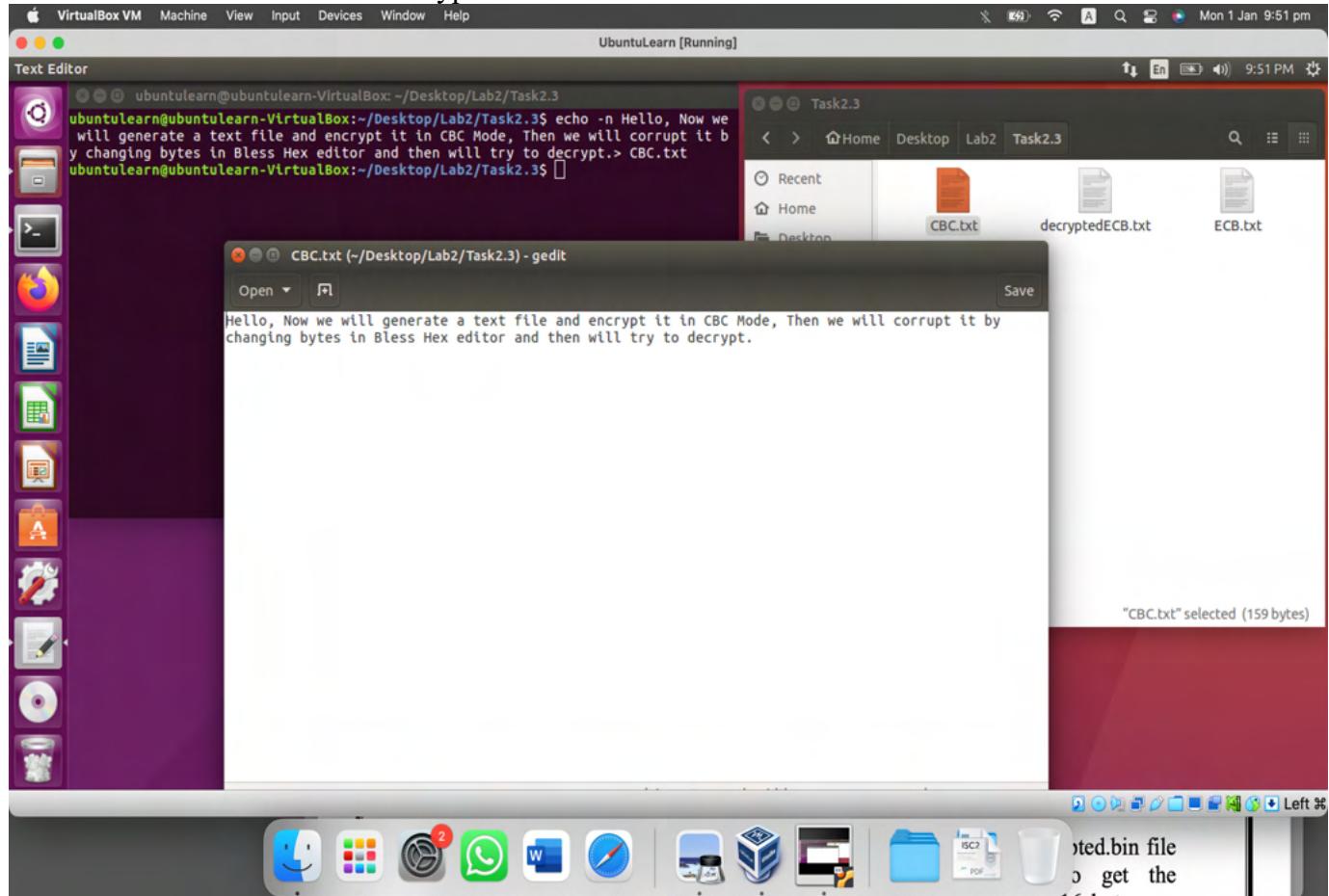
- Terminal Window:** The terminal window is titled "Text Editor" and shows the following command and output:

```
ubuntulearn@ubuntulearn-VirtualBox: ~/Desktop/Lab2/Task2.3
ubuntulearn@ubuntulearn-VirtualBox:~/Desktop/Lab2/Task2.3$ openssl enc -aes-128-
ecb -d -in encryptedECB.txt -out decryptedECB.txt -K 123a5678b012c456789f12d4567
f9d12
ubuntulearn@ubuntulearn-VirtualBox:~/Desktop/Lab2/Task2.3$
```
- File Manager Window:** The file manager window is titled "Task 2.3" and shows three files: "decryptedECB.txt" (selected), "ECB.txt", and "encryptedECB.txt". The sidebar shows recent files and locations: Recent, Home, Desktop, Documents, Downloads.
- Text Editor Window:** The text editor window is titled "decryptedECB.txt (~/Desktop/Lab2/Task2.3) - gedit" and contains the text: "Now we will learn about Abnormalities and Corrupted Files." A status bar at the bottom right indicates "encryptedECB.txt selected (55 bytes)".

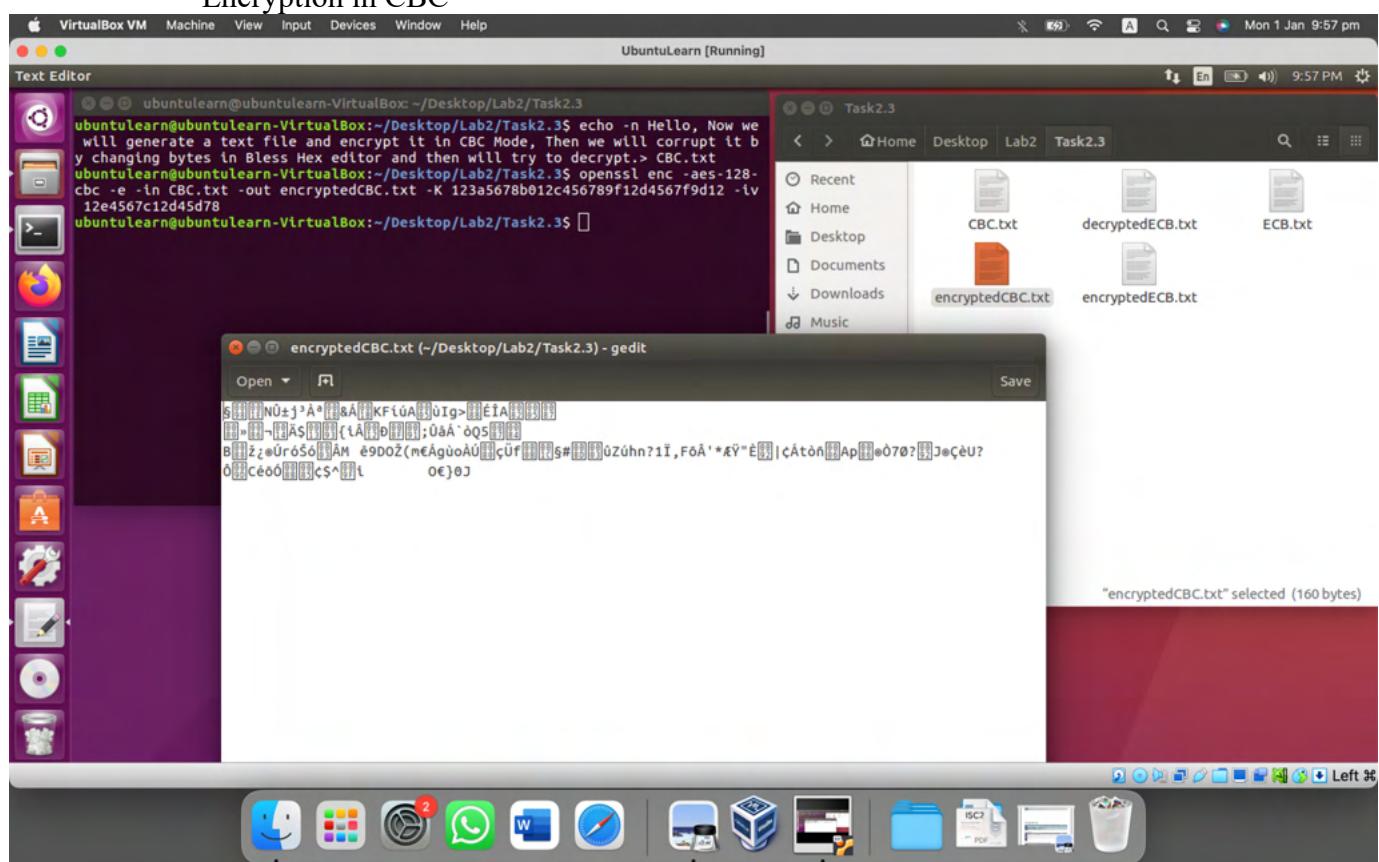
Now we have corrupted the 30th byte of the task3_aes_128_ecb_corrupted.bin file using the hex editor and decrypt the corrupted bin file to get the task3_ecb_corrupted.txt file. If we inspect it we observe that first 16 bytes are recovered, next 16 bytes are corrupted and the rest of the content is also fine (only 2nd block is corrupted because 30th byte is in second block.)

b. Encryption Mode: CBC

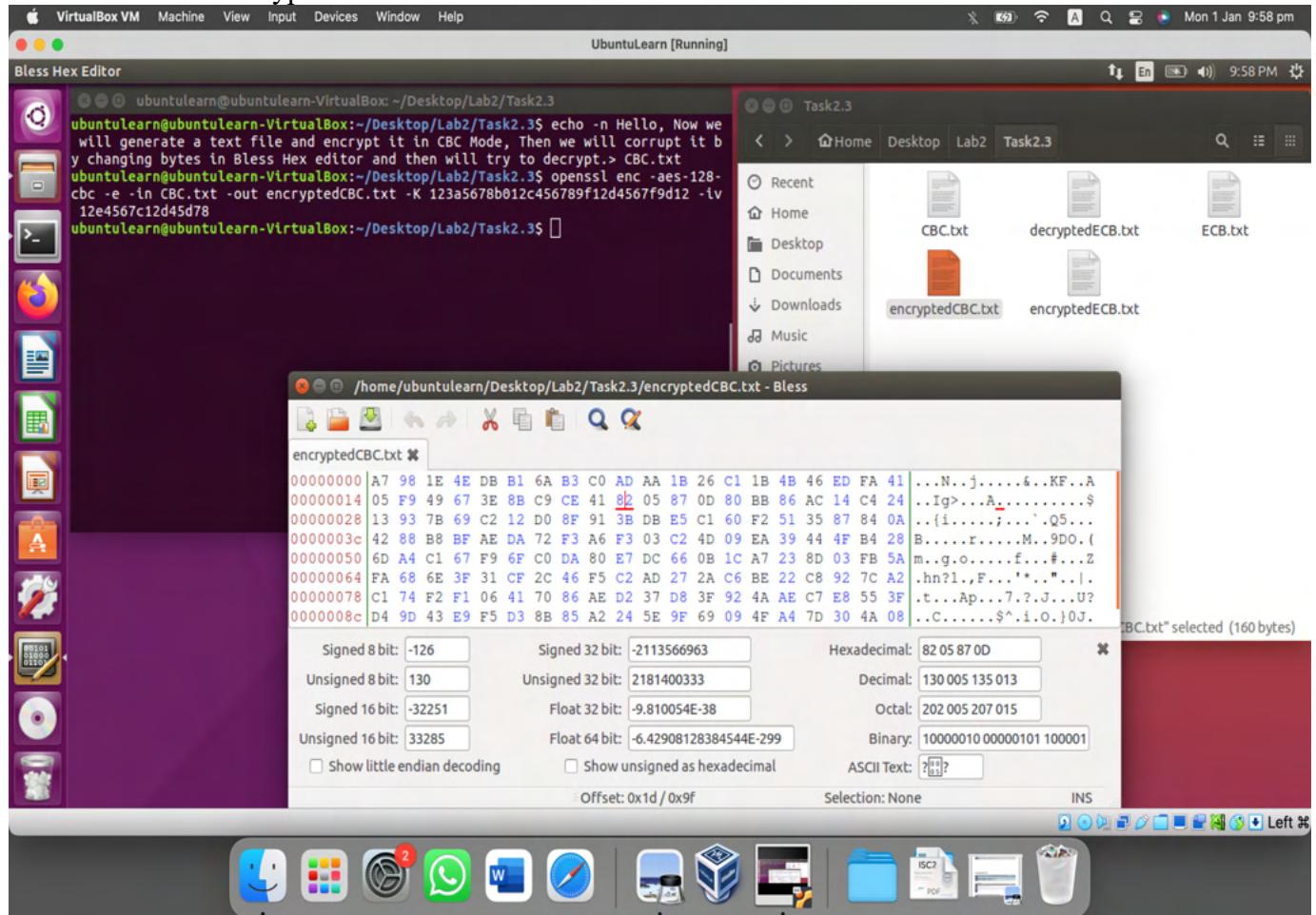
Plaintext for CBC Encryption



Encryption in CBC

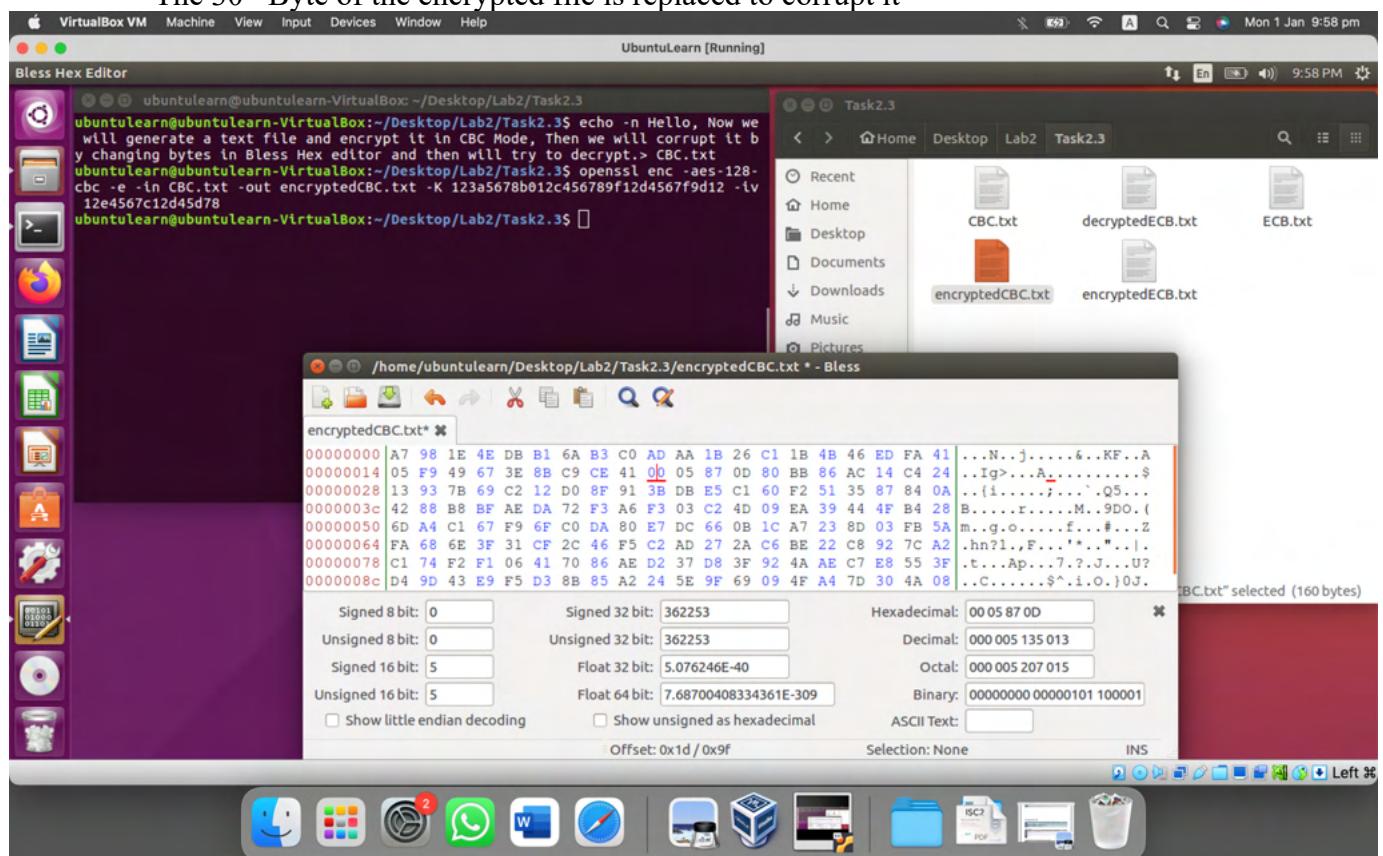


CBC Encrypted File in Bless Editor

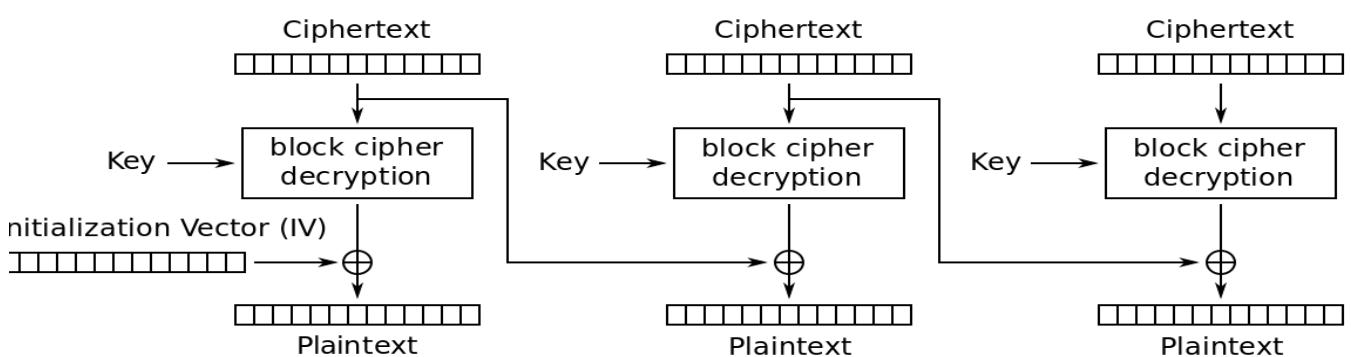
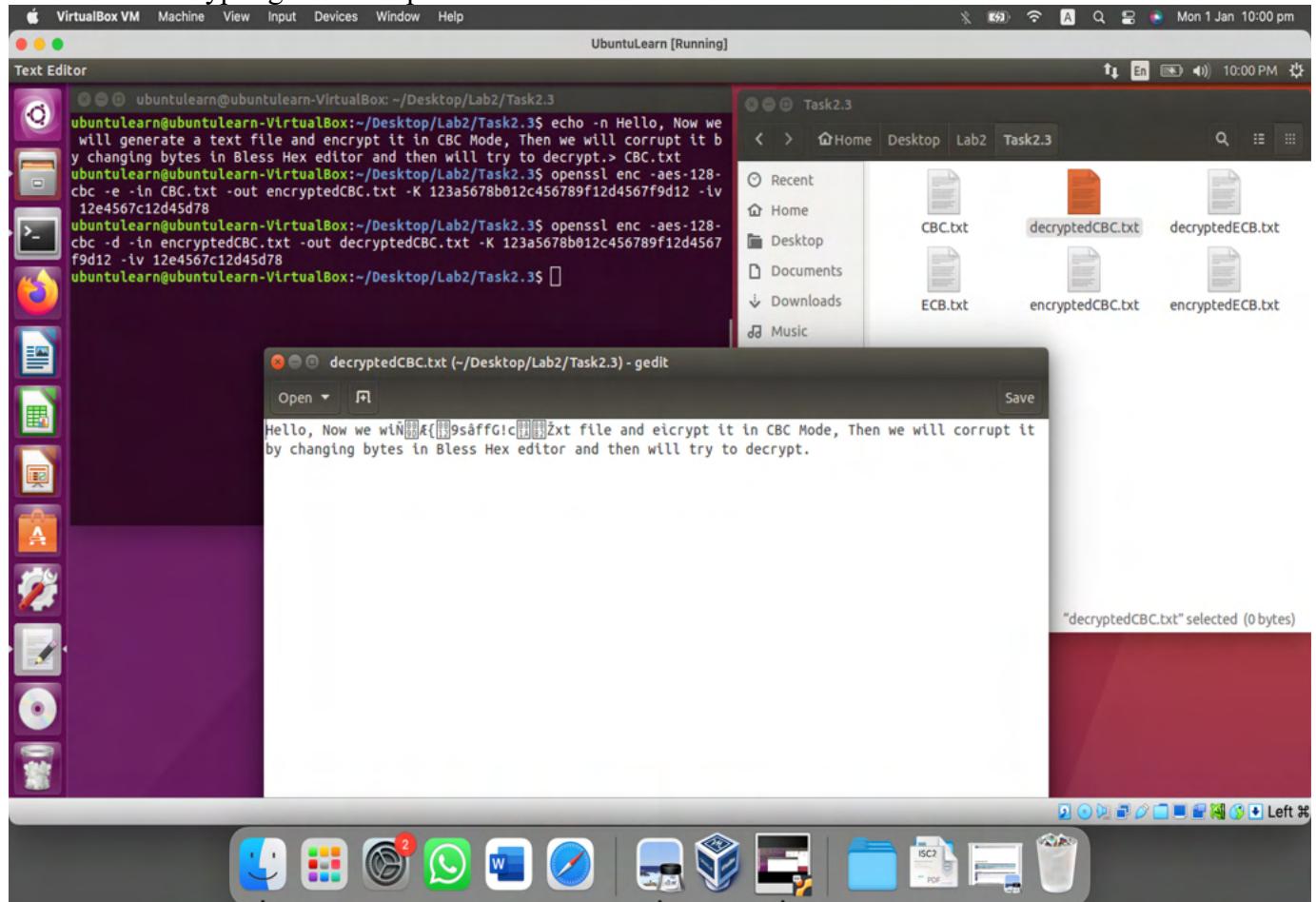


Byte Replacement in Bless Editor

The 30th Byte of the encrypted file is replaced to corrupt it



Decrypting the Corrupted File

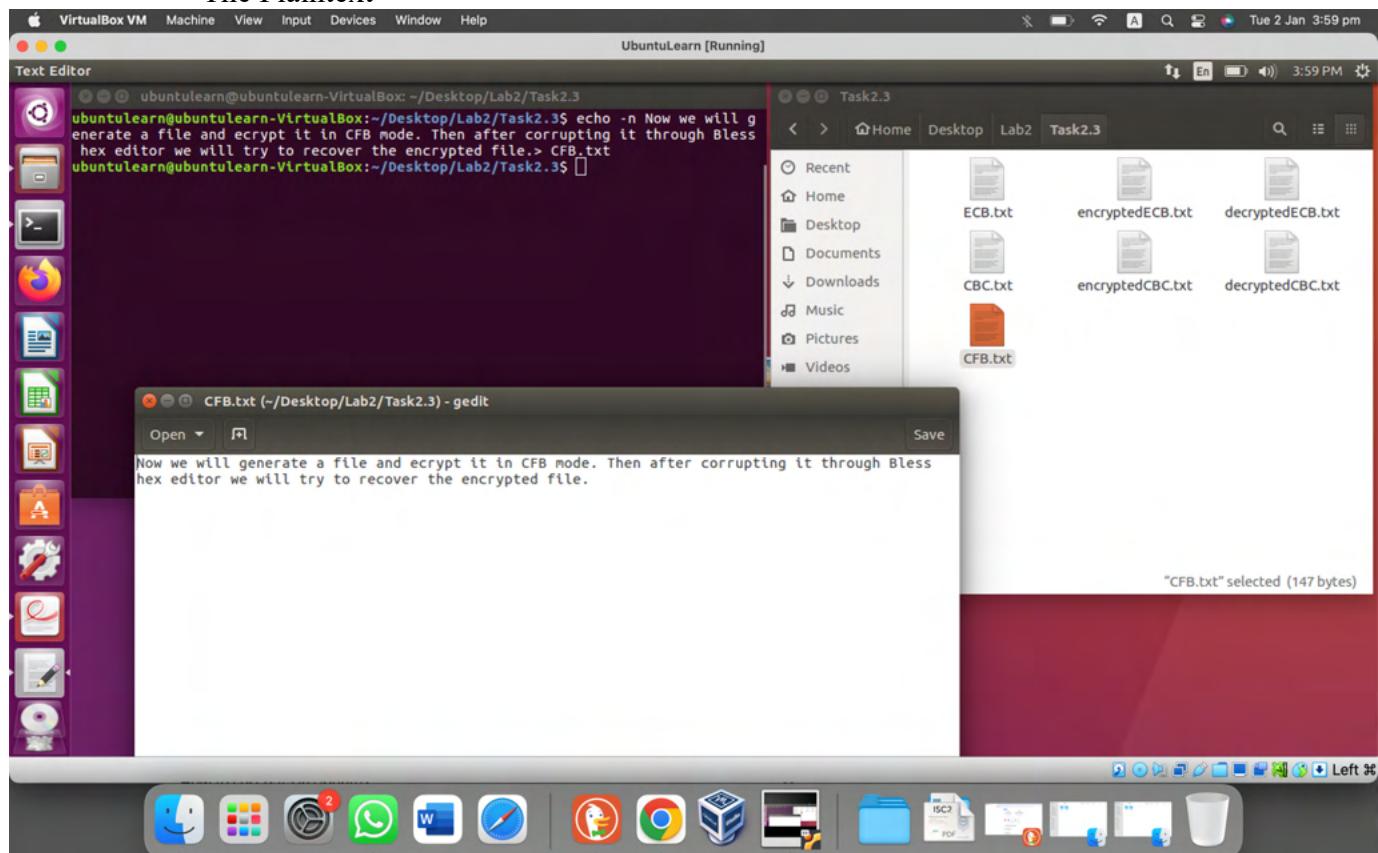


We observe that entire 2nd block is corrupted, 1 byte of third block (46th byte to be precise because it is xored with 30th byte of 2nd block) is corrupted and then rest of the content can be recovered.

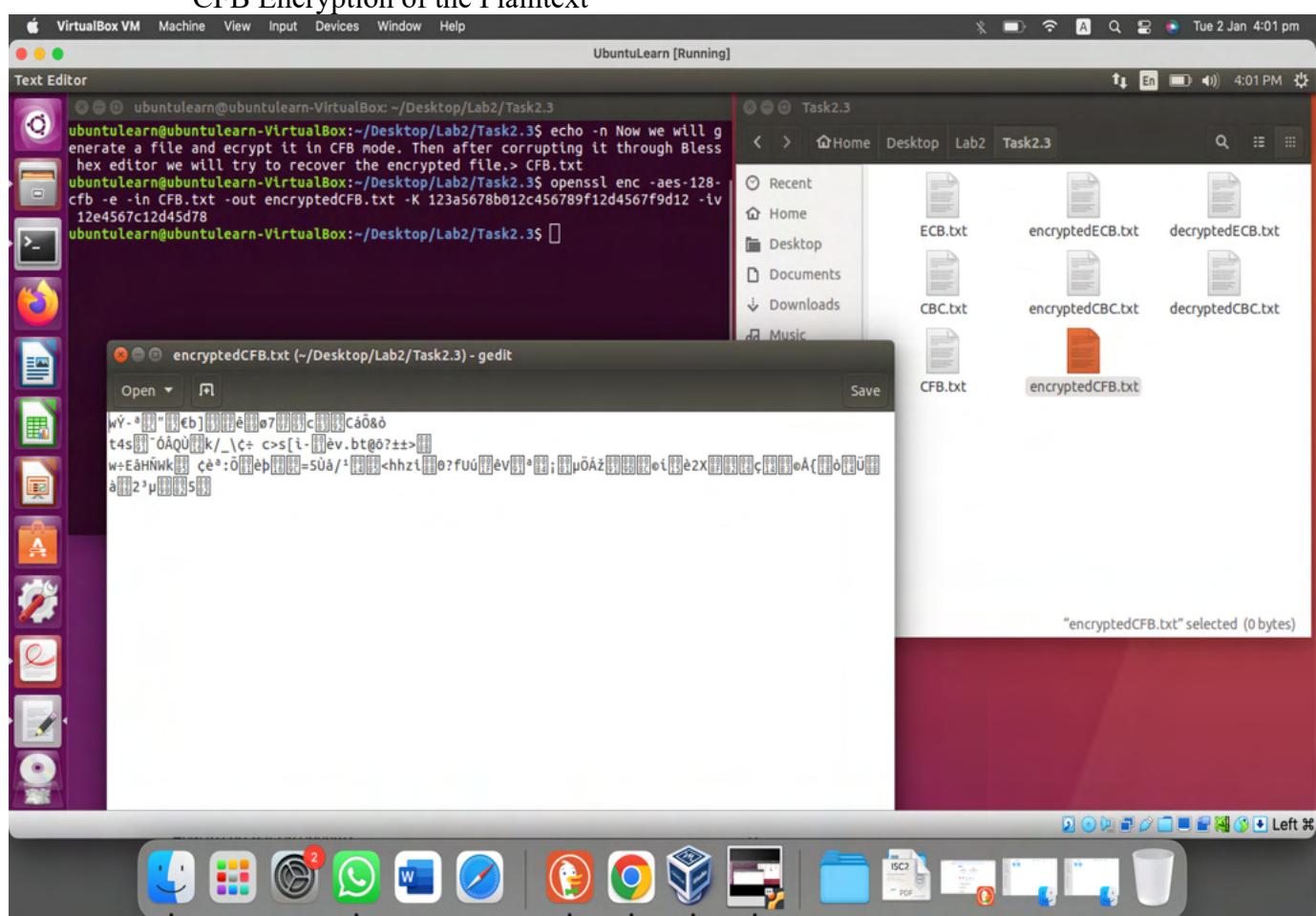
c. Encryption Mode: CFB

A text file is generated to be encrypted in CFB Mode

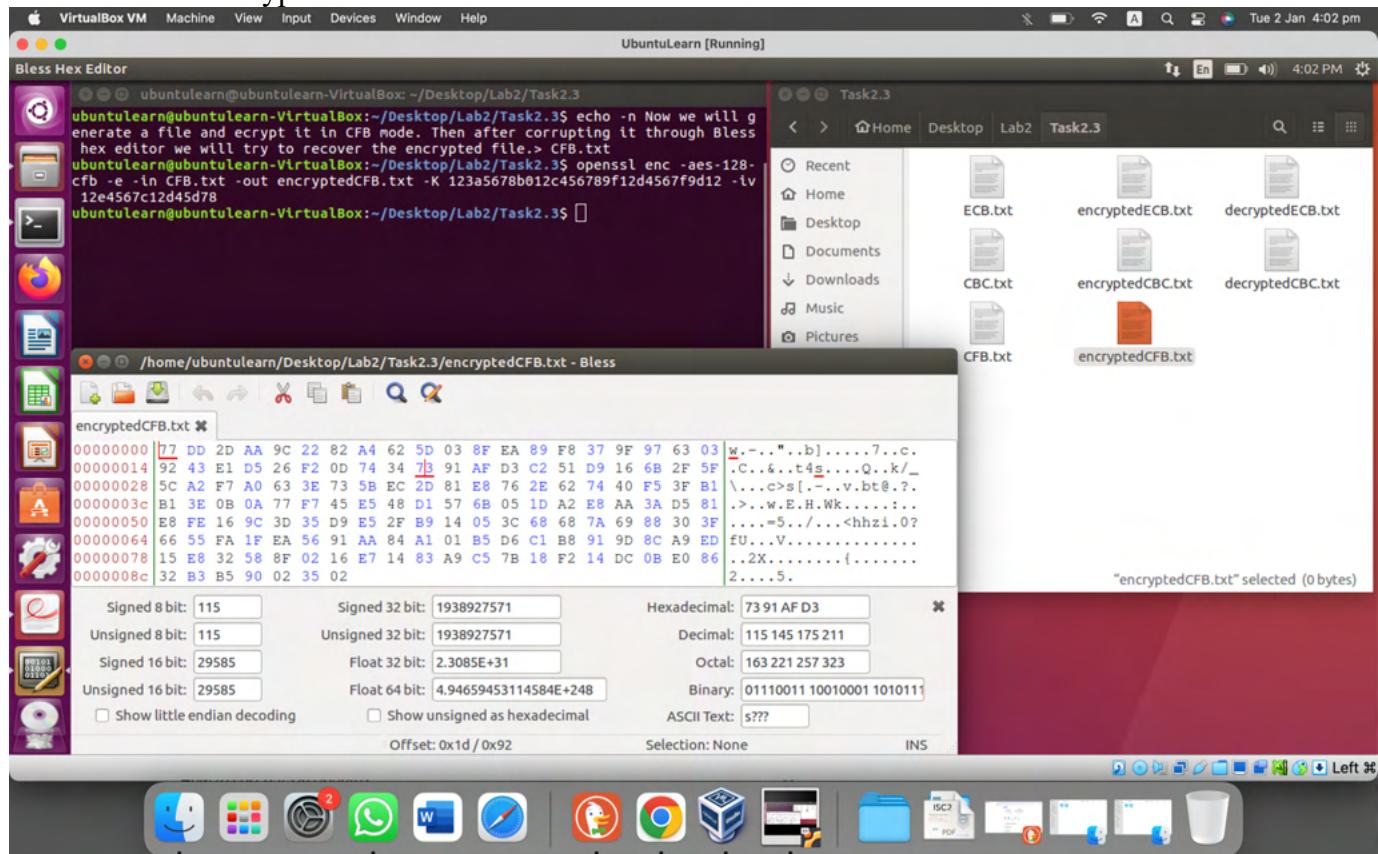
The Plaintext



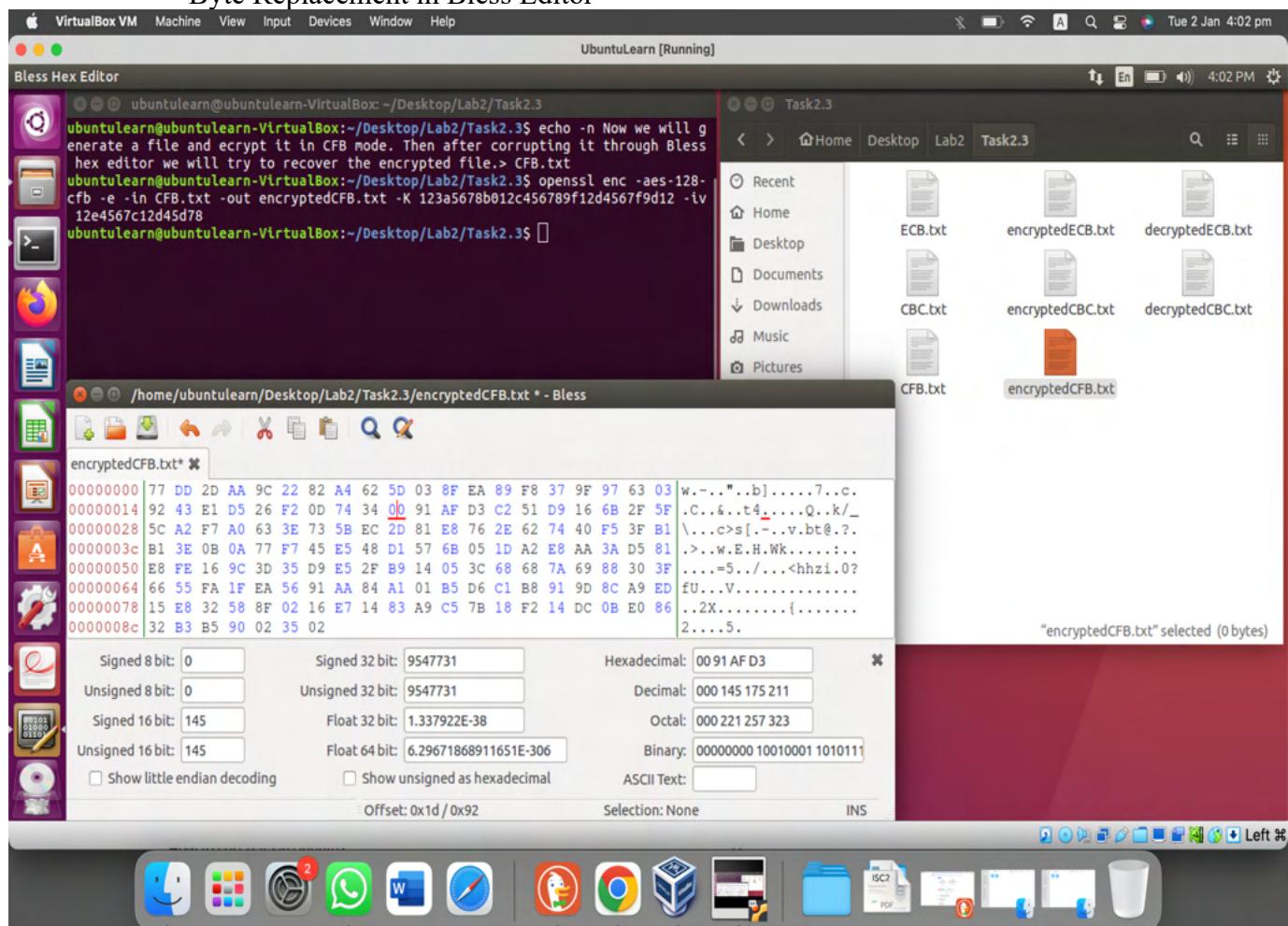
CFB Encryption of the Plaintext



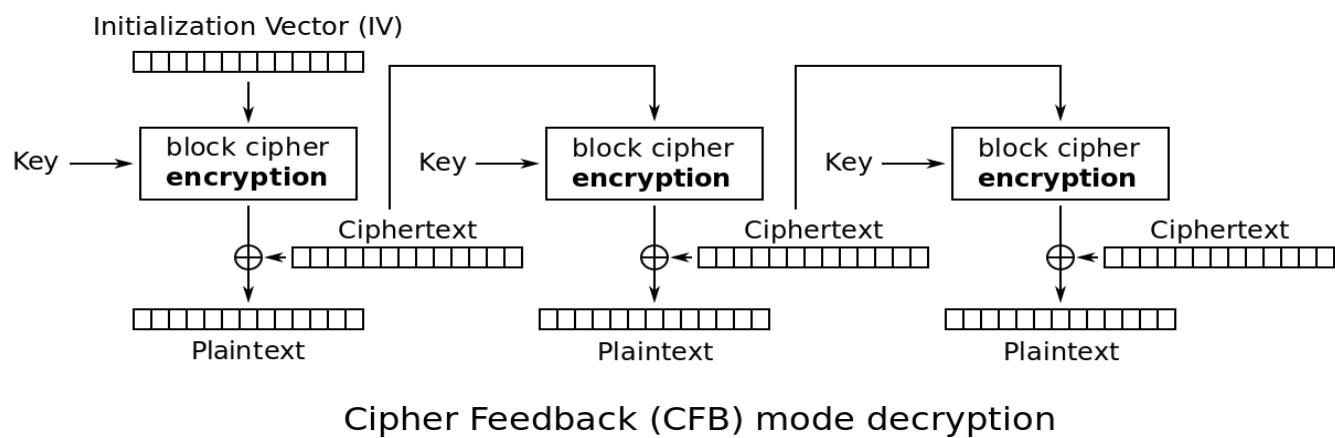
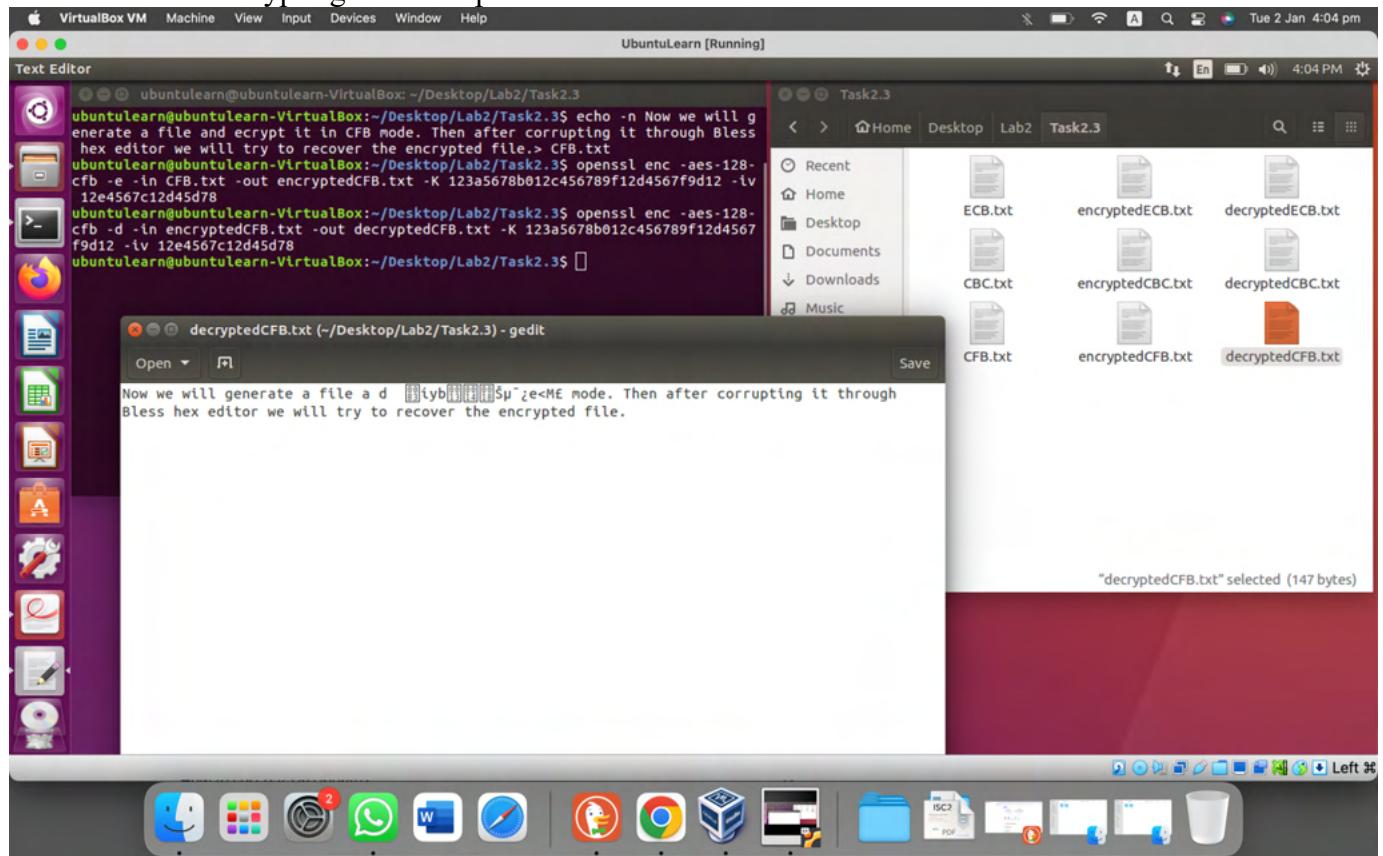
Encrypted File in Bless Editor



Byte Replacement in Bless Editor



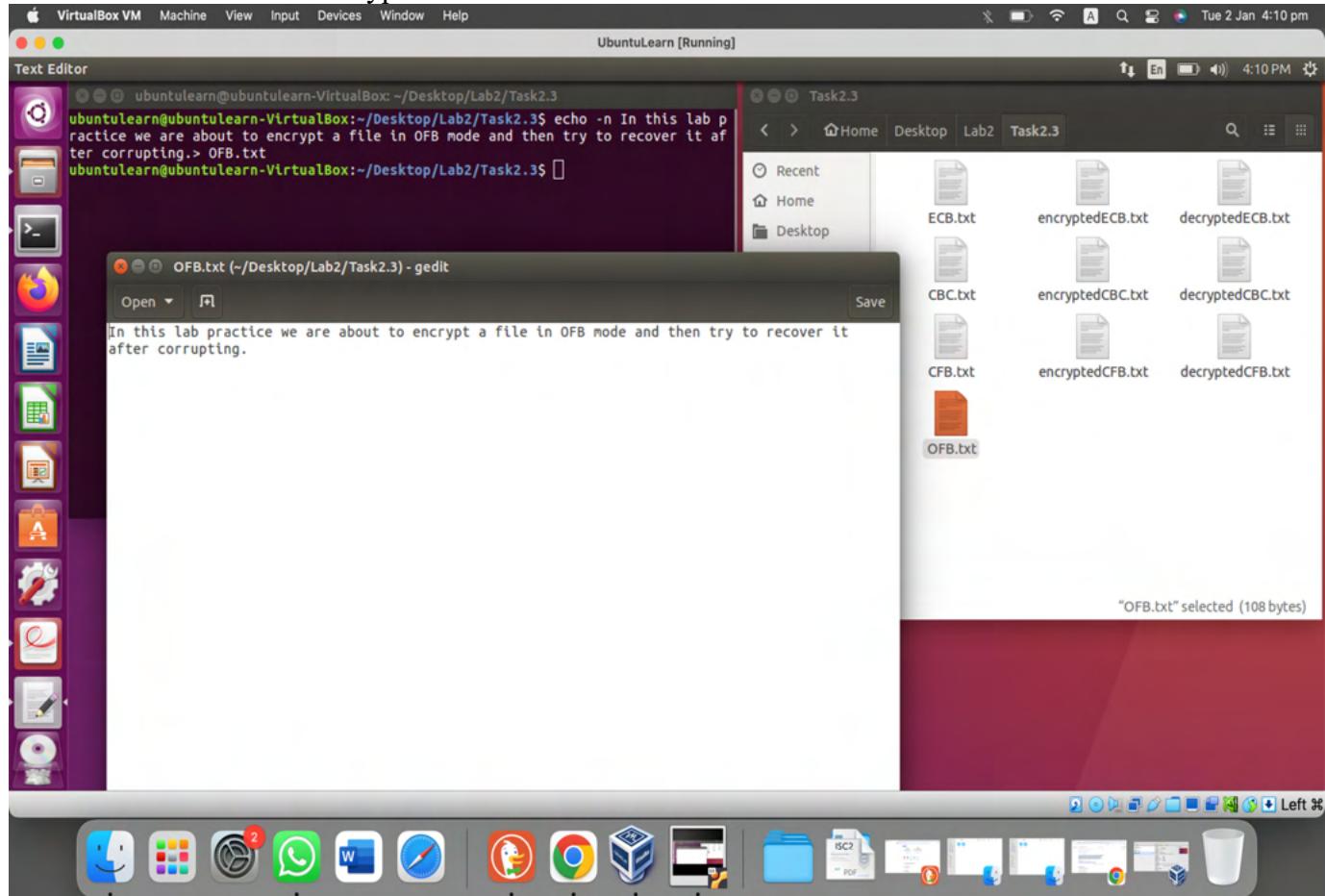
Decrypting the Corrupted File



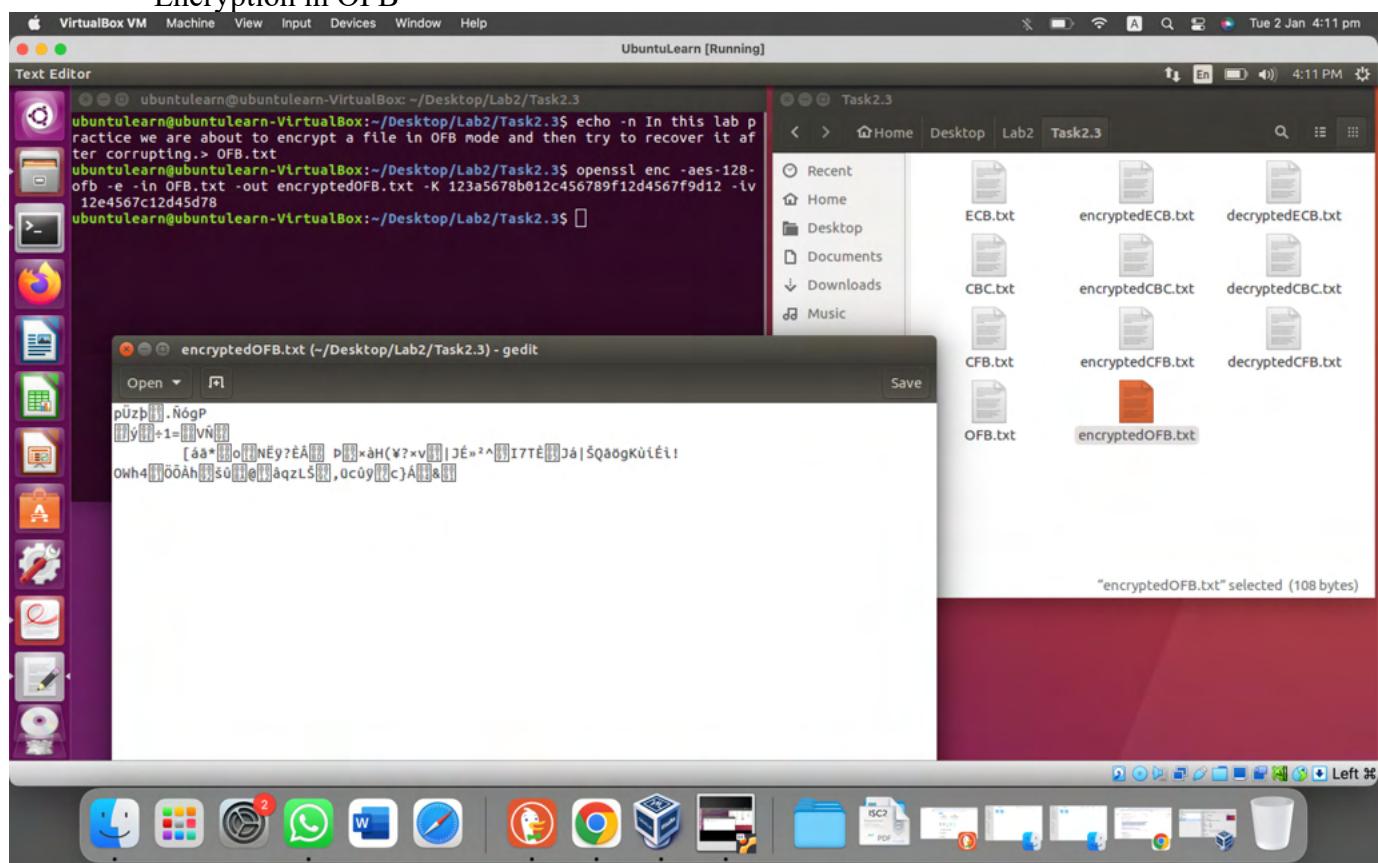
2nd block 1 byte is corrupted, 30th block is completely corrupted. Rest of the content is recovered. (In the decryption diagram above, ciphertext of 1st block is second to be decrypted and then xor is performed with ciphertext of 2nd block, so 1byte of plain text is corrupted, ciphertext of 2nd block is decrypted, so entire plain text of 3rd block is corrupted).

d. Encryption Mode: OFB

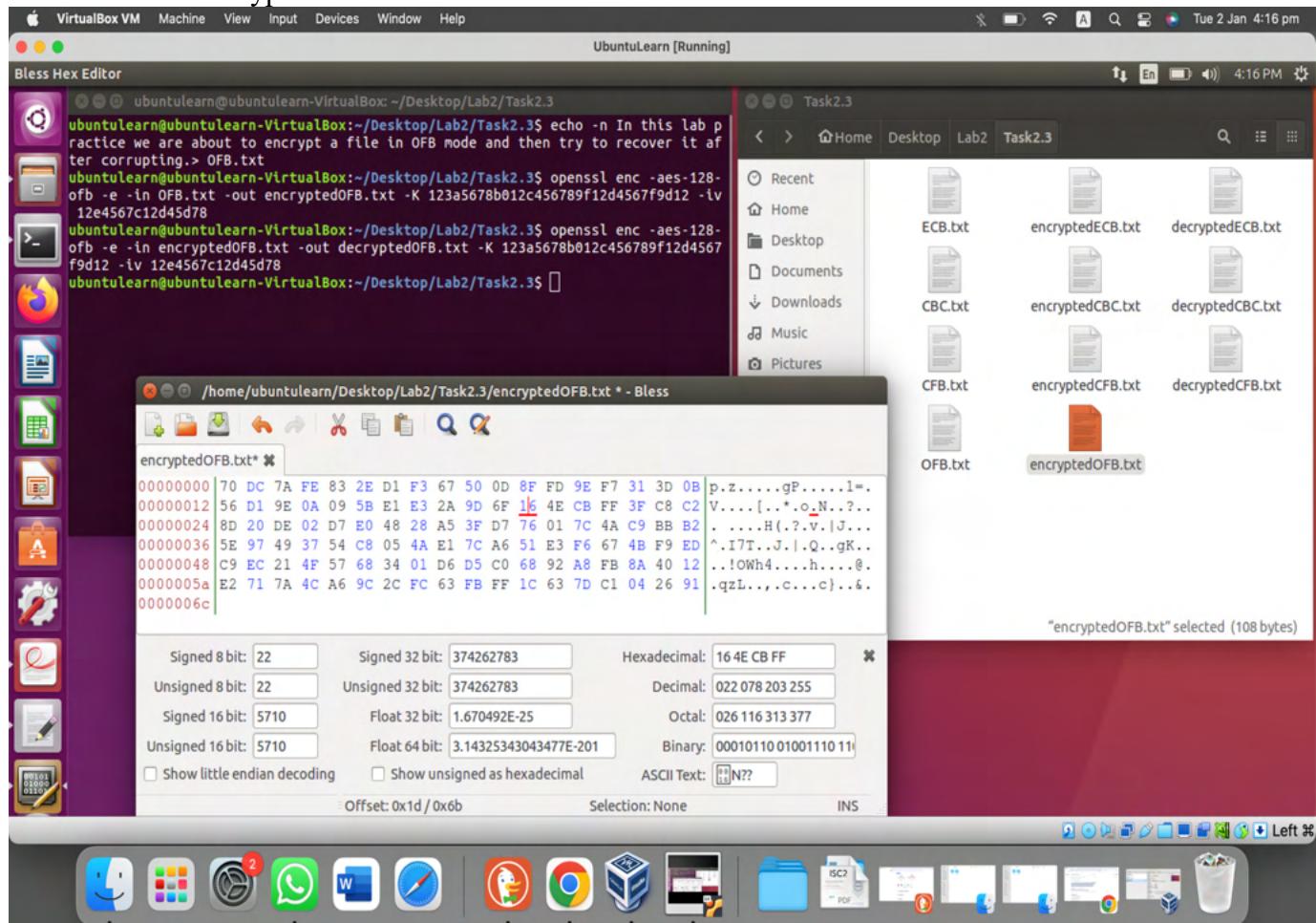
Plaintext for Encryption in OFB



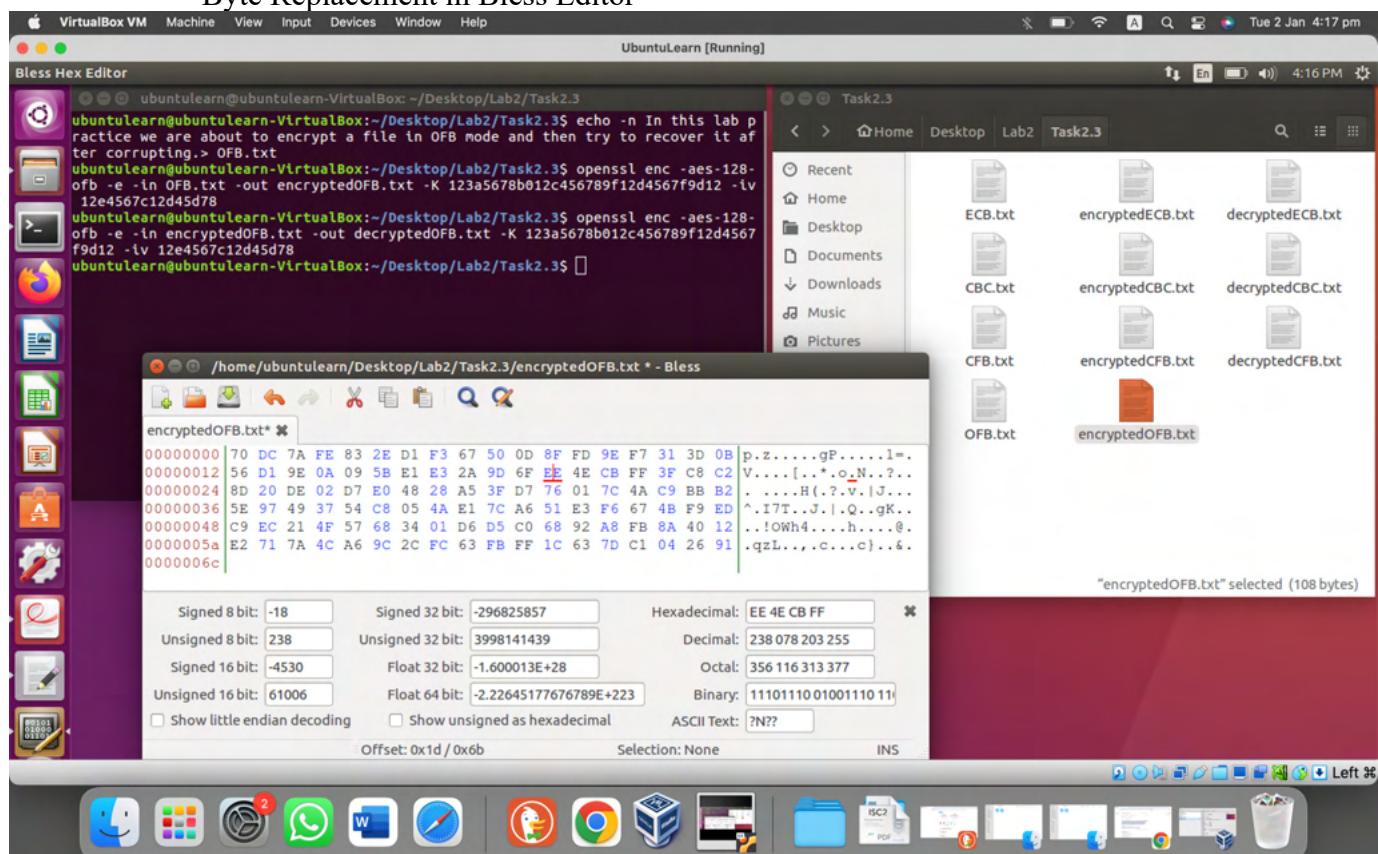
Encryption in OFB



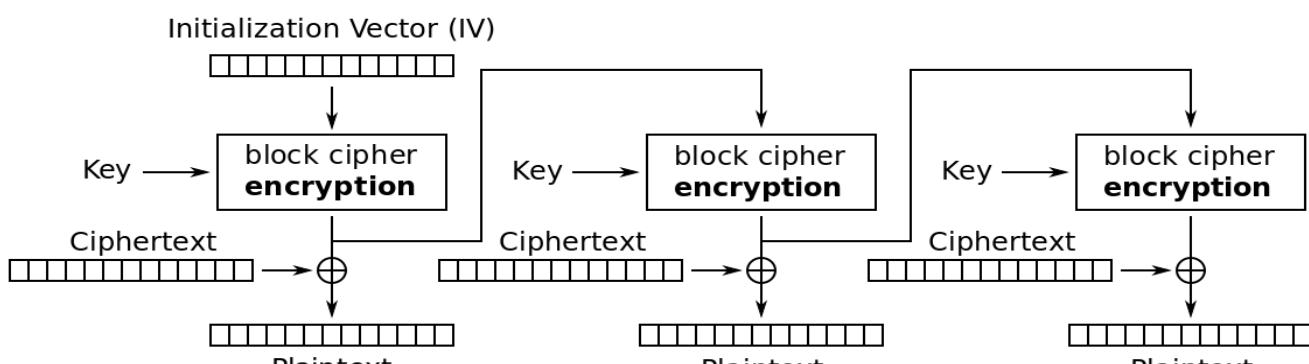
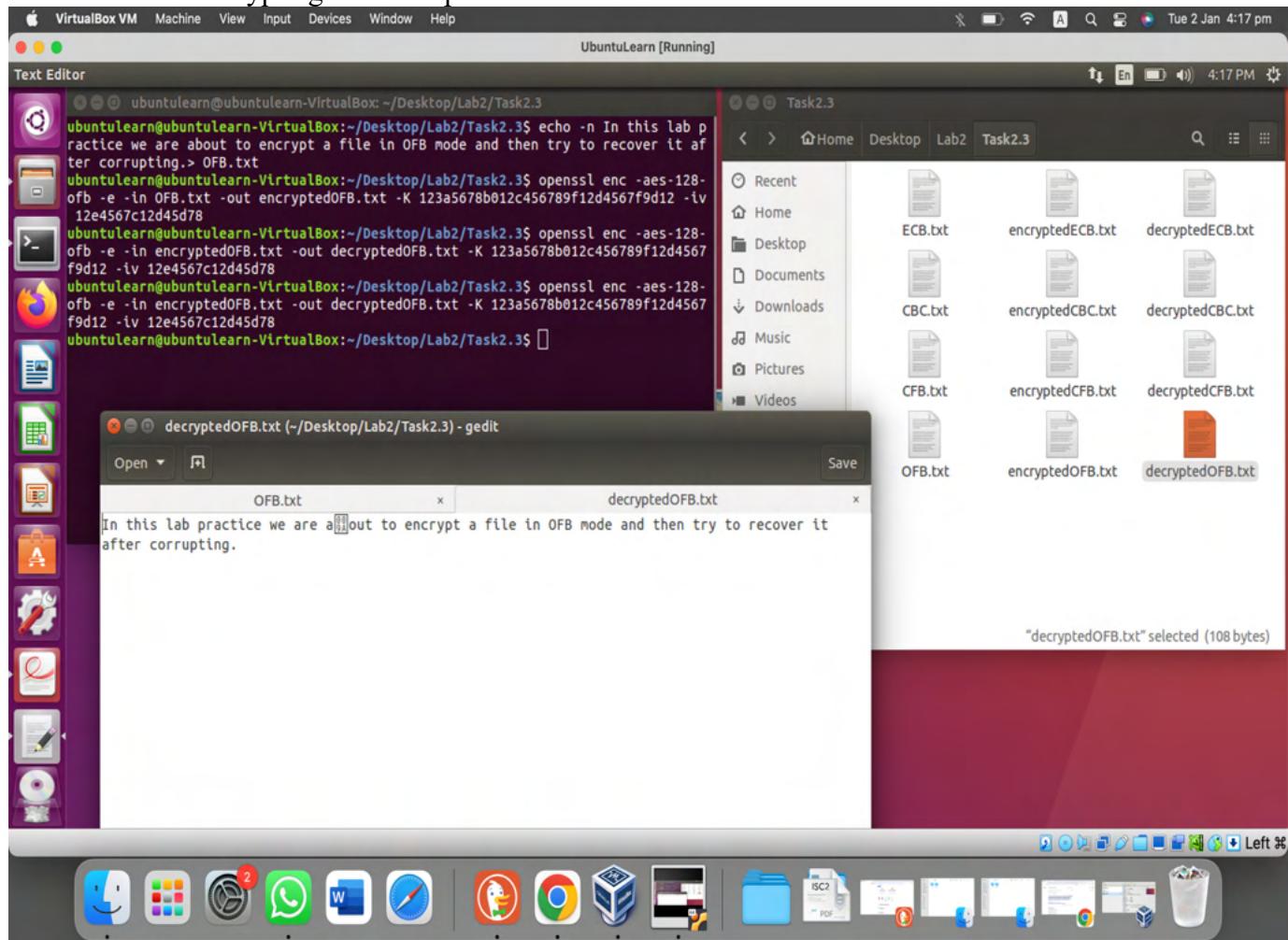
Encrypted File in Bless Editor



Byte Replacement in Bless Editor



Decrypting the Corrupted File



Output Feedback (OFB) mode decryption

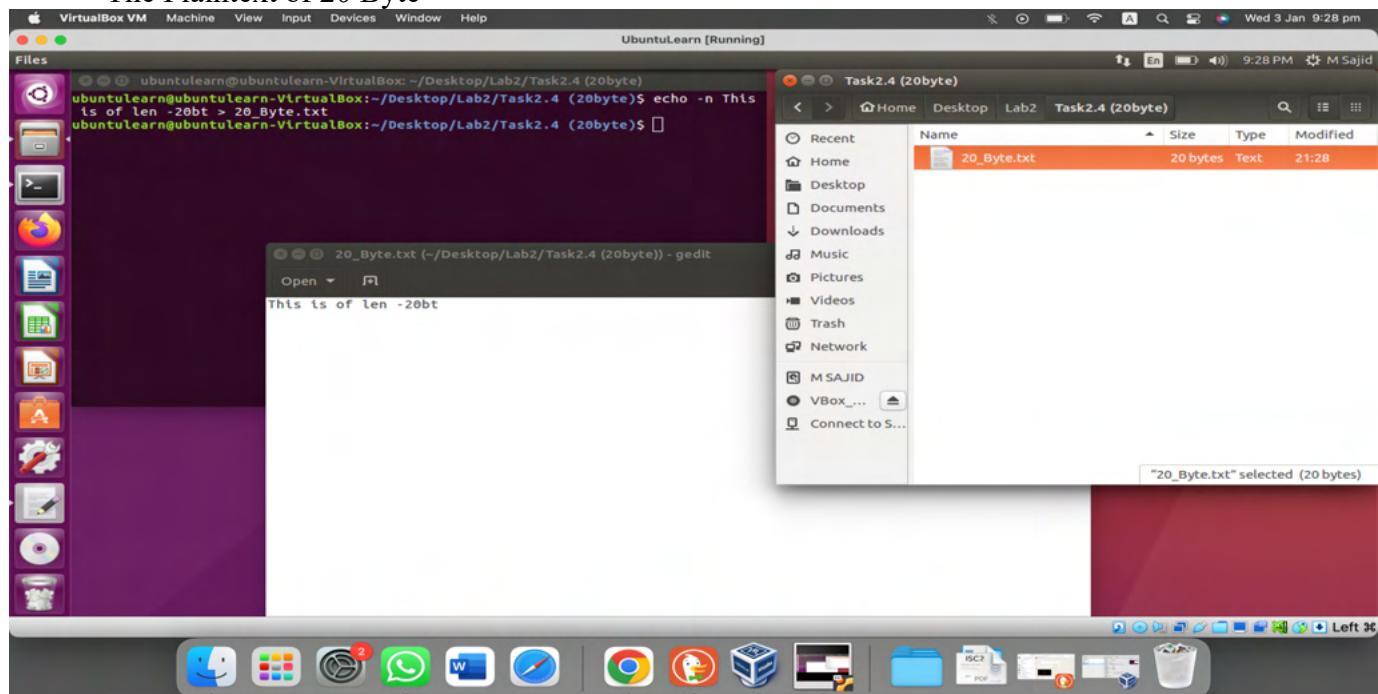
1st block is recovered, 1 byte of second block is corrupted, Rest of the content is recovered after decryption. (Please observe _ character at 30th byte is corrupted, rest of the content is same as original text)

2.4. PADDING

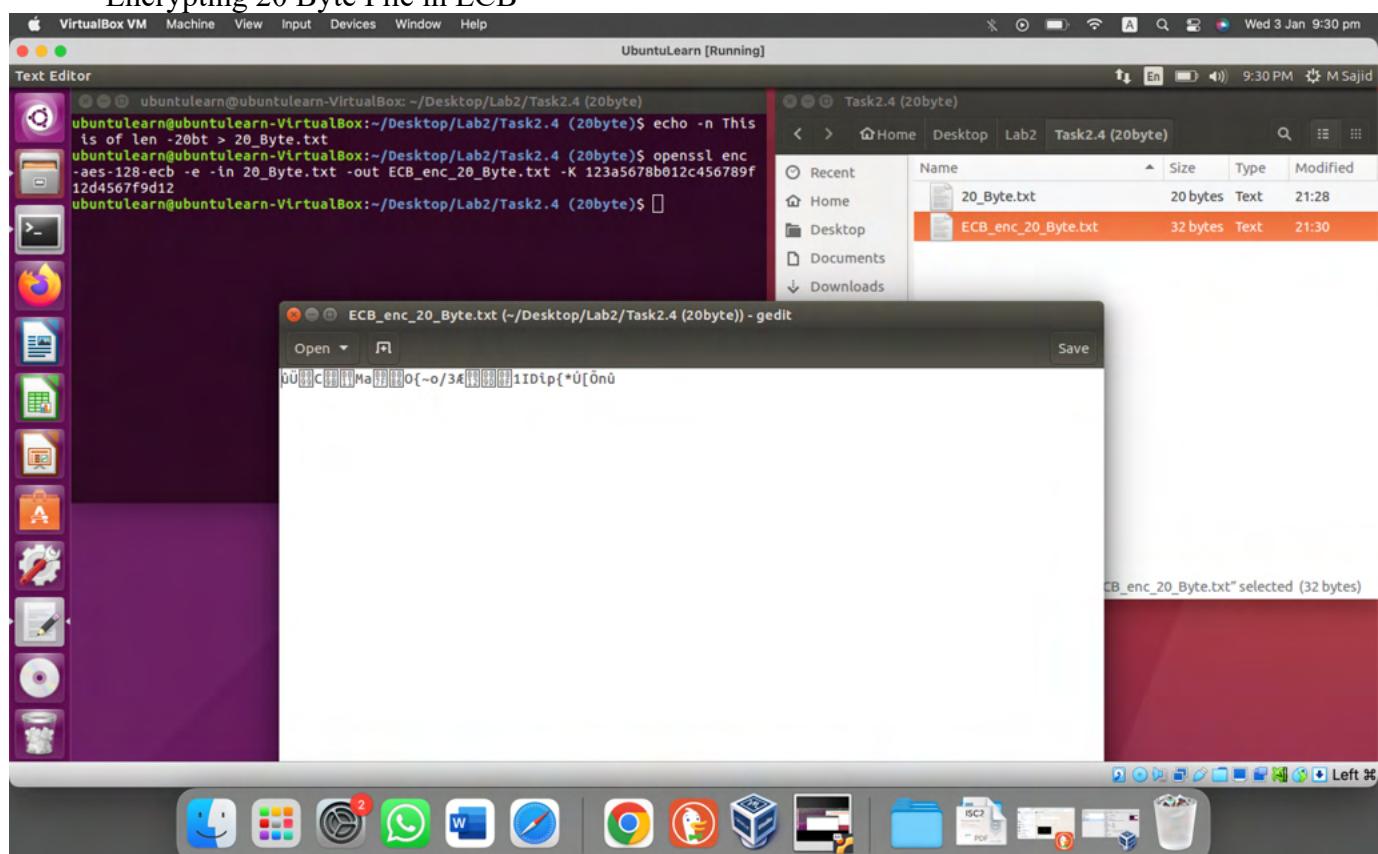
When the size of plaintext is not a multiple of the block size, padding is required. openssl uses PKCS5 standard for its padding. We will experiment to figure out the paddings in the AES encryption for text of length 20 and 32 bytes. In AES-128 each block should be 16 bytes long to encrypt. Any size that is not multiple of 16 is padded accordingly

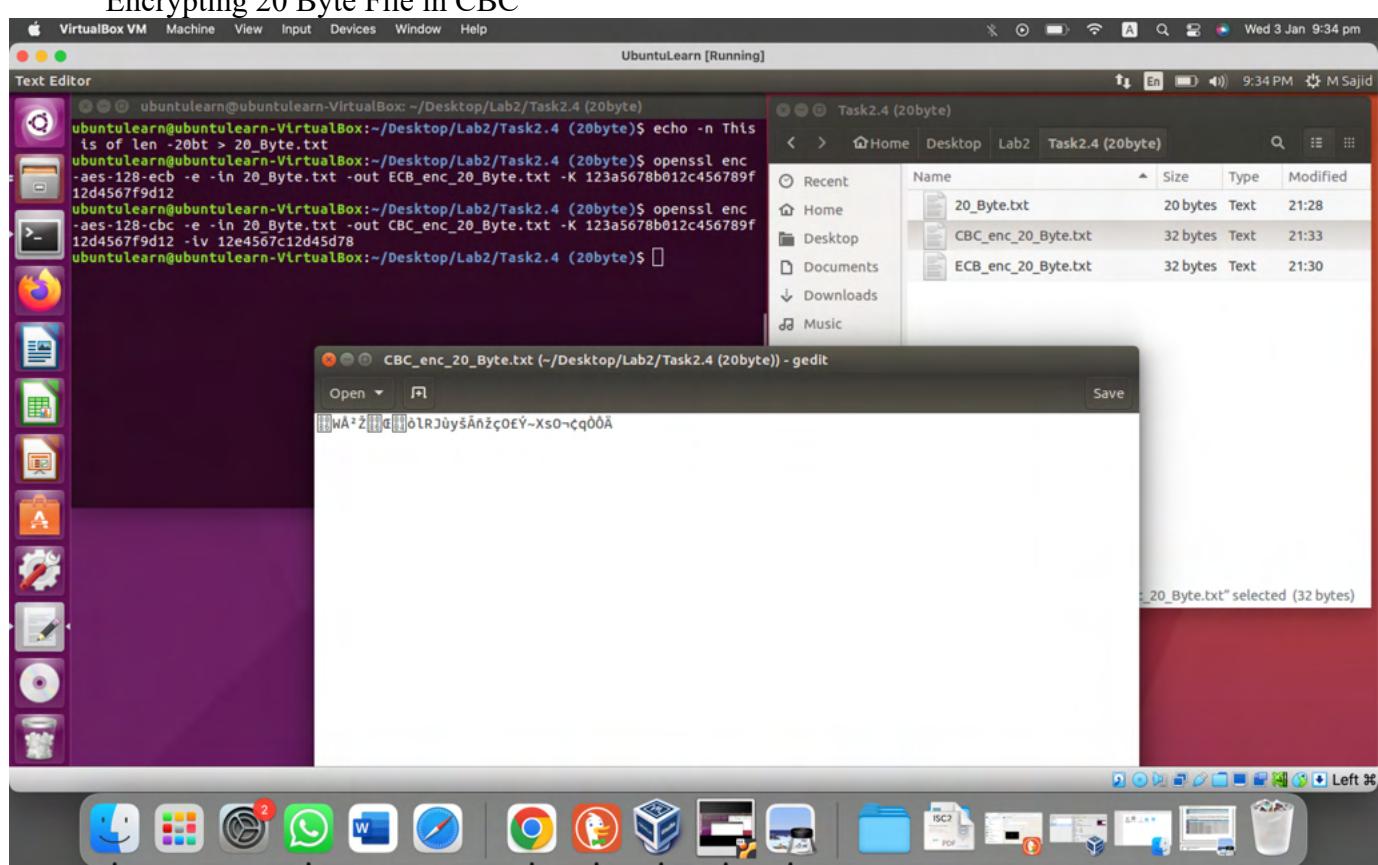
a. 20 Byte File Encryption

The Plaintext of 20 Byte

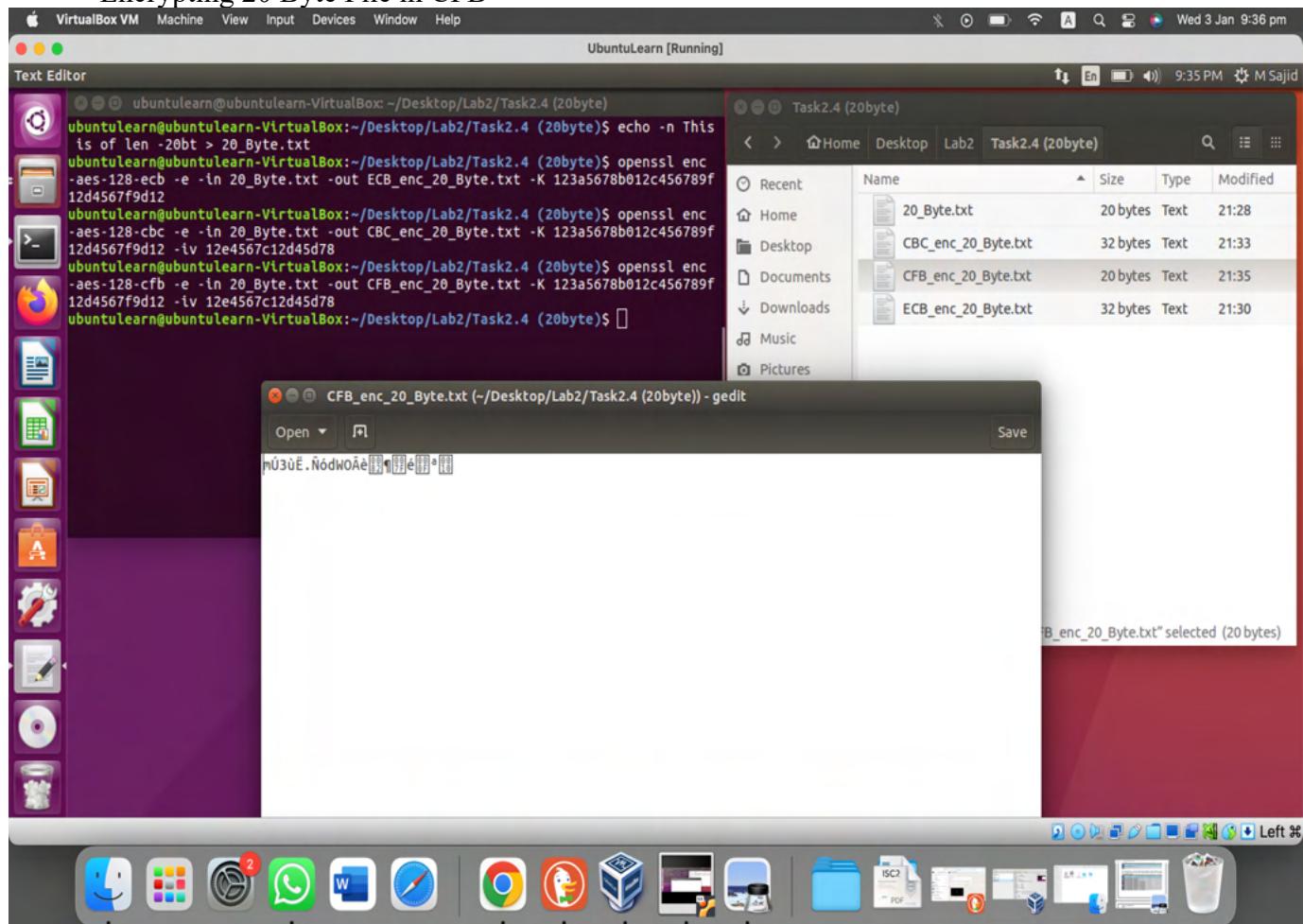


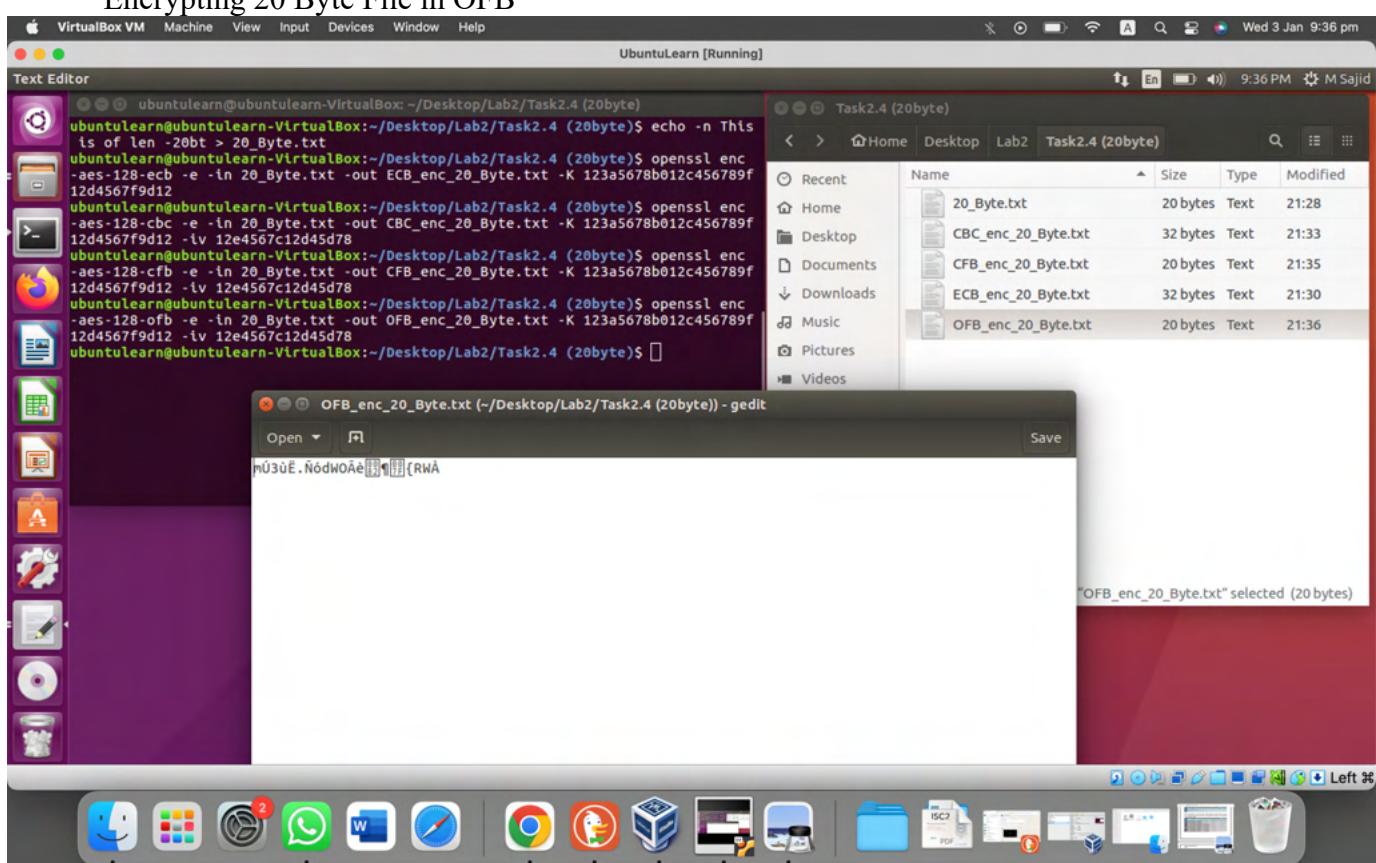
Encrypting 20 Byte File in ECB



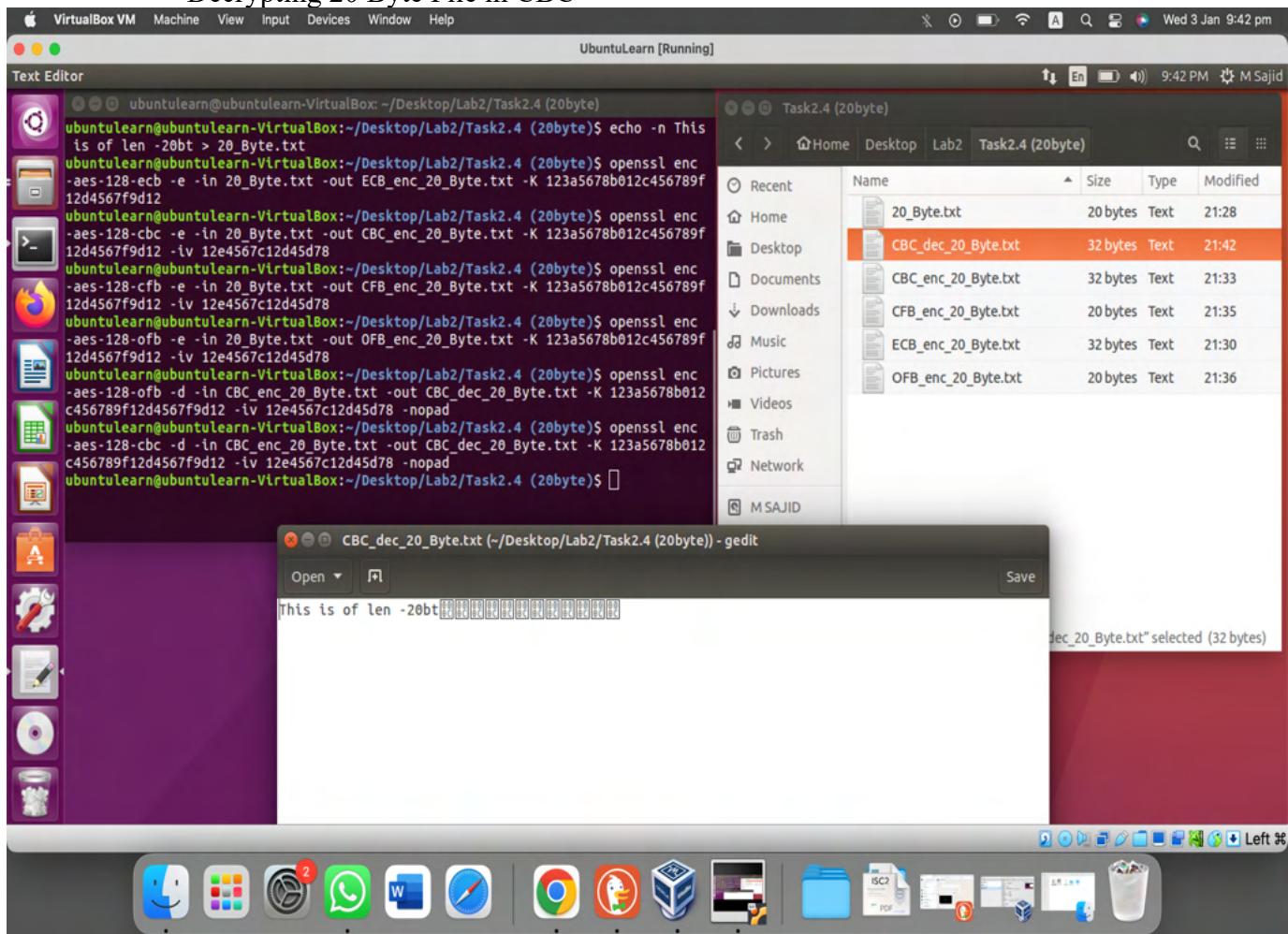


Encrypting 20 Byte File in CFB



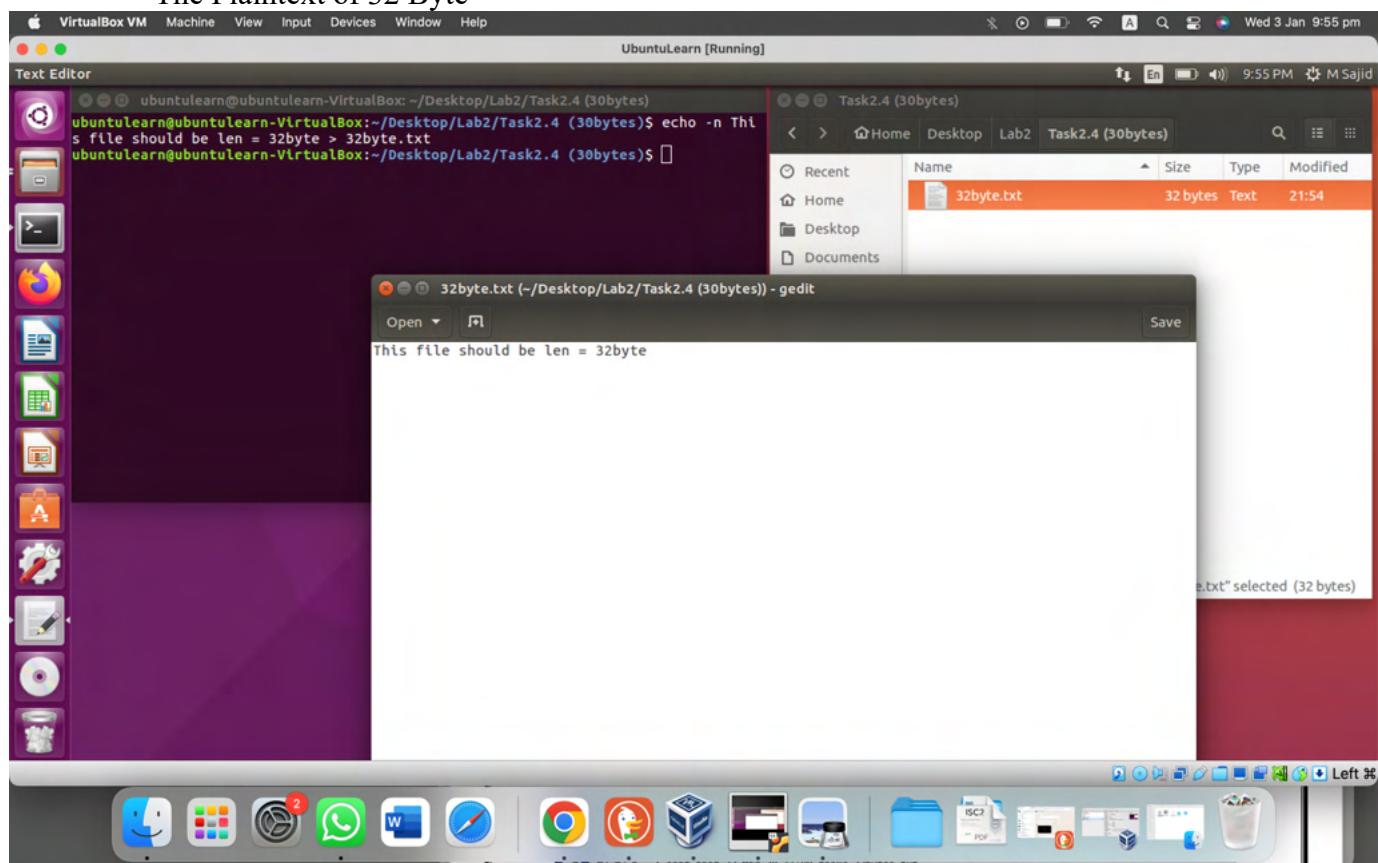


Decrypting 20 Byte File in CBC

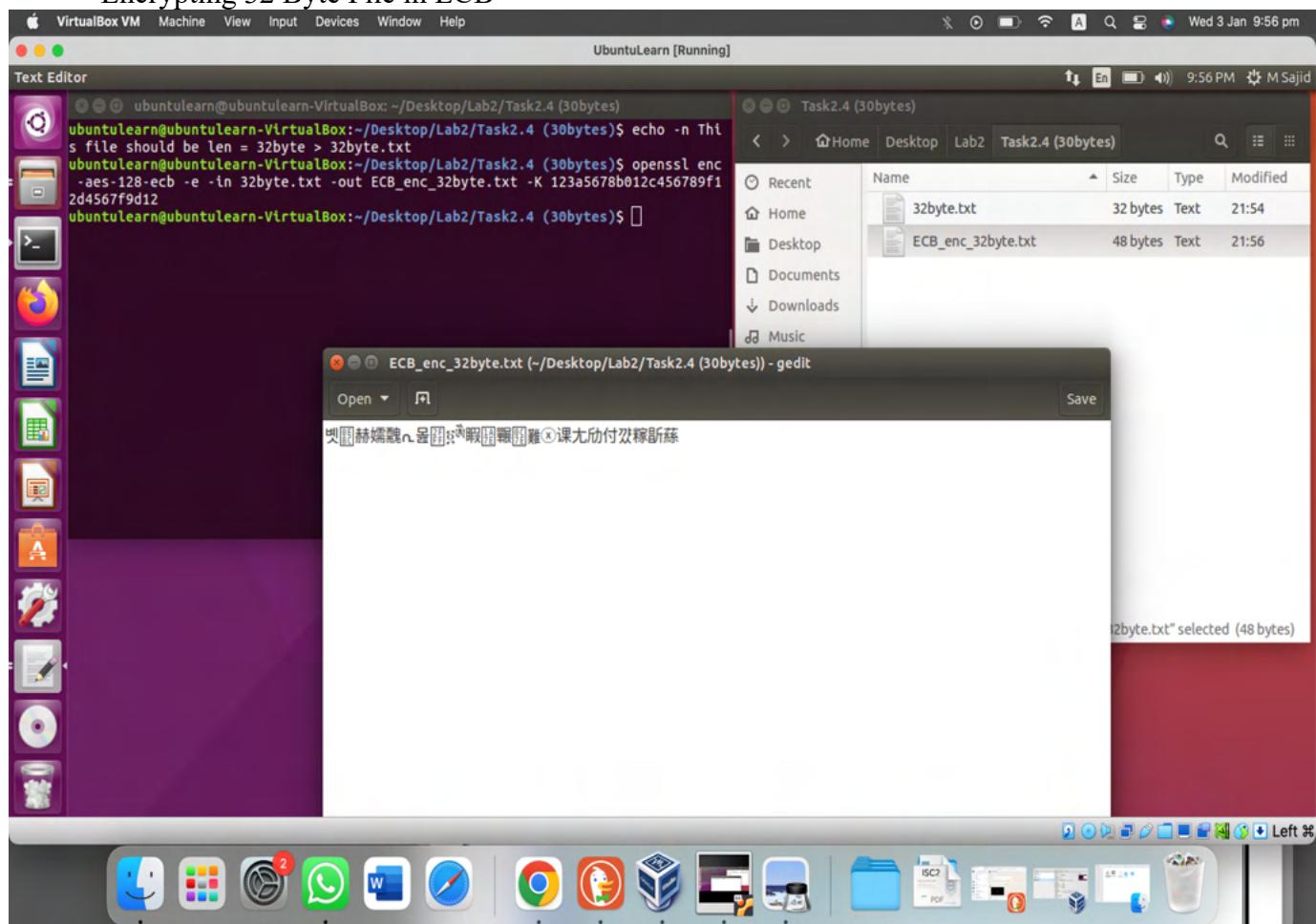


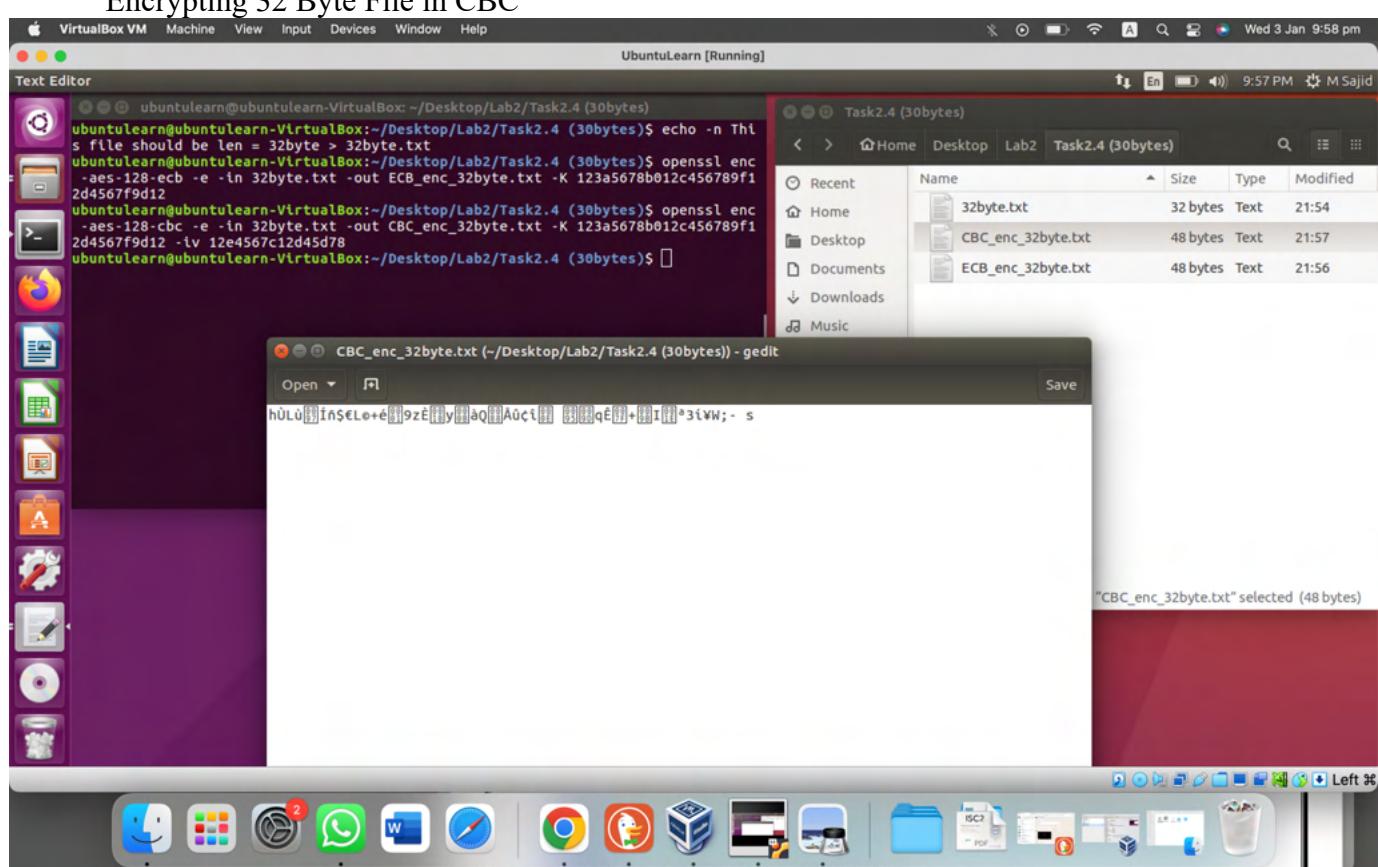
b. 32 Byte File Encryption

The Plaintext of 32 Byte

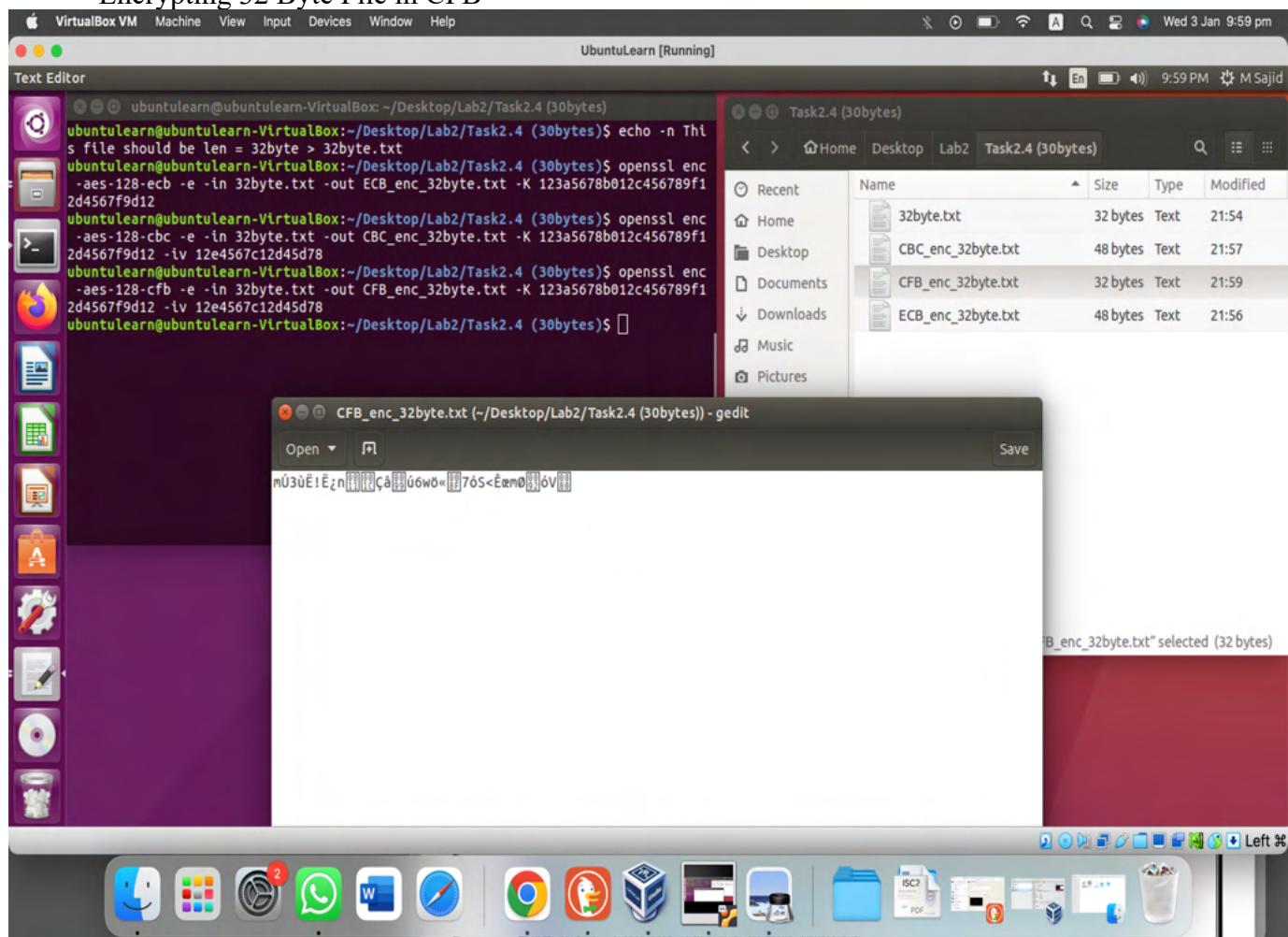


Encrypting 32 Byte File in ECB

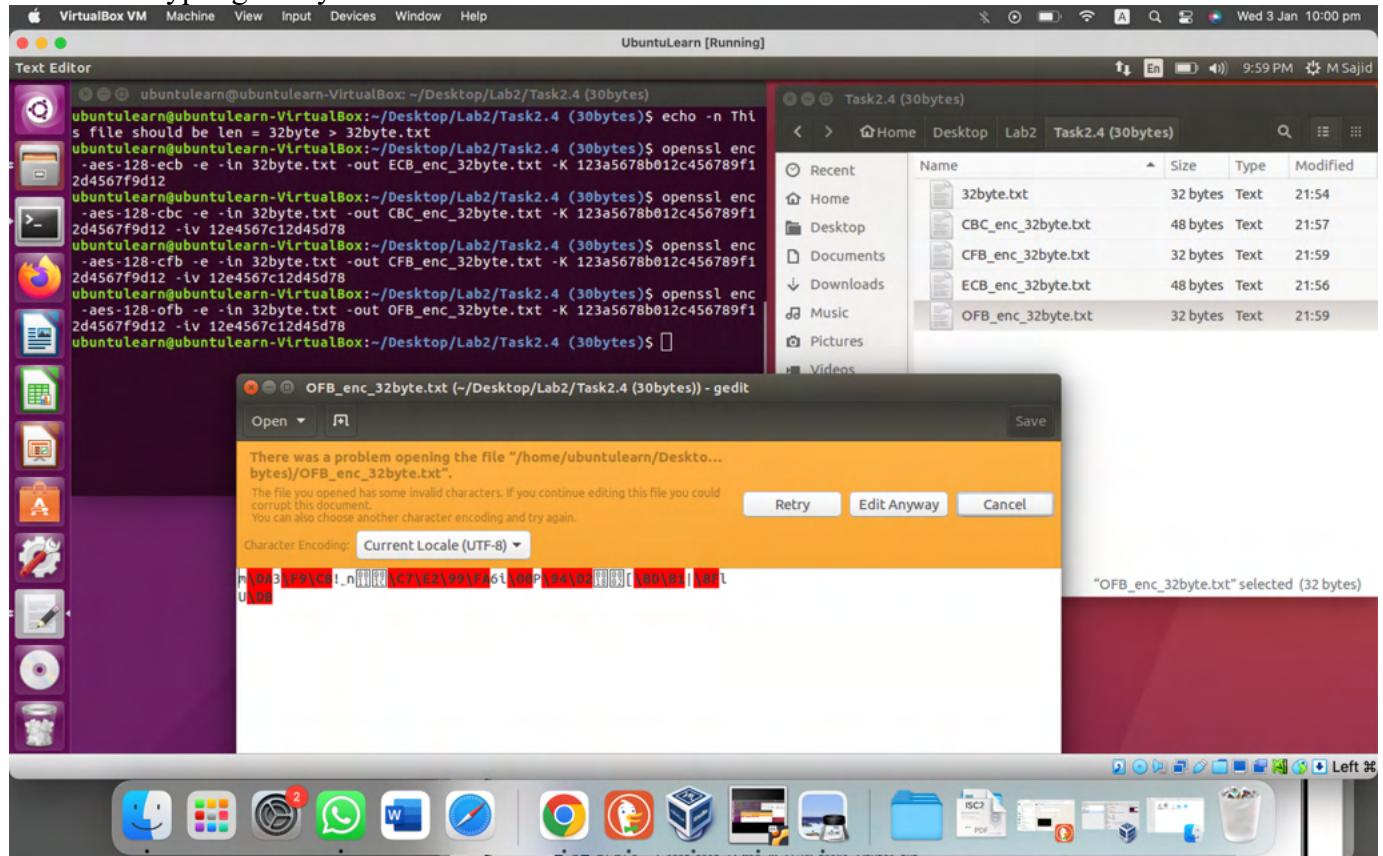




Encrypting 32 Byte File in CFB



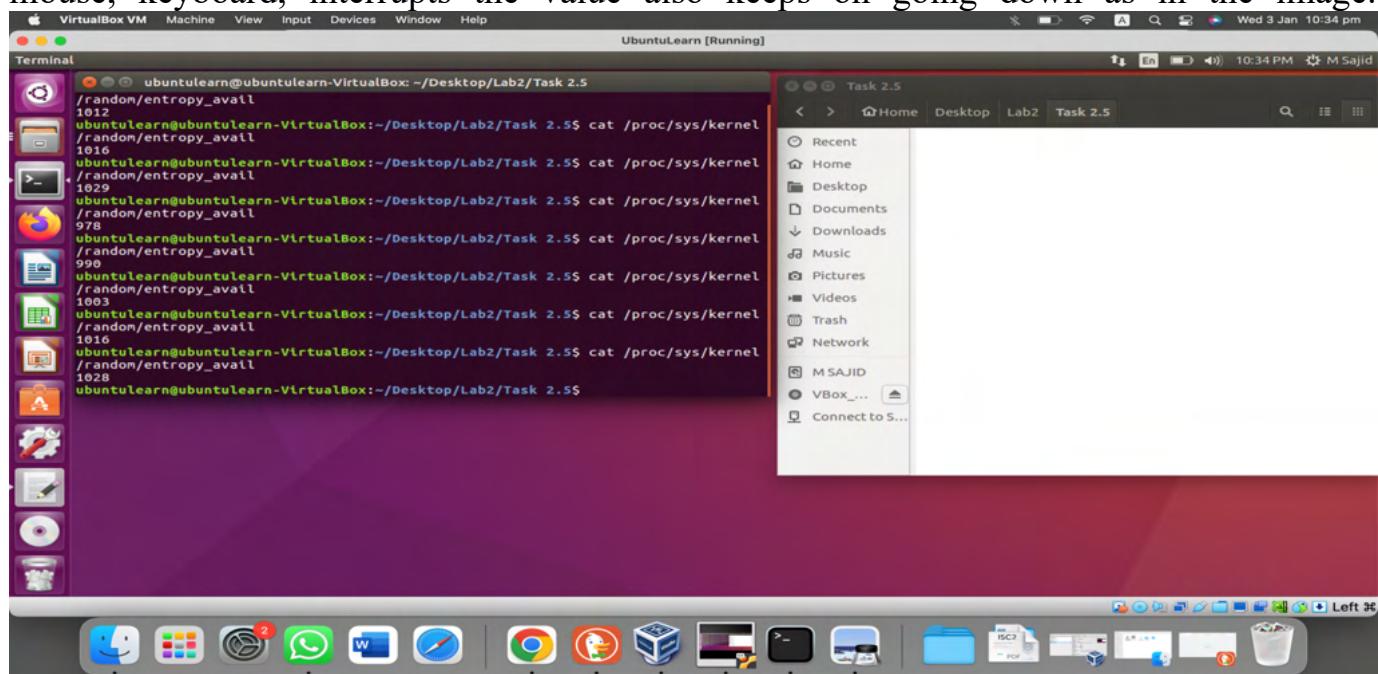
Encrypting 32 Byte File in OFB



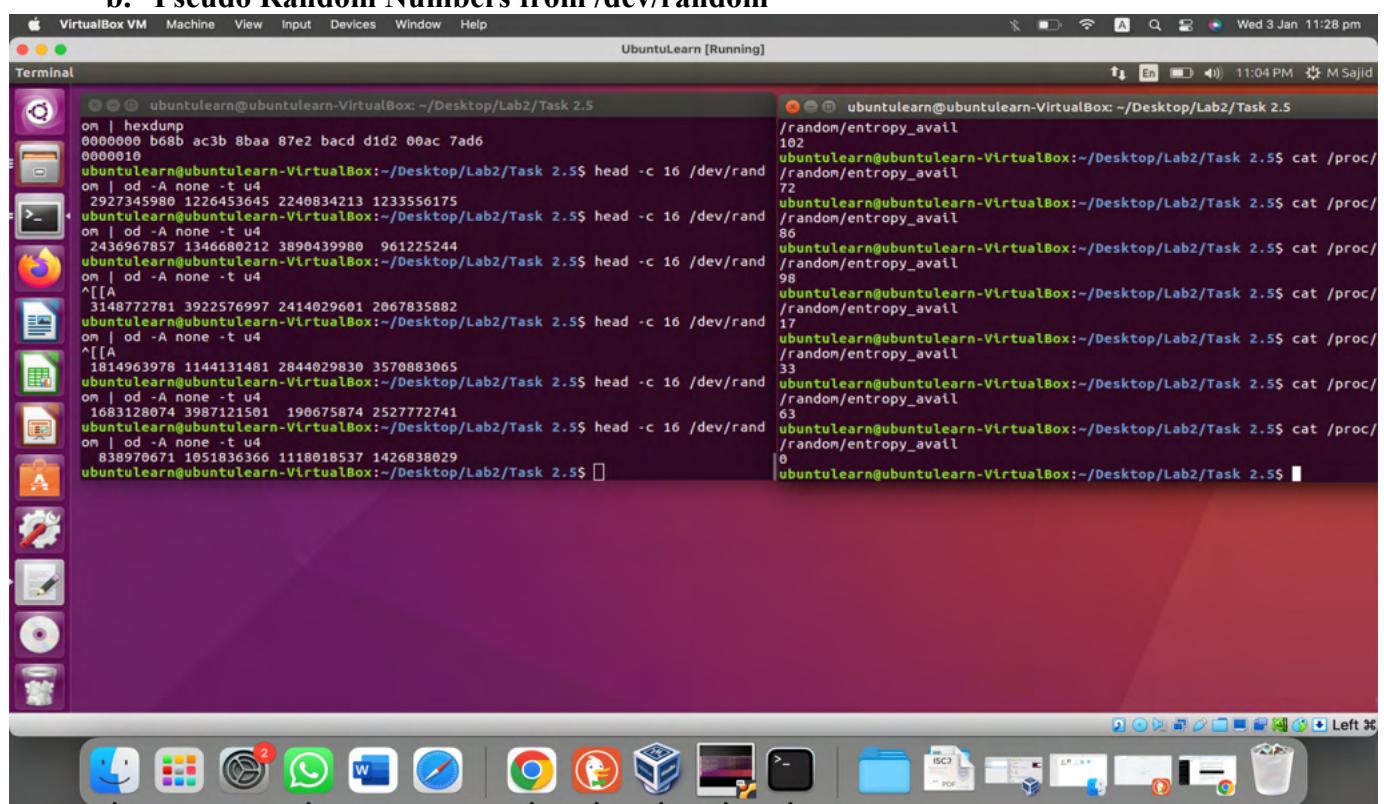
2.5. Pseudo Random Number Generation

a. Measure the Entropy of Kernel

Software might not be good at generating randomness, so it has to resort to physical world to gain the randomness. Some events can be finishing time of block device requests, inter-key press timings, mouse movements etc. Randomness is measured using entropy. It simply means how many bits of random numbers the system currently has. As we keep on moving the mouse and type some command before executing the “cat /proc/sys/kernel/random/entropy_avail” value increases with randomness like moving mouse and giving inputs through keyboard. As the system goes idle with less randomness of mouse, keyboard, interrupts the value also keeps on going down as in the image:

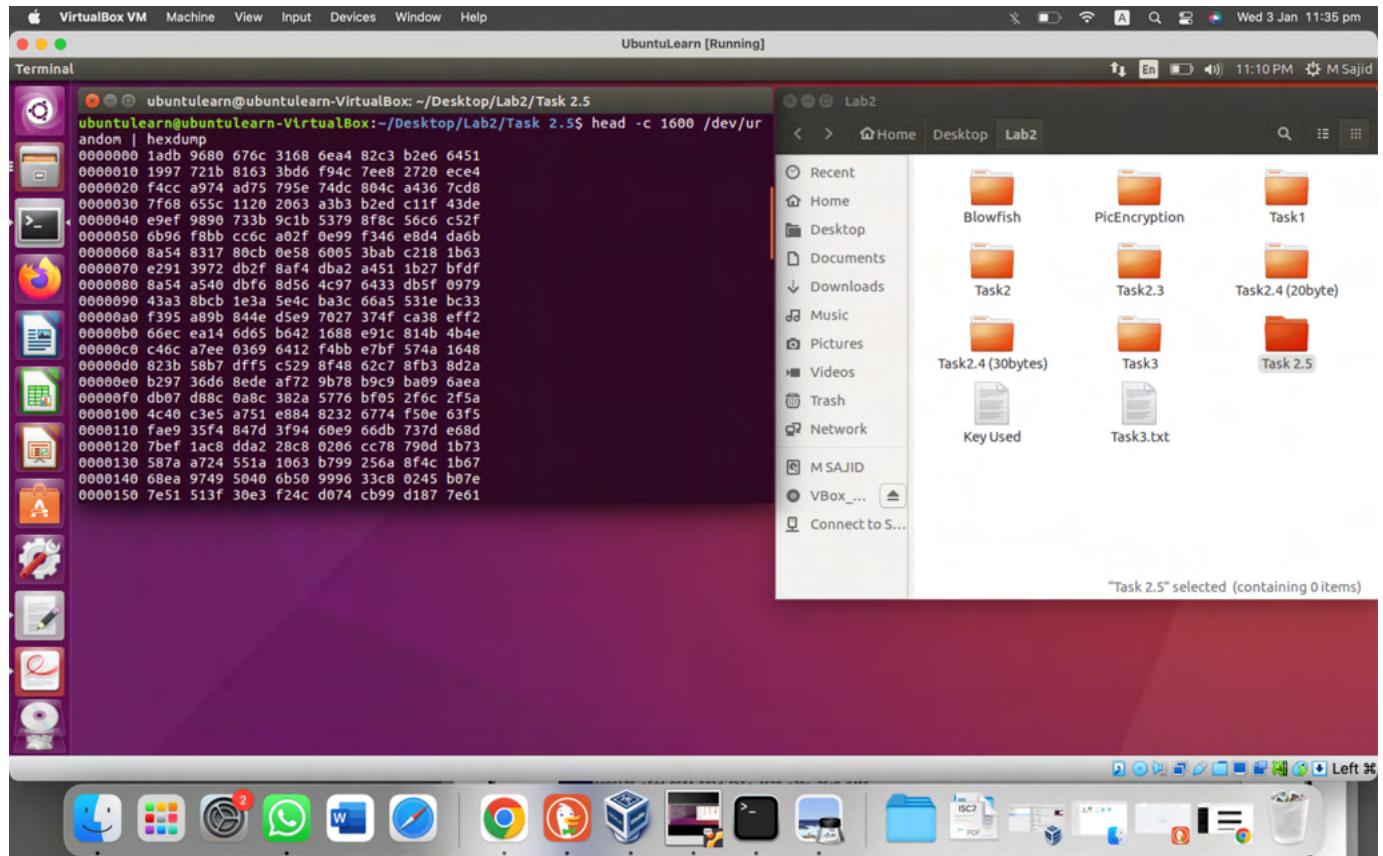


b. Pseudo Random Numbers from /dev/random



c. Get Random Numbers from /dev/urandom

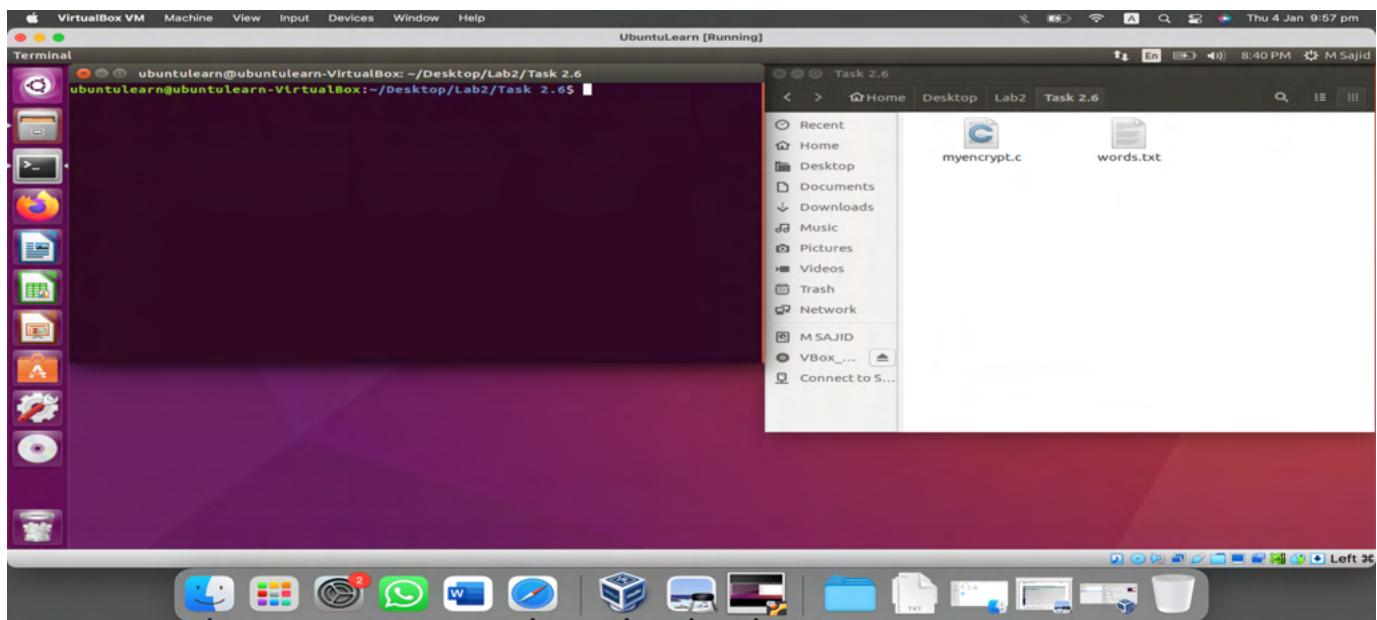
/dev/urandom device doesn't block even if the entropy of the pool runs low. /dev/urandom tries to reseed when new random data is available. It is recommended to use /dev/urandom because /dev/random may lead to denial of service attacks. With /dev/urandom it was not blocked even after running multiple iterations

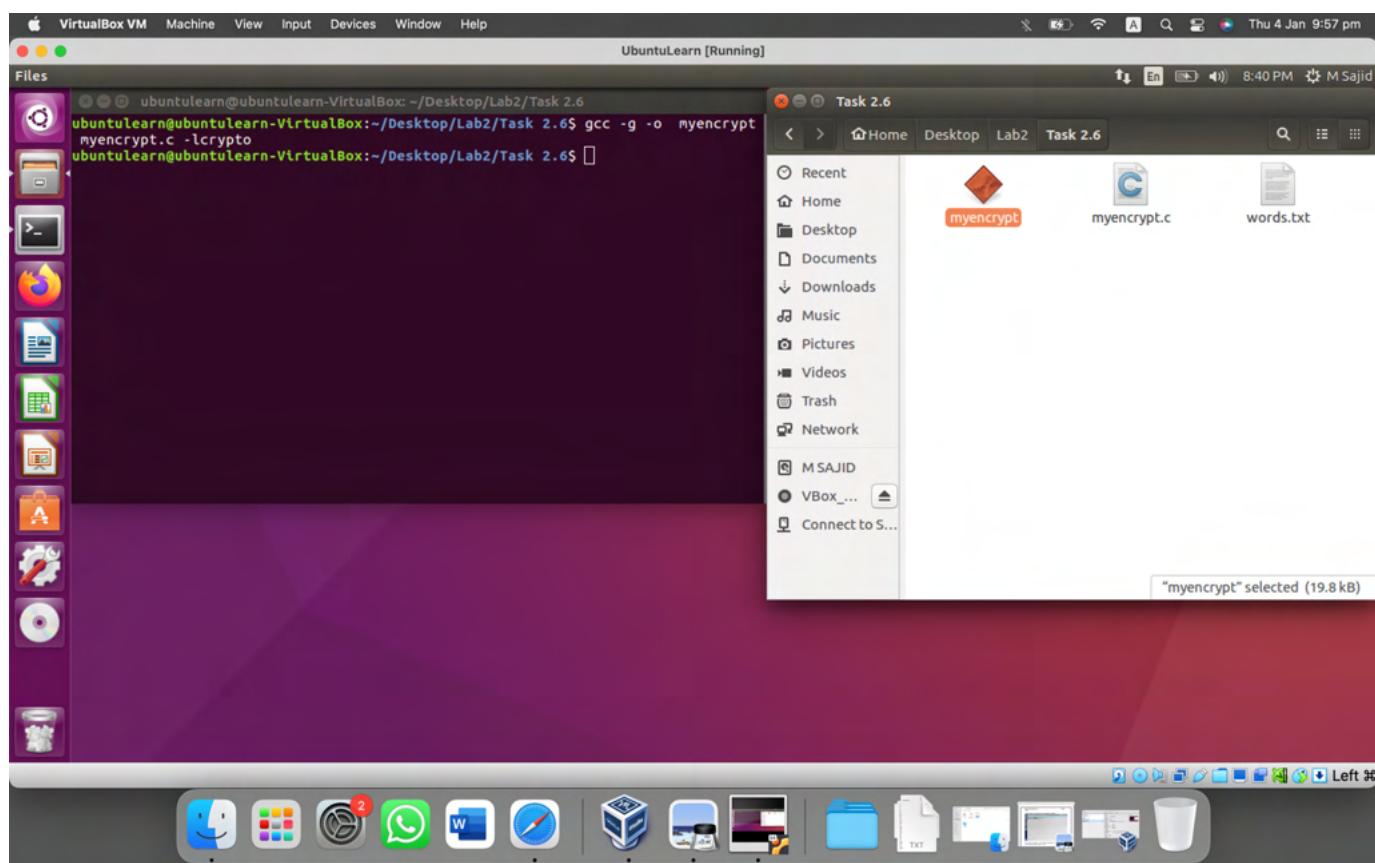
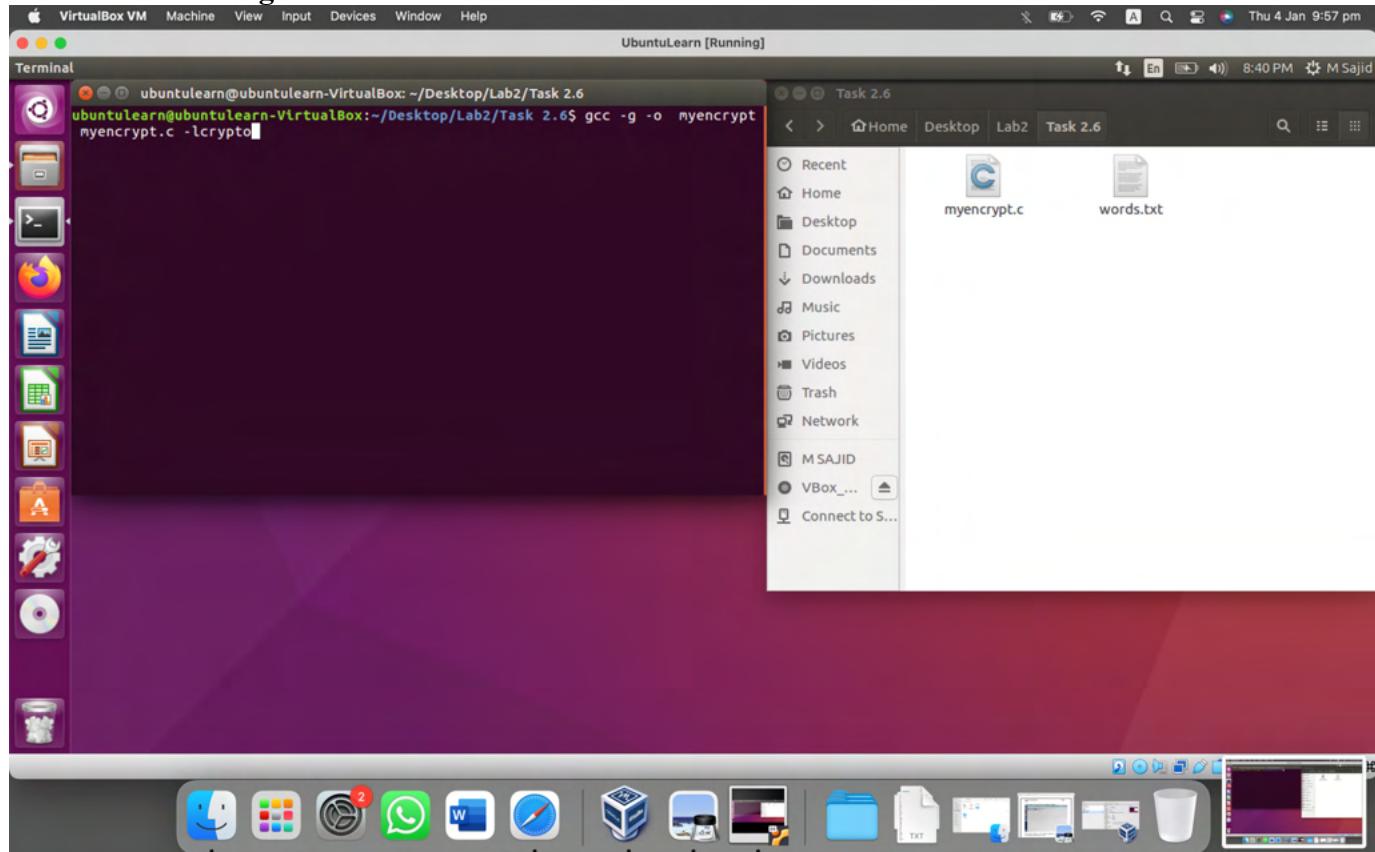


2.6. Programming using the Crypto Library

In this task, we are given a plaintext and ciphertext, and we are using aes-128-cbc to generate ciphertext from plaintext. We are told that initialization vectors are zero. (-iv argument in openssl enc command). Another clue is that the key used to encrypt the plain text is an English word shorter than 16 characters. Since the key has less than 16 bytes, space characters are appended to the end of word to form 128 bits. We have to write a program to find this key.

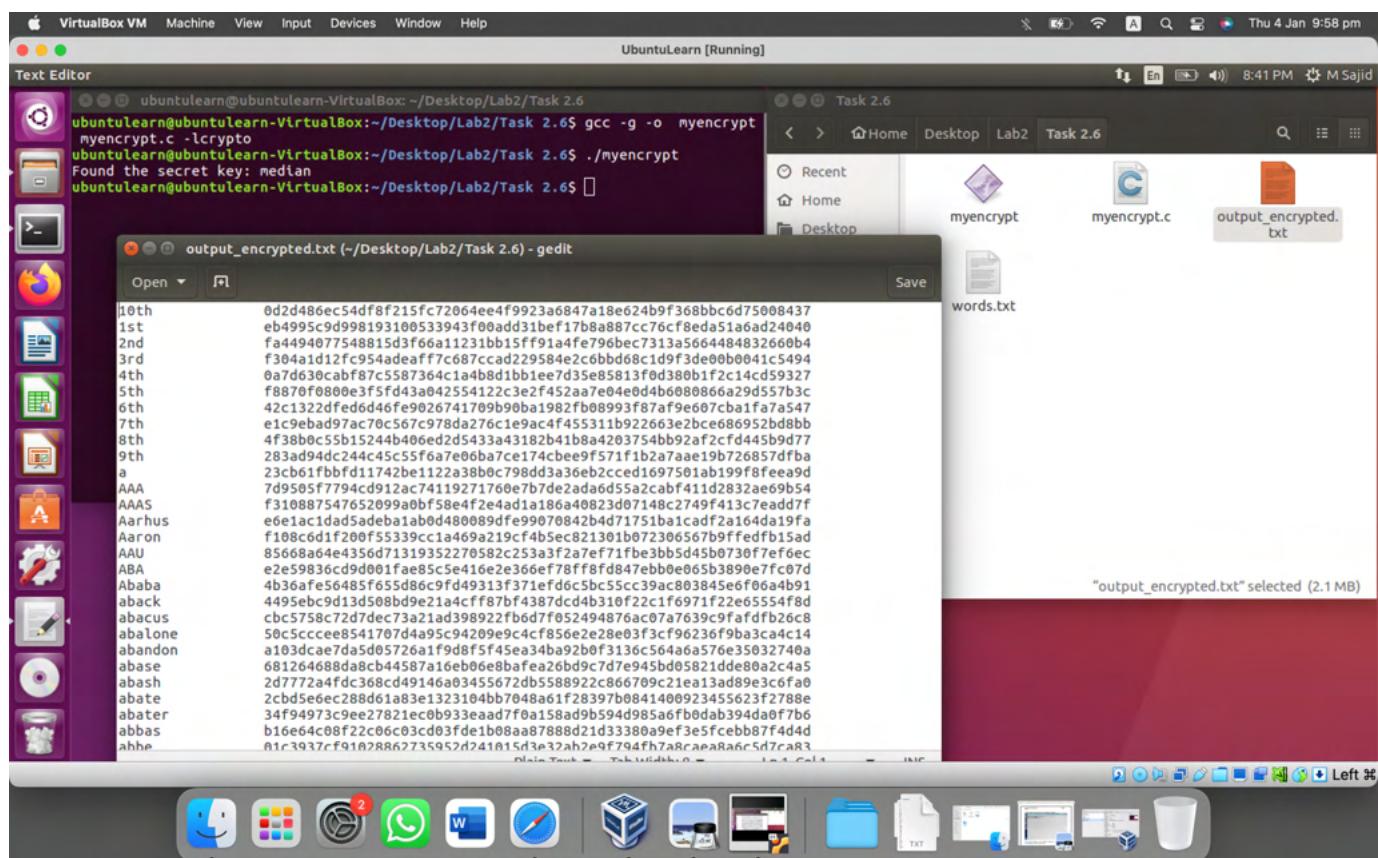
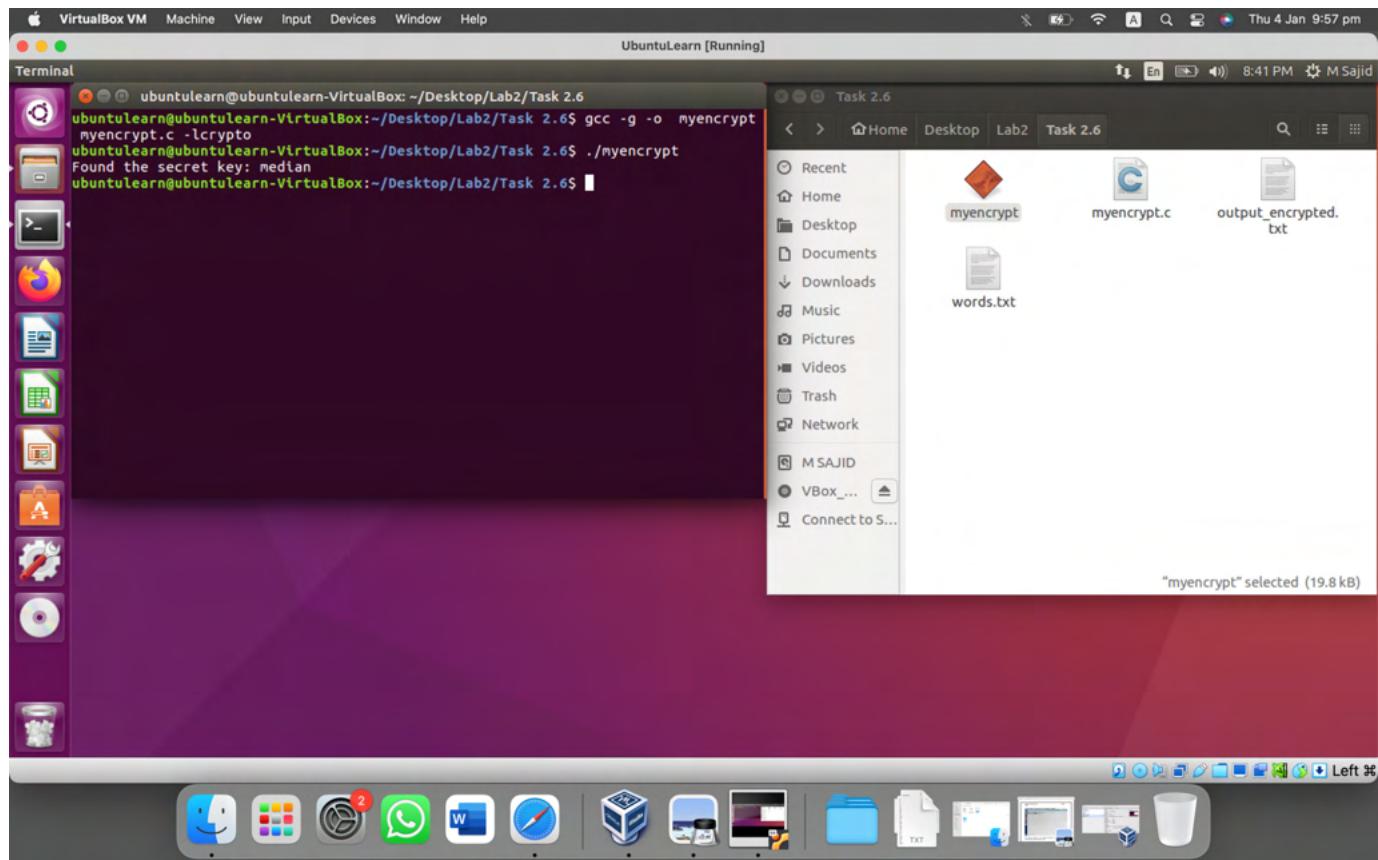
a. The Available Files



b. Running the Command

c. Program Output

The secret key in our example is “median”



d. Actual Program:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<openssl/evp.h>
void pad_with_spaces(char* singleword, int len) {
    int curlen = strlen(singleword);
    while (curlen < len) {
        singleword[curlen] = ' ';
        curlen++;
    }
    singleword[len] = '\0';
}
/*One problem I found out is that we need to pass unsigned char *buf (not char
 *buf) because
otherwise each character is considered signed and hex digits are mostly printed
as ffffff... */
void print_result_output_file(char singleword[], unsigned char *buf, int len,
FILE* outputList) {
    for (int i = 0; i < 16; i++) {
        fprintf(outputList, "%c", singleword[i]);
    }
    /* Add a space between key and encrypted string */
    fprintf(outputList, "%c", ' ');
    for (int i = 0; i < len; i++) {
        fprintf(outputList, "%02x", buf[i]);
    }
    /* Add an end of file for each pair of key and encrypted string */
    fprintf(outputList, "%c", '\n');
}
int main() {
    int encryptp = 1;
    // use aes-128-cbc to generate ciphertext from plaintext
    unsigned char key[] = "00112233445566778899aabbcdddeeff";
    //unsigned char iv[] = "0102030405060708";
    unsigned char iv[16] = {0}; //0000000000000000 ; // "0102030405060708";
    /* for (int i = 0; i < 16; i++) {
        iv[i] = 0x00;
    }*/
    /* Don't set key or IV right away ; we want to check lengths */
    EVP_CIPHER_CTX ctx;
    EVP_CIPHER_CTX_init(&ctx);
    EVP_CipherInit_ex(&ctx, EVP_aes_128_cbc(), NULL, NULL, NULL, encryptp);
    OPENSSL_assert(EVP_CIPHER_CTX_key_length(&ctx) == 16);
    OPENSSL_assert(EVP_CIPHER_CTX_iv_length(&ctx) == 16);
    FILE *wordsList, *outputList;
    wordsList = fopen("words.txt", "r");
    outputList = fopen("output_encrypted.txt", "a+");
    if (wordsList < 0) {
        perror("Error ");
    }
    char inptext[] = "This is a top secret.";
    // printf("Plain string is: %s and length = %d\n", inptext, strlen(inptext));
    char
    ciphertext[]
    =
    "8d20e5056a8d24d0462ce74e4904c1b513e10d1df4a2ef2ad4540faelca0aaaf9";
```

```
// printf("Cipher string is: %s and length = %d\n", ciphertext,
strlen(ciphertext));
FILE* output;
unsigned char singleword[16];
int count = 0;
int templen = 0;
unsigned char outbuf[1024 + EVP_MAX_BLOCK_LENGTH];
// memset(&outbuf, 0, sizeof(outbuf));
int outlen, templen;
// printf("Before While Plain string is: %s and length = %d\n", inptext,
strlen(inptext));
while ((fgets(singleword, 16, wordsList) != NULL) ) {
count++;
/* Since the words.txt ends with \n for each line, we replace that '\n' with '\0'
to end that string */
singleword[strlen(singleword)-1] = '\0';
// printf("Len of string is %d\n", strlen(singleword));
if (strlen(singleword) < 16) {
pad_with_spaces(singleword, 16);
}
for(int i = 0; i < strlen(inptext); i++) {
printf("%c ", inptext[i]);
}
// printf("\nBefore EncryptInit String is %s , len of inptext is %d\n",inptext,
strlen(inptext));
EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, singleword, iv);
// printf("String is %s , len of inptext is %d\n",inptext, strlen(inptext));
// Here I am hardcoding the inptext string. need to debug why when inptext is
given as
// input it is producing outlen as zero
if (!EVP_CipherUpdate(&ctx, outbuf, &outlen, "This is a top secret.", 21)) {
// if (!EVP_EncryptUpdate(&ctx, outbuf, &outlen, inptext, strlen(inptext))) {
/* Error */
EVP_CIPHER_CTX_cleanup(&ctx);
return 0;
}
// printf("Outlen Before final is %d\n", outlen);
/* Clean up any last bytes left in the output buffer */
if (!EVP_CipherFinal_ex(&ctx, outbuf+outlen, &templen)) {
// if (!EVP_EncryptFinal_ex(&ctx, outbuf+outlen, &templen)) {
/* Error */
EVP_CIPHER_CTX_cleanup(&ctx);
return 0;
}
// printf("Outlen is %d\n", outlen);
// printf("templen is %d\n", templen);
outlen += templen;
/* We create double the size of outbuf because for every character
in outbuf we have 2 hexadecimal characters in buf_hex_encrypt */
char buf_hex_encrypt[2*outlen + 1];
char *buf_operate = buf_hex_encrypt;
for (int i = 0; i < outlen; i++) {
buf_operate += sprintf(buf_operate, "%02x", outbuf[i]);
}
/* End the string with \0 value */
*(buf_operate+1) = '\0';
// printf("Key is %s\n", singleword);
// printf("Outlen is %d\n", outlen);
if (strcmp(buf_hex_encrypt, ciphertext) == 0 ) {
printf("Found the secret key: %s\n", singleword);
}
print_result_output_file(singleword, outbuf, outlen, outputList);
```

```
memset(singleword, 0, sizeof(singleword));  
}  
/* else { }*/  
EVP_CIPHER_CTX_cleanup(&ctx);  
fclose(wordsList);  
fclose(outputList);  
// perror("Closed successfully");  
return 0;  
}
```