



Al Imam Mohammad ibn Saud University  
College of Computer and Information Sciences  
Computer Science Department  
First semester of 1444



## **CS 334: Information Security**

### **Project: Secret-Key Encryption Lab**

#### **Team Members:**

<b>Student Name:</b>	<b>Academic Number:</b>
Aljowharah Aldawood	
Reem Alothman	
Teef Mohammed Alosan	

#### **Instructor:**

Tahani Albalawi

#### **Section:**

371

#### **Due Date:**

9/11/2022

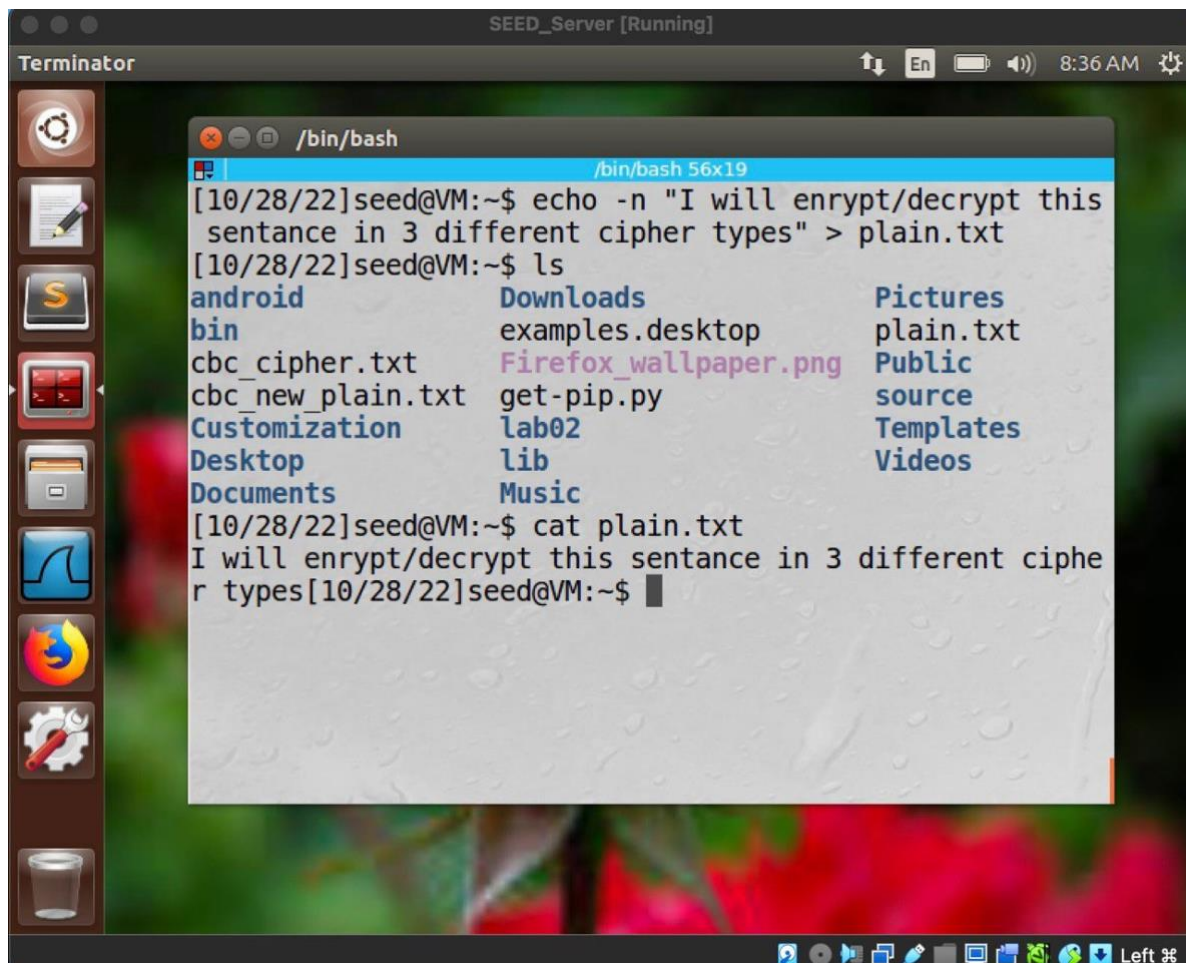
## **Table of Contents:**

Task 2 .....	3
Task 3 .....	13
Task 7 .....	20

## Task 2 : Encryption using different cipher and modes:

Before ciphering the sentence “I will encrypt/decrypt this sentence in 3 different cipher types” has been entered. Each type of the 3 types will be described and explained in detail in this document. The 3 types that have been used in this lab are:

- 1- aes-128-cbc.
- 2- bf-cbc. (blow-fish cipher).
- 3- aes-128-cfb.



```
SEED_Server [Running]
Terminator
/bin/bash
/bin/bash 56x19
[10/28/22]seed@VM:~$ echo -n "I will enrypt/decrypt this
sentence in 3 different cipher types" > plain.txt
[10/28/22]seed@VM:~$ ls
android          Downloads        Pictures
bin              examples.desktop plain.txt
cbc_cipher.txt   Firefox_wallpaper.png Public
cbc_new_plain.txt get-pip.py       source
Customization    lab02            Templates
Desktop          lib              Videos
Documents        Music
[10/28/22]seed@VM:~$ cat plain.txt
I will enrypt/decrypt this sentence in 3 different ciphe
r types[10/28/22]seed@VM:~$
```

### 1- aes-128-cbc:

As shown in the first figure (in red) is the length of the sentence that we will use before ciphering is 63 letters. This cipher type requires a key (-K) and an initial value (-iv).

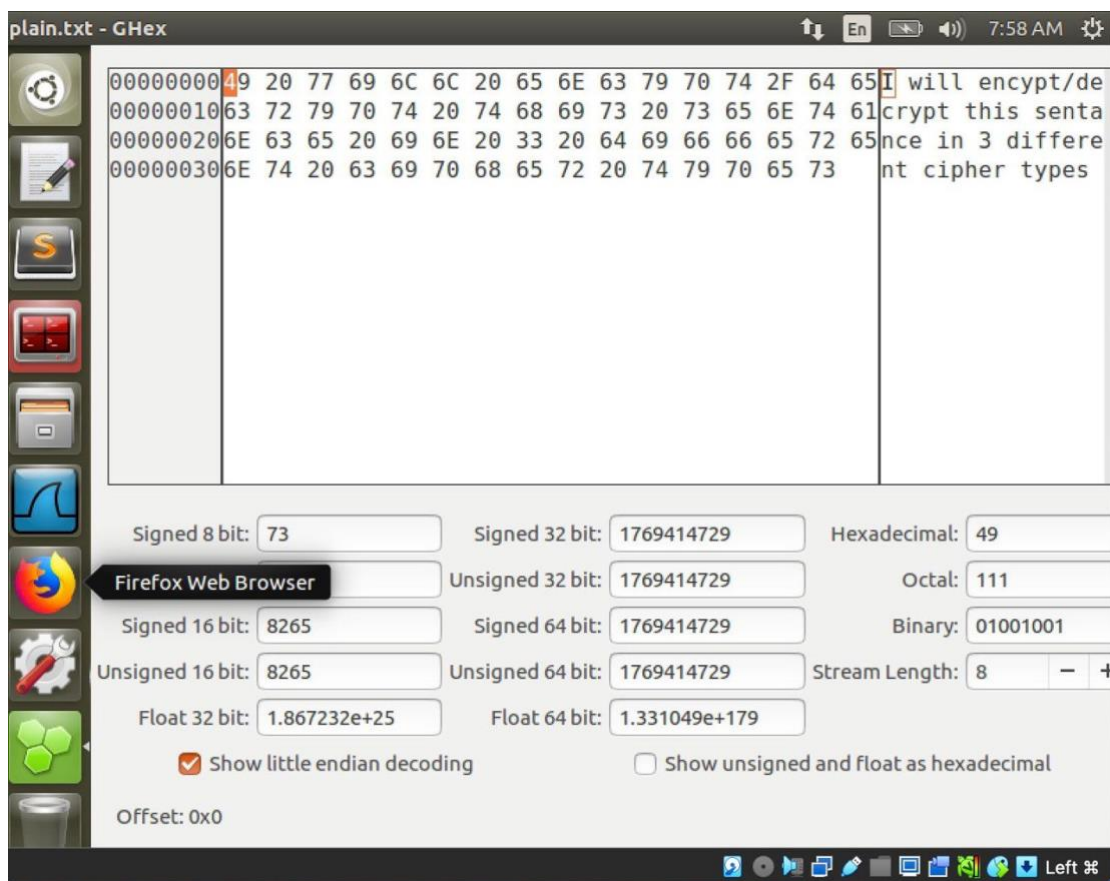
We have written the commands to cipher the sentence where we specified the cipher type (-aes-128-cbc) where -in (input) is the sentence and -out (output) is the sentence ciphered adding the key (-K) and the initial value (-iv).

The second figure shows the hex value of the original sentence before ciphering using (ghex) command.

Figure 1:

```
[10/28/22]seed@VM:~$ ls -l plain.txt
-rw-rw-r-- 1 seed seed 63 Oct 28 07:52 plain.txt
[10/28/22]seed@VM:~$ openssl enc -aes-128-cbc -e -in plain.txt -out cbc_cipher.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

Figure 2:



As seen here in the figure the sentence has been ciphered using the ciphering commands and the length of the ciphered sentence have changed to 64 letters.

The second figure shows the hex value of the ciphered sentence using (ghex) command.

Figure 1:

```
[10/28/22]seed@VM:~$ cat cbc_cipher.txt
|00000000 09v\0'00s00<0000Ha40000 00090!0dv000Z;0]"0DE00 ?[1
0/28/22]seed@VM:~$
```

Figure 2:

The screenshot shows the GHex application window titled 'cbc\_cipher.txt - GHex'. The main pane displays hex data with corresponding ASCII characters. The data is as follows:

Hex Address	Hex Data	ASCII Data
00000000	7C EF F2 00 0E F9 B2 C1 AC 00 20 1C 76 5C EB 27	..... .v\.'
00000010	E9 DF 73 95 DF 3C 12 D9 83 1B 39 48 61 34 30 B8	..s..<....9Ha40.
00000020	80 ED 84 B3 E6 C6 F8 39 C5 21 05 91 64 ED 0F 76	.....9.!...d..v
00000030	C4 1D 89 5A 3B D6 5D 22 8F 11 BD 44 45 19 60 3F	...Z;.]"...DE.`?

Below the main pane, there are various conversion tools and settings:

- Signed 8 bit: 124
- Unsigned 8 bit: 124
- Signed 16 bit: -4228
- Unsigned 16 bit: 61308
- Float 32 bit: 2.231010e-38
- Signed 32 bit: 15921020
- Unsigned 32 bit: 15921020
- Signed 64 bit: 15921020
- Unsigned 64 bit: 15921020
- Float 64 bit: -3.183119e+08
- Hexadecimal: 7C
- Octal: 174
- Binary: 01111100
- Stream Length: 8
- ☒ Show little endian decoding
- ☐ Show unsigned and float as hexadecimal
- Offset: 0x0

Figure 3:

```
[10/28/22]seed@VM:~$ ls -l cbc_cipher.txt
-rw-rw-r-- 1 seed seed 64 Oct 28 07:57 cbc_cipher.txt
[10/28/22]seed@VM:~$
```

To check and verify the ciphering and to make sure that it have been done correctly we have deciphered the sentence (figure 1) and checked on the length, the length of the deciphered sentence (in red) matched the original length of the original sentence (figure 2).

Original 63 letters = Deciphered 63 letters.

Figure 1:

```
[10/28/22]seed@VM:~$ openssl enc -aes-128-cbc -d -in cbc_cipher.txt -out cbc_new_plain.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

Figure 2:

```
[10/28/22]seed@VM:~$ ls -l cbc_new_plain.txt
-rw-rw-r-- 1 seed seed 63 Oct 28 08:03 cbc_new_plain.txt
[10/28/22]seed@VM:~$
```



## 2- bf-cbc (blow-fish cipher):

The first steps has been repeated where the same sentence has been used again. we checked for the length shown in the first figure. This cipher type requires a key (-K) and an initial value (-iv).

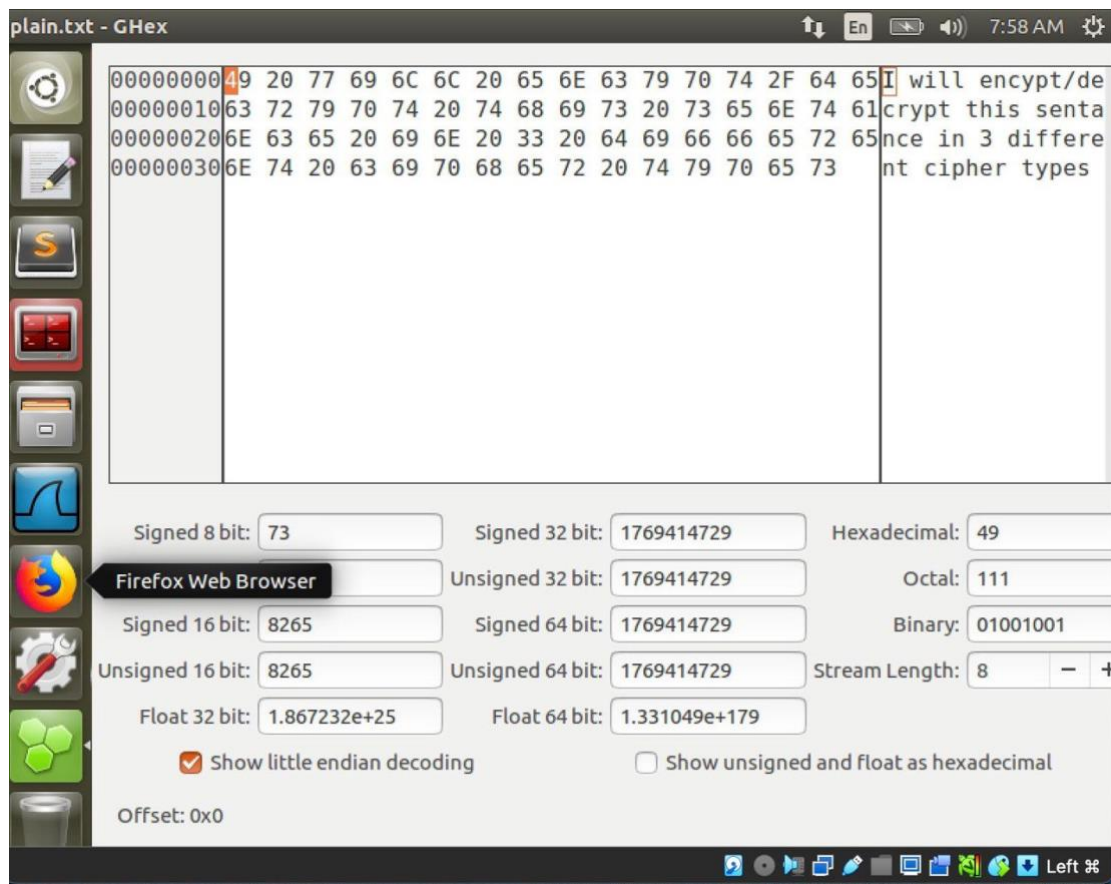
We have written the commands to cipher the sentence where we specified the cipher type (-bf-cbc) and where -in (input) is the sentence and -out (output) is the sentence ciphered adding the key (-K) and the initial value (-iv) (figure 1).

The second figure shows the hex value of the original sentence using (ghex) command (figure 2).

Figure 1:

```
[10/28/22]seed@VM:~$ ls -l plain.txt
-rw-rw-r-- 1 seed seed 63 Oct 28 07:52 plain.txt
[10/28/22]seed@VM:~$ openssl enc -bf-cbc -d -in cbc_cipher.txt -out cbc_new_plain.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

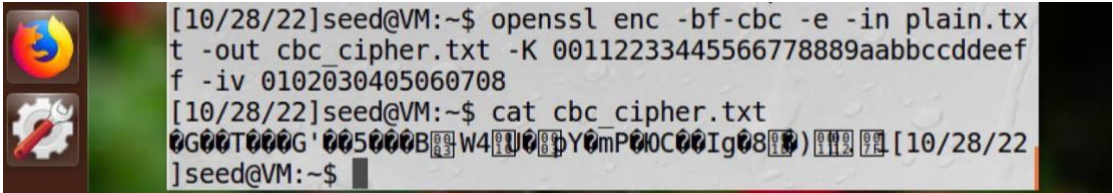
Figure 2:



As seen here in the figure the sentence has been ciphered using the ciphering commands (figure 1) and the length of the ciphered sentence have changed to 64 letters (in red) (figure 3).

The second figure shows the hex value of the ciphered sentence using (ghex) command (figure 2).

Figure 1:



```
[10/28/22]seed@VM:~$ openssl enc -bf-cbc -e -in plain.txt -out cbc_cipher.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
[10/28/22]seed@VM:~$ cat cbc_cipher.txt
0G00T000G'005000B04W40000pY0mP00C00Ig0800)00000[10/28/22
]seed@VM:~$
```

Figure 2:

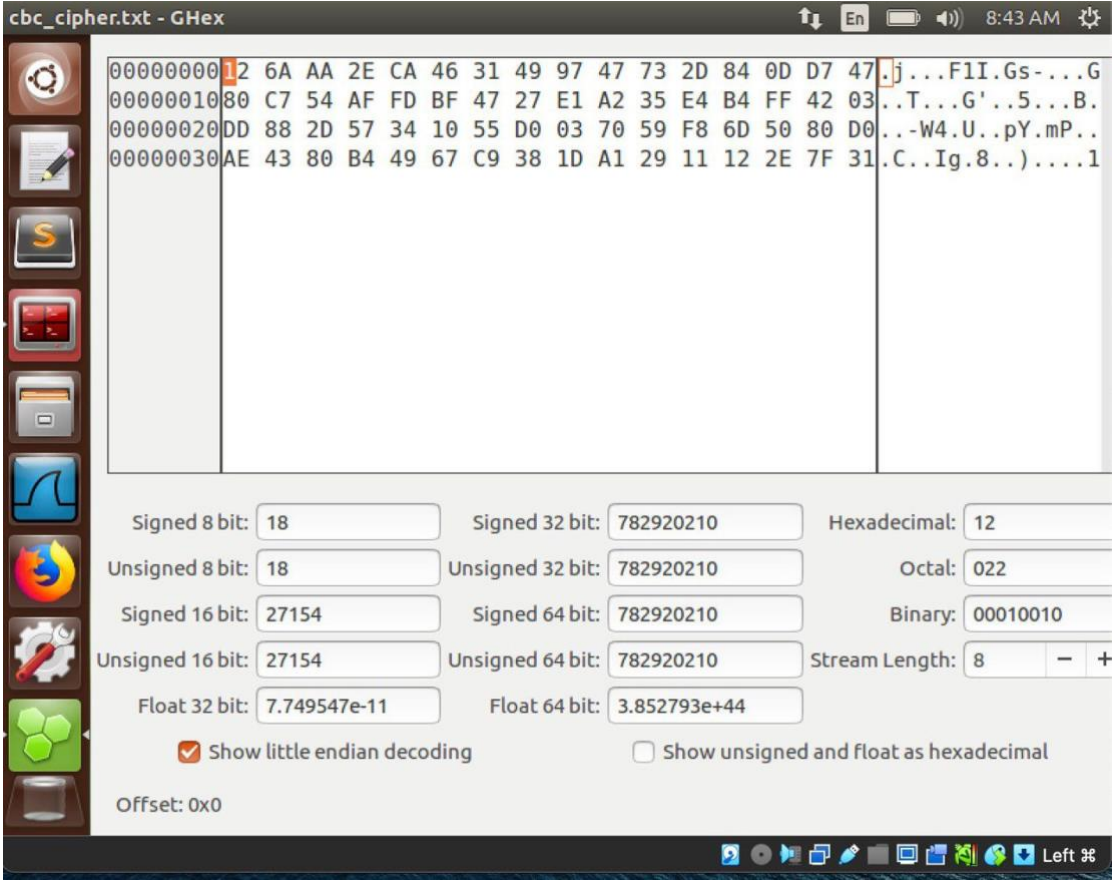
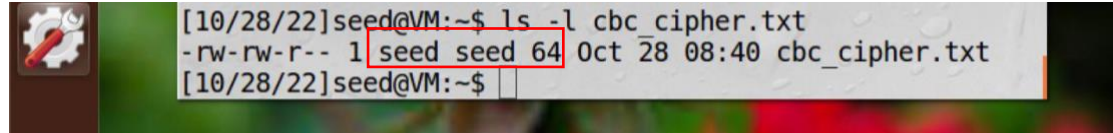


Figure 2 displays the GHex application interface. The main window shows a hex dump of the file 'cbc\_cipher.txt'. The hex data is displayed in a grid, with the first few lines visible. Below the hex dump, there are various conversion options and settings, including Signed 8 bit, Unsigned 8 bit, Signed 16 bit, Unsigned 16 bit, Signed 32 bit, Unsigned 32 bit, Signed 64 bit, Unsigned 64 bit, Float 32 bit, Float 64 bit, Hexadecimal, Octal, Binary, and Stream Length. The 'Show little endian decoding' checkbox is checked, and the 'Offset' is set to 0x0.

Figure 3:



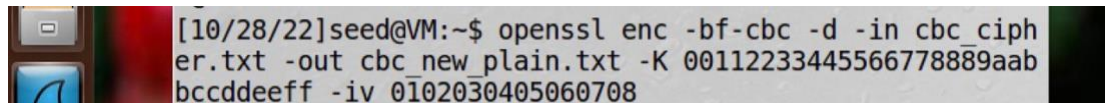
```
[10/28/22]seed@VM:~$ ls -l cbc_cipher.txt
-rw-rw-r-- 1 seed seed 64 Oct 28 08:40 cbc_cipher.txt
[10/28/22]seed@VM:~$
```



To check the ciphering and to make sure that it have been done correctly we have deciphered the sentence (figure 1) and checked on the length, the length of the deciphered sentence (in red) matched the original length of the original sentence (figure 2).

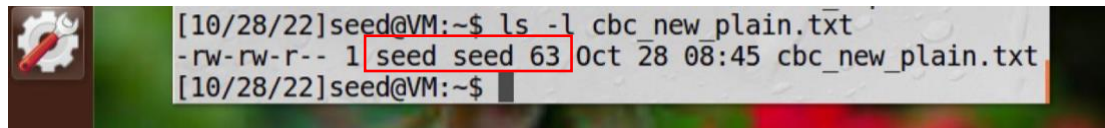
Original 63 letters = Deciphered 63 letters.

Figure 1:

A terminal window with a dark background and light text. The command being entered is: `[10/28/22]seed@VM:~$ openssl enc -bf-cbc -d -in cbc_cipher.txt -out cbc_new_plain.txt -K 00112233445566778889aab bccddeeff -iv 0102030405060708`

```
[10/28/22]seed@VM:~$ openssl enc -bf-cbc -d -in cbc_cipher.txt -out cbc_new_plain.txt -K 00112233445566778889aab bccddeeff -iv 0102030405060708
```

Figure 2:

A terminal window showing the output of the `ls -l` command. The output line is: `-rw-rw-r-- 1 seed seed 63 Oct 28 08:45 cbc_new_plain.txt`. The text "seed seed 63" is highlighted with a red rectangular box.

```
[10/28/22]seed@VM:~$ ls -l cbc_new_plain.txt
-rw-rw-r-- 1 seed seed 63 Oct 28 08:45 cbc_new_plain.txt
[10/28/22]seed@VM:~$
```

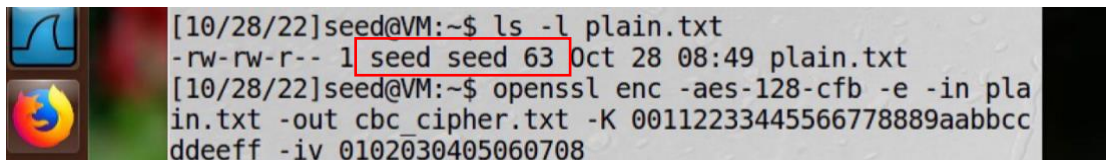
### 3- aes-128-cfb:

The first steps has been repeated where the same sentence has been used again. we checked for the length shown in the first figure. This cipher type requires a key (-K) and an initial value (-iv).

We have written the commands to cipher the sentence where we specified the cipher type (-aes-128-cfb) and where -in (input) is the sentence and -out (output) is the sentence ciphered adding the key (-K) and the initial value (-iv) (figure 1).

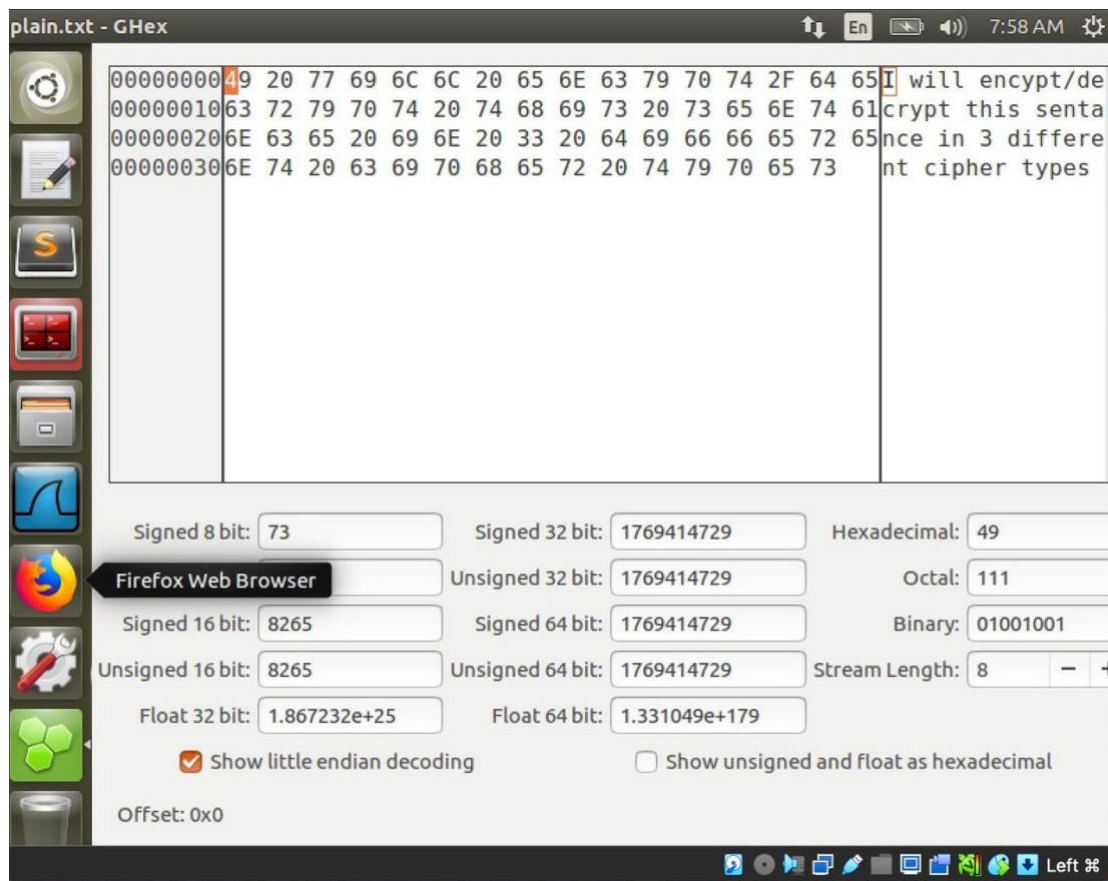
The second figure shows the hex value of the original sentence using (ghex) command (figure 2).

Figure 1:



```
[10/28/22]seed@VM:~$ ls -l plain.txt
-rw-rw-r-- 1 seed seed 63 Oct 28 08:49 plain.txt
[10/28/22]seed@VM:~$ openssl enc -aes-128-cfb -e -in plain.txt -out cbc_cipher.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

Figure 2:



The second figure shows the hex value of the ciphered sentence using (ghex) command (figure 2).

```
[10/28/22]seed@VM:~$ openssl enc -aes-128-cfb -e -in plain.txt -out cbc_cipher.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708  
[10/28/22]seed@VM:~$ cat cbc_cipher.txt  
  L S      m y    @?A   t     H{  E4Q n     B7lg! V  [10
```

The screenshot shows a hex editor window with a menu bar (File, Edit, View, Windows, Help) and a toolbar. The main area displays four rows of hexadecimal data with their corresponding ASCII characters. The first row is highlighted in blue.

Hex Address	Hex Data	ASCII
00000000	CE A6 F8 4C AD 53 9A B4 C1 EC C6 6D 01 3F 38 08	...L.S....m.?8.
00000010	79 E2 E5 9B 96 40 3F 41 94 01 CE 74 90 C8 E4 5F	y....@?A...t...
00000020	E0 48 7B 8E B0 DA FF 45 34 51 1D 6E 1E E9 E9 F8	.H{....E4Q.n....
00000030	07 8F A9 DF C2 90 38 37 6C 67 21 86 56 06 C5	.....87lg!.V..

Below the hex data, there are various conversion options and fields:

- Signed 8 bit: -50
- Signed 32 bit: 1291364046
- Hexadecimal: CE
- Unsigned 8 bit: 206
- Unsigned 32 bit: 1291364046
- Octal: 316
- Signed 16 bit: -22834
- Signed 64 bit: 1291364046
- Binary: 11001110
- Unsigned 16 bit: 42702
- Unsigned 64 bit: 1291364046
- Stream Length: 8
- Float 32 bit: 1.303650e+08
- Float 64 bit: -2.684231e-55

At the bottom, there are two checkboxes: "Show little endian decoding" (checked) and "Show unsigned and float as hexadecimal" (unchecked). The "Offset: 0x0" is also displayed.

```
[10/28/22] seed@VM:~$ ls -l cbc_cipher.txt
-rw-rw-r-- 1 seed seed 63 Oct 28 08:52 cbc_cipher.txt
```

To check the ciphering and to make sure that it have been done correctly we have deciphered the sentence (figure 1) and checked on the length, the length of the deciphered sentence (in red) matched the original length of the original sentence (figure 2).

Original 63 letters = Deciphered 63 letters.

Figure 1:

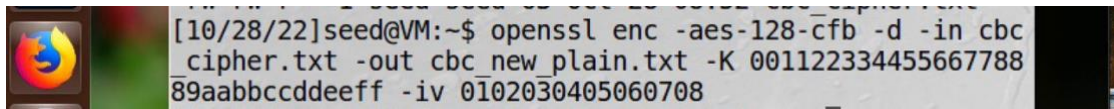
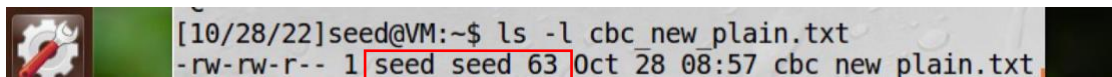


Figure 2:



### Task 3: Encryptions Mode -ECB VS. CBC:

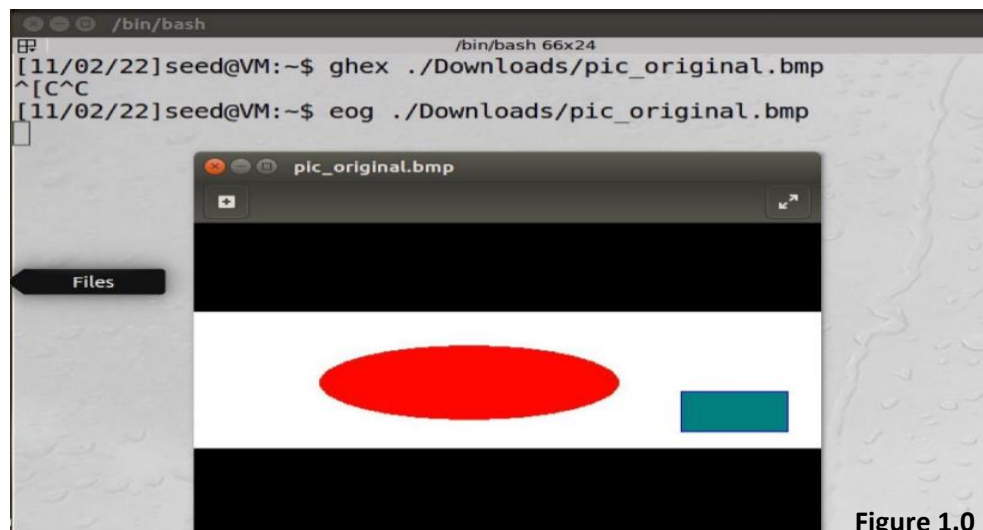
In this task we have an image that we want to encrypt using ECB (Electronic Code Book) and CBC (Cipher Blocks Chaining) modes. On each mode, we have to follow certain steps. the first one is encrypting the header information c is consists of the first 54 bytes of the image. Then we encrypt the body which consist of +55 bytes.

Following the step to combine them together.

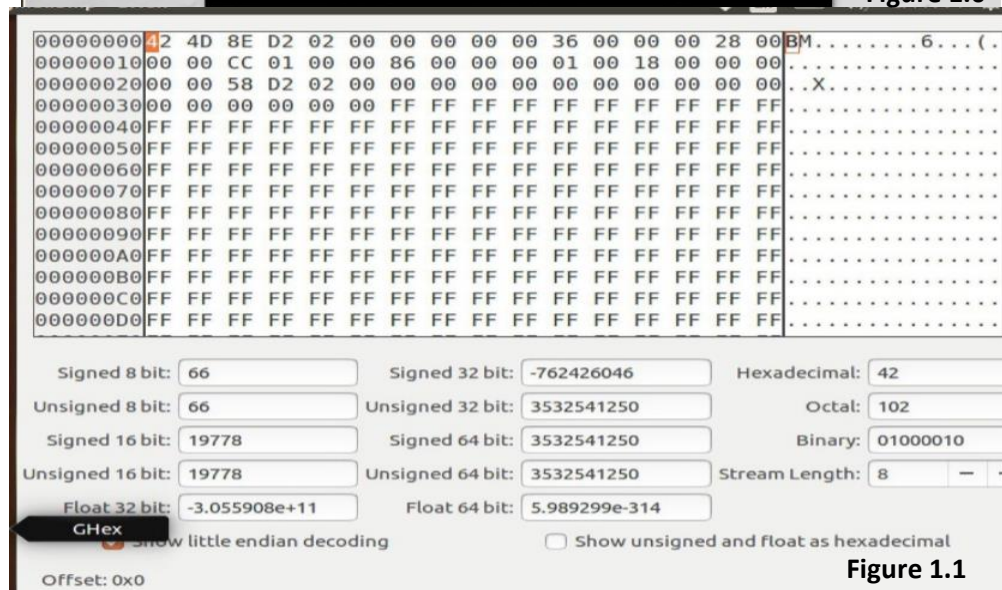
Before encrypting the image in a bmp form in two different modes. Each of these two modes will be described and explained in detail. The two modes as mentioned above are:

1. ECB (Electronic Code Book).
2. CBC (Cipher Blocks Chaining).

In figure (1.0) is the image in bmp form before encrypting it, used ego command to view the image. As well, in figure (1.1) is the hex for the original image before encrypting it using the (ghex) command.



### Figure 1.0



### Figure 1.1



## 1. CBC - Cipher Block Chaining

For CBC, step one we encrypted the image as shown in figure (2.1) using the specified cipher type aes-128-cbc. Where we entered the commands to cipher only the header which consists of the first 54 bytes. Where (-in) is the original image in bmp form is the input, and out is the output of the image after encrypting the header only. Which also needs the key (-k) and the initial value (-iv).

In figure (2.0) where we entered the command to encrypt the header.

```

/bin/bash
[11/02/22]seed@VM:~$ ghex ./Downloads/pic_original.bmp
^[C^C
[11/02/22]seed@VM:~$ eog ./Downloads/pic_original.bmp
^C
[11/02/22]seed@VM:~$ openssl enc -aes-128-cbc -in ./Downloads/pic_
original.bmp -out cbc_pic_cipher.bmp -K 00112233445566778889aabbcc
ddeeff -iv 0102030405060708
[11/02/22]seed@VM:~$ eog cbc_pic_cipher.bmp

(eog:2702): EOG-WARNING **: Failed to open file '/home/seed/.cache
/thumbnails/normal/ca53fb69552c4149992ed147eel6c66d.png': No such
file or directory

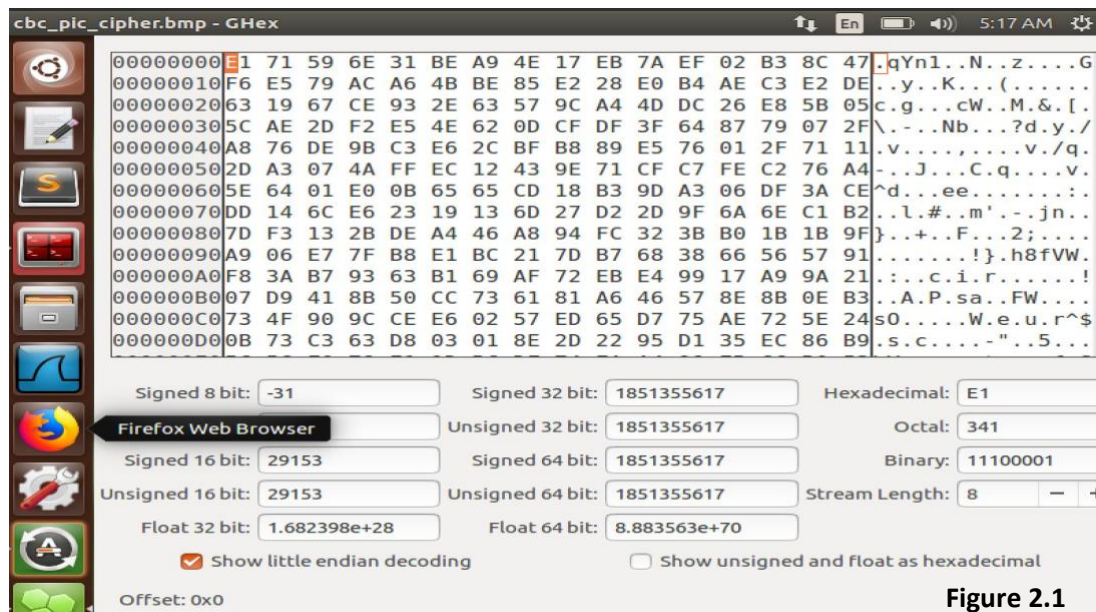
(eog:2702): EOG-WARNING **: Failed to open file '/home/seed/.cache
/thumbnails/normal/a6a04b27dae0995df46631b39cfc4326.png': No such
file or directory

```

Figure 2.0

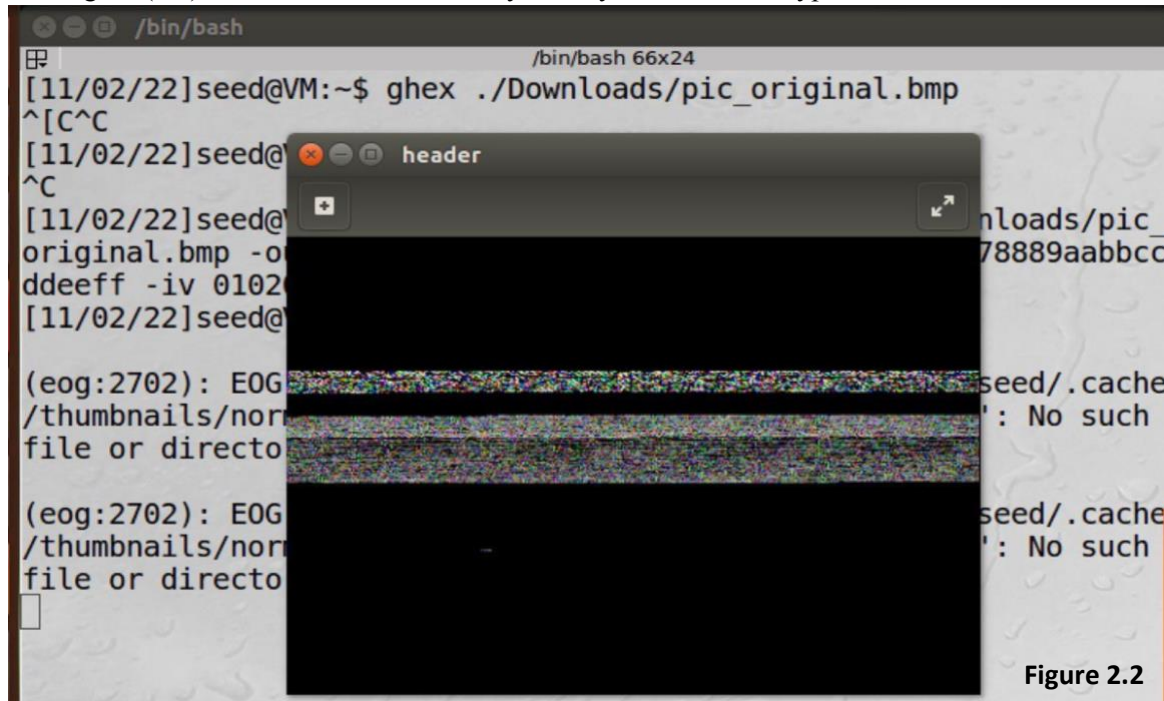
### Figure 2.0

(ghex) command used to output the hexadecimal after the whole image is ciphered. shown in the following figure (2.1)



### Figure 2.1

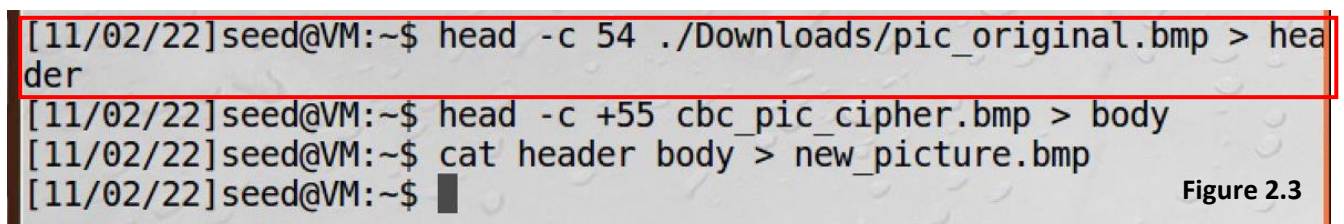
in figure (2.2) is after the header of 54 bytes only have been encrypted.



Step two, we redefined the encrypted header of 54 bytes as the original image. in order for it to encrypt the body of +55 bytes of the original image. which the first 54 bytes of the original image is already encrypted. In this case, after encrypting the body of +55 bytes it would result as the whole image being ciphered.

We used the following command to redefine the header as though it's the original image. As shown in figure (2.3) in red.

- Head -c 54 ./Downloads/ pic\_original.bmp > header



Continuing on step 2 is where we encrypt the body of +55 bytes. As shown in figure (2.4) for the whole image being ciphered. And that means we have reached the goal that we wanted to achieve for CBC mode.

In order to view the image, use eog image viewer.

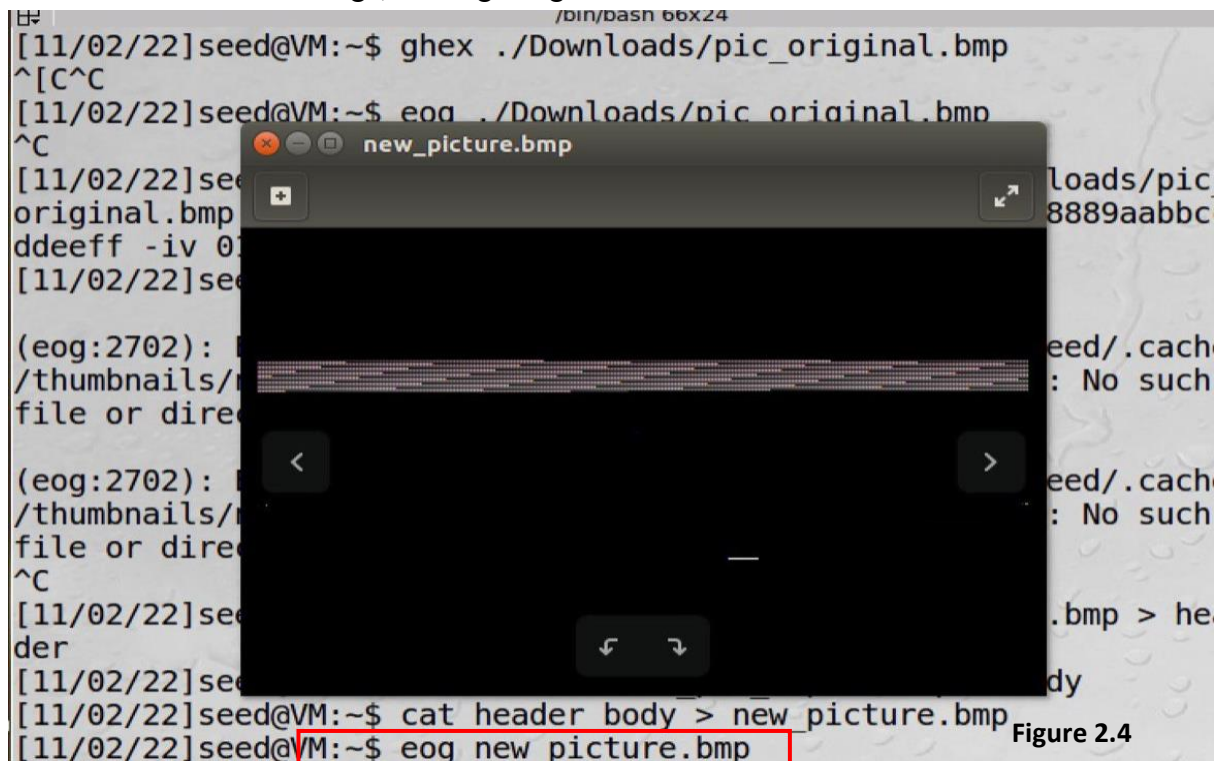


Figure 2.4

In following figure (2.5) after the ghex command. If you noticed that the header of the first 54 bytes is aligned with the header of the original image after encryption and defined it as new\_picture.

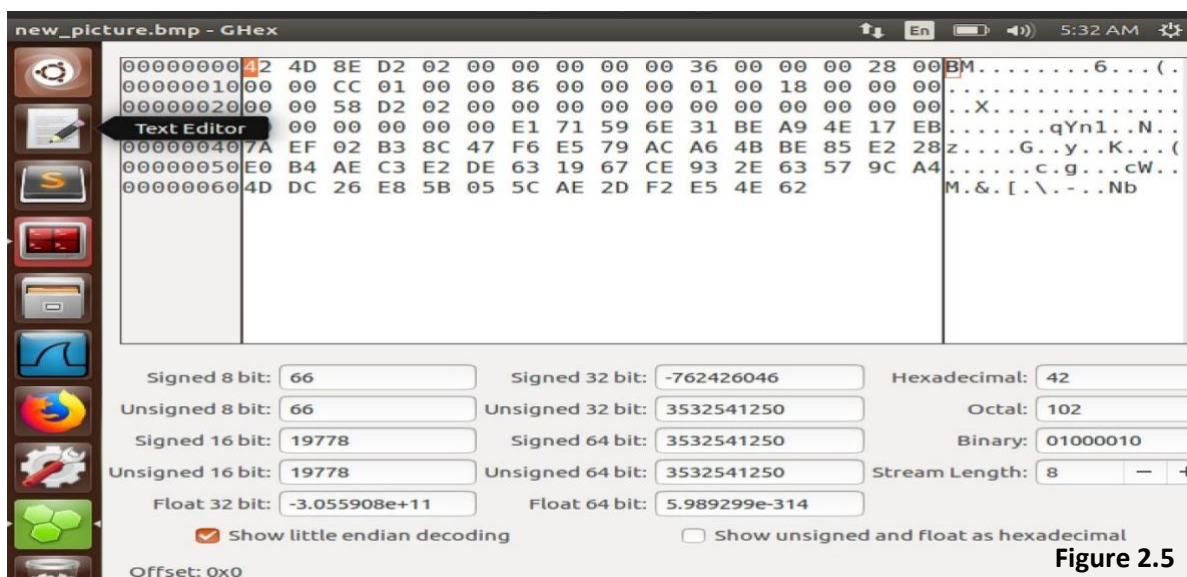


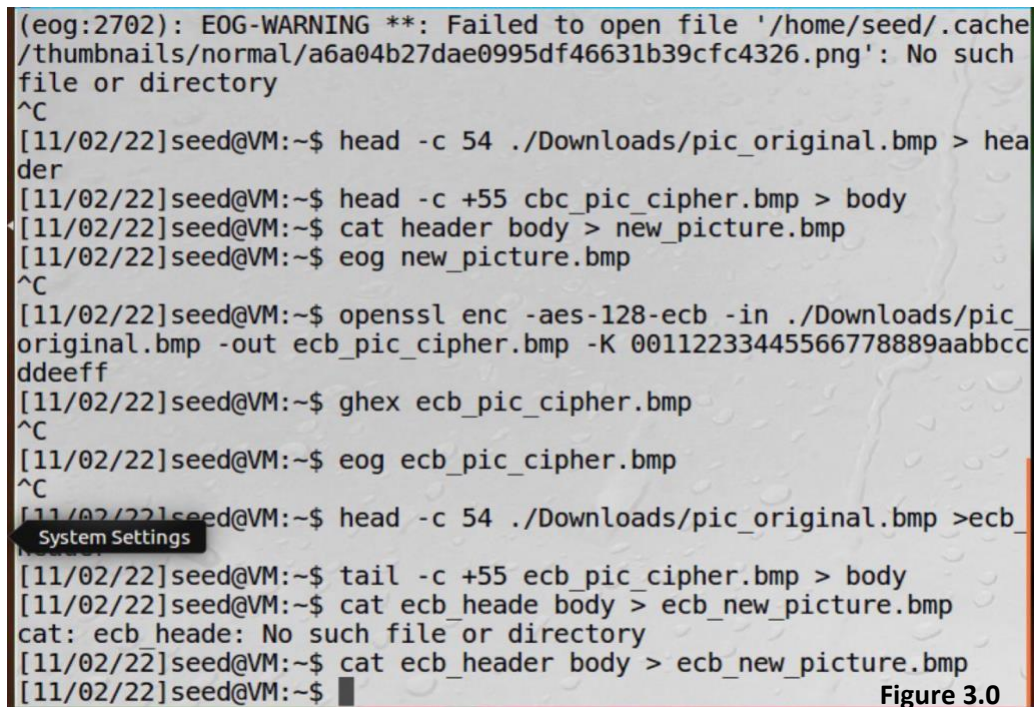
Figure 2.5



## 2. ECB - Electronic Code Book

For ECB, we repeat the same process with a different cipher type. Step one we encrypted the image as shown in figure (3.0) using the specified cipher type aes-128-ecb. Then we entered the commands to encrypt only the header which consists of the first 54 bytes. Where (-in) is the input consists of the original image in a bmp form, and (-out) is the output of the image after encrypting the header only. Which requires only the key (-k).

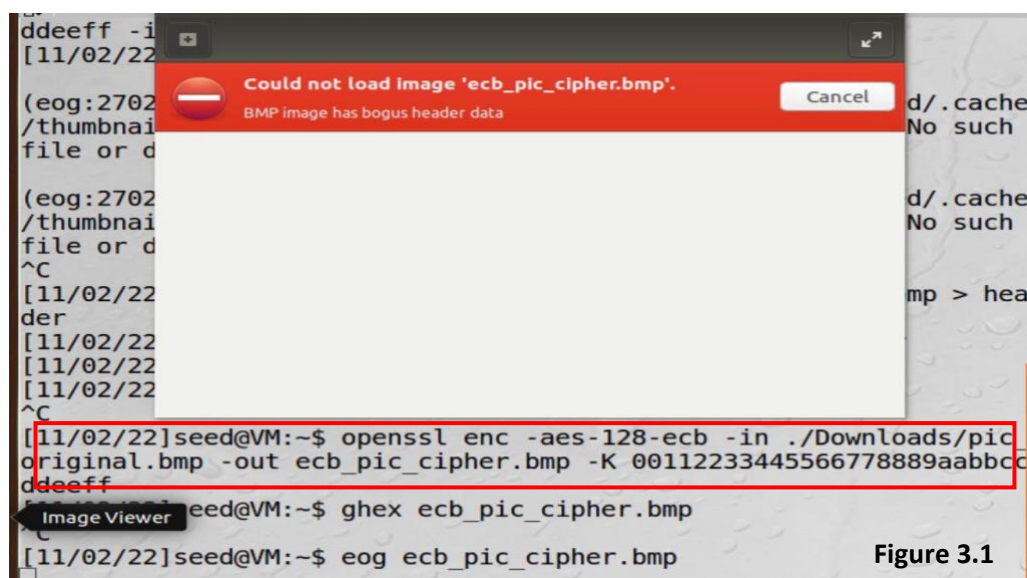
In figure (3.0) where we entered the command to encrypt the header. In figure (3.1) is after the header of the first 54 bytes have been encrypted.



```
(eog:2702): EOG-WARNING **: Failed to open file '/home/seed/.cache/thumbnails/normal/a6a04b27dae0995df46631b39cfc4326.png': No such file or directory
^C
[11/02/22]seed@VM:~$ head -c 54 ./Downloads/pic_original.bmp > header
[11/02/22]seed@VM:~$ head -c +55 ecb_pic_cipher.bmp > body
[11/02/22]seed@VM:~$ cat header body > new_picture.bmp
[11/02/22]seed@VM:~$ eog new_picture.bmp
^C
[11/02/22]seed@VM:~$ openssl enc -aes-128-ecb -in ./Downloads/pic_original.bmp -out ecb_pic_cipher.bmp -K 00112233445566778889aabbccddeeff
[11/02/22]seed@VM:~$ ghex ecb_pic_cipher.bmp
^C
[11/02/22]seed@VM:~$ eog ecb_pic_cipher.bmp
^C
[11/02/22]seed@VM:~$ head -c 54 ./Downloads/pic_original.bmp > ecb_header
[11/02/22]seed@VM:~$ tail -c +55 ecb_pic_cipher.bmp > body
[11/02/22]seed@VM:~$ cat ecb_header body > ecb_new_picture.bmp
cat: ecb_header: No such file or directory
[11/02/22]seed@VM:~$ cat ecb_header body > ecb_new_picture.bmp
[11/02/22]seed@VM:~$
```

Figure 3.0

After we wrote the following command shown in figure (3.1) in red box, to encrypt it resulted differently, that it is not able to load the image shown in figure (3.1).



```
ddeeff -i
[11/02/22]seed@VM:~$ openssl enc -aes-128-ecb -in ./Downloads/pic_original.bmp -out ecb_pic_cipher.bmp -K 00112233445566778889aabbccddeeff
(eog:2702): EOG-WARNING **: Failed to open file '/home/seed/.cache/thumbnails/normal/a6a04b27dae0995df46631b39cfc4326.png': No such file or directory
^C
[11/02/22]seed@VM:~$ ghex ecb_pic_cipher.bmp
[11/02/22]seed@VM:~$ eog ecb_pic_cipher.bmp
```

Could not load image 'ecb\_pic\_cipher.bmp'.  
BMP image has bogus header data

Figure 3.1

On step 2, we encrypt the body of +55 bytes. As shown in figure (3.2) for the whole image being ciphered. And that means we have reached the goal that we wanted to achieve for CBC mode. We used the eog to view the image.

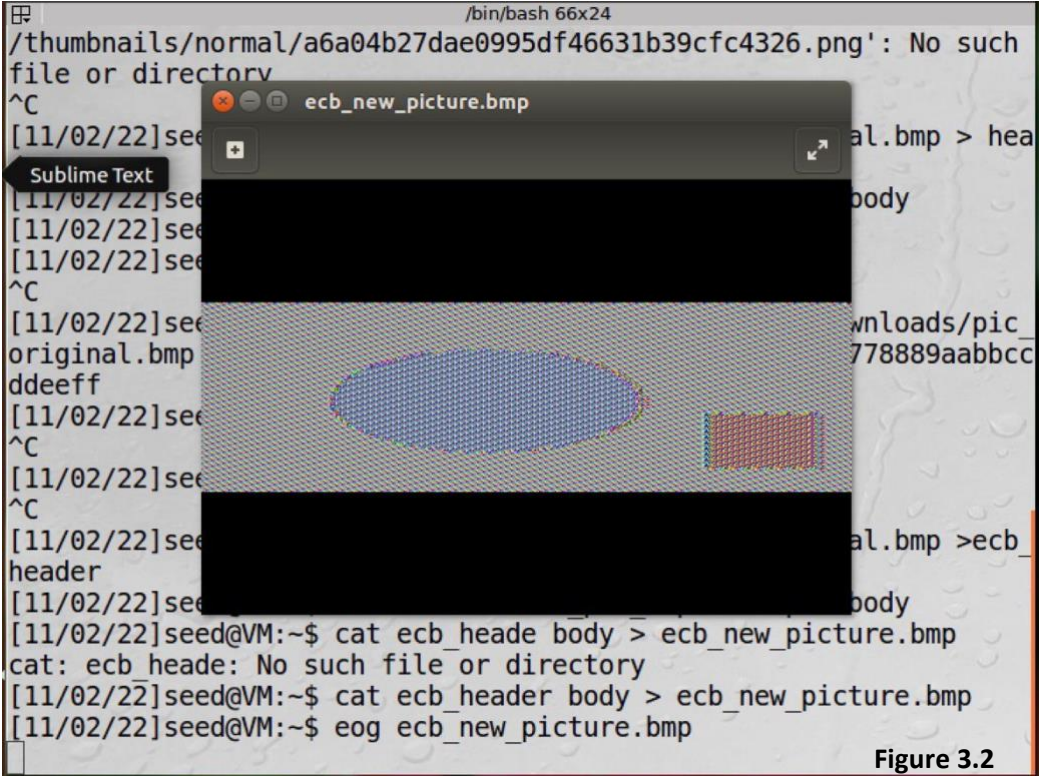


Figure 3.2

(ghex) command used to output the hexadecimal after the whole image is ciphered. shown in the following figure (3.3)

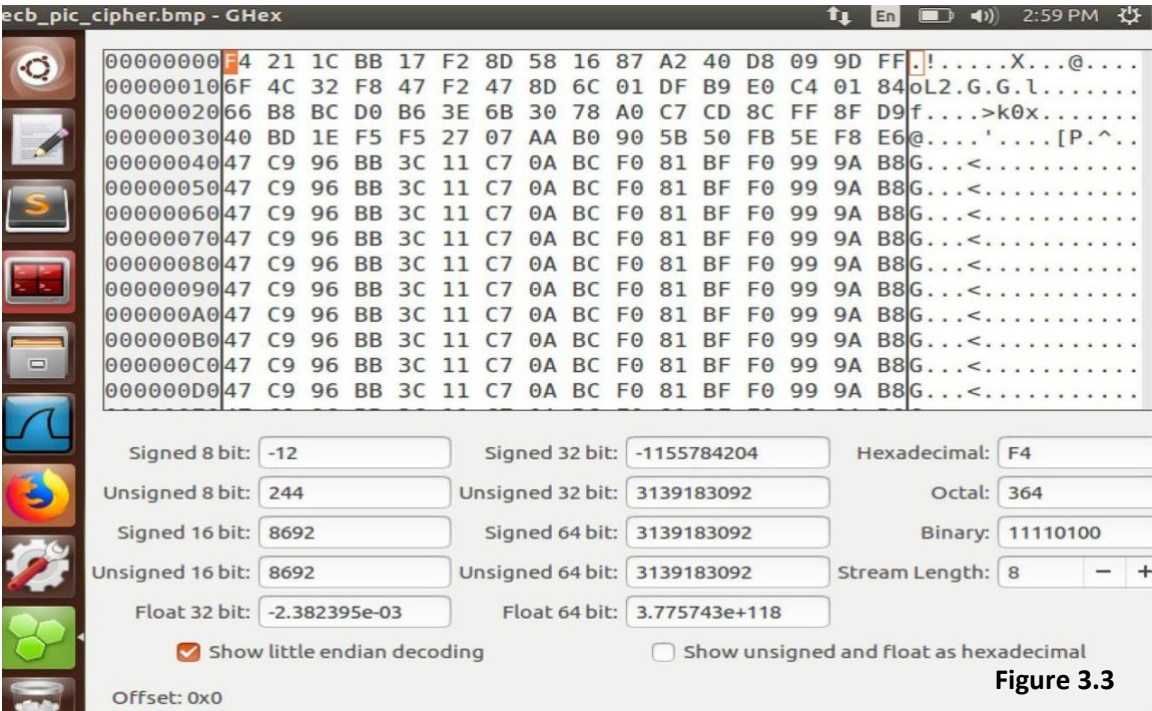


Figure 3.3



CBC stands for Cipher Block Chaining while ECB stands for Electronic Code Book as mentioned before. For more understanding, ECB is suitable for encrypting small messages, while CBC is used to encrypt large messages. As you see below ECB and CBC are a bit different from each other. CBC after encrypting it, you will not notice what the image is. While, ECB the image would be slightly noticeable, and could see slight detail of the image and its lines after ciphering it. additionally, CBC requires the key and initial value other than ECB which only requires the key.

```

[11/02/22]seed@VM:~$ ghex ./Downloads/pic_original.bmp
^C
[11/02/22]seed@VM:~$ eog ./Downloads/pic_original.bmp
^C
[11/02/22]seed@VM:~$ cat ecb_header body > ecb_new_picture.bmp
[11/02/22]seed@VM:~$ cat ecb_header body > ecb_new_picture.bmp
[11/02/22]seed@VM:~$ eog ecb_new_picture.bmp

```

**ECB**

```

[11/02/22]seed@VM:~$ ghex ./Downloads/pic_original.bmp
^C
[11/02/22]seed@VM:~$ eog ./Downloads/pic_original.bmp
^C
[11/02/22]seed@VM:~$ cat ecb_header body > ecb_new_picture.bmp
[11/02/22]seed@VM:~$ cat ecb_header body > ecb_new_picture.bmp
[11/02/22]seed@VM:~$ eog ecb_new_picture.bmp

```

**CBC**

## Task 7: Programming using the Crypto Library:

The goal of this task is to build a program that is able to find the encryption key, as a .txt file has been provided for us to use that consists of different words as well as the key and the initial value (iv) that was available in the PDF. The program will read each word from the .txt file and guess which word is the correct key to the given message.

### The Code:

#### 1- Imports:

For starters we have imported all the necessary tools that we will be using to build the program, The imports that we will use to read from file are: the io.File and util.Scanner, As for the ones that we will need for the Encryption, Decryption are the Crypto tools for ciphering, iv and the key.

#### 2- HexToByte:

This method converts the hexadecimal values into byte values by accepting the hexadecimal value as String. Defined an array value byte[] data which will create a new byte array by calculating the length of the hexadecimal string / 2, After that we need to move along the length of the hexadecimal String to convert each hex value (Using the for loop). The loop will start implementing the formula of converting the hexadecimal value to a byte value Finally returning the byte value (return data).

#### The formula:

Figure 1:

```
data[i/2] = (byte) ((Character.digit(hex.charAt(i), 16) << 4) + Character.digit(hex.charAt(i+1), 16));
```

\*This formula is fixed to converting hexadecimal values to byte.

Figure 2:

```
1
2
3 // All the imports that we need for this project:
4 import java.io.File;
5 import java.util.Scanner;
6 import javax.crypto.Cipher;
7 import javax.crypto.spec.IvParameterSpec;
8 import javax.crypto.spec.SecretKeySpec;
9
10
11 public class Task7 {
12
13     public static byte[] HexToByte(String hex){ // Turn Hex to Bytes value.
14
15         byte[] data = new byte[hex.length()/2];
16         for (int i = 0; i < hex.length(); i += 2){
17             data[i/2] = (byte) ((Character.digit(hex.charAt(i), 16) << 4) + Character.digit(hex.charAt(i+1), 16));
18         }
19         return data;
20     }
21 }
22
```

### 3- BytesToHex:

This method converts the byte values to hex values by accepting the byte value as an array. Defined a char value char[] HexChars which will create a new char array by calculating the bytes length \* 2, Another array has been defined char[] HexArrayValues where it will take the hexadecimal values (0-9 and A-F) and turn them to a charArray, After that we need to move along the length of the bytes to convert each byte value (Using the for loop).

The loop will start implementing the formula of converting the Byte values to a hexadecimal value Finally returning the String of the Hexadecimal value (return new String(HexChars);).

#### The formula:

Figure 3:

```
int x = bytes[i] & 0xFF;

HexChars[i*2] = HexArrayValue[x >>> 4];

HexChars[i*2+1] = HexArrayValue[x & 0x0F];
```

\*This formula is fixed to converting byte values to hexadecimal.

Figure 4:

```
22
23
24 // Turning Byte to hex values:
25 public static String BytesToHex(byte[] bytes){
26
27     char[] HexChars = new char[bytes.length * 2];
28     char[] HexArrayValue = "0123456789ABCDEF".toCharArray();
29
30     for (int i = 0; i < bytes.length; i++){
31         int x = bytes[i] & 0xFF;
32         HexChars[i*2] = HexArrayValue[x >>> 4];
33         HexChars[i*2+1] = HexArrayValue[x & 0x0F];
34     }
35     return new String(HexChars); // return new hex char.
36 }
37
```

#### 4- encrypt:

In this method we have started the encryption process where this method will accept the plaintext, initial value and the key. The process begins by a try{} statement that will allow us to define the values that we inserted and test for any errors. We will be using the crypto tool here as we will be needing the IvParameterSpec, SecretKeySpec and Cipher.

- **IvParameterSpec:** specifies the initialization vector, where it will take the value of the initiation vector that have been given (invex).
- **SecretKeySpec:** constructs the given key (K) that will be represented as bytes (K.getBytes()) using the aes algorithm ("AES").
- **Cipher:** providing the functionality of cryptographic cipher used in encryption and decryption, It will get the Instance (.getInstance()) of the default values ("AES/CBC/PKCS5Padding").

After that the cipher will initialize (.init()) the value by taking the Cipher and using the Encrypt\_Mode tool (Cipher.ENCRYPT\_MODE) , the value of the SecretKeySpec(skySpec) and the IvParameterSpec (iv).

Finally returning the cipher text value (return cipherText). Catch exception to end the try{}, returning null if nothing (return null).

figure 5:

```
38
39 //The Encryption Method:
40 public static byte[] encrypt(String PT, byte[] invex, String K){ // PT: plaintext, invex: initial vector, K: key.
41
42     try{
43
44         IvParameterSpec iv = new IvParameterSpec(invex);
45         SecretKeySpec skeySpec = new SecretKeySpec(K.getBytes(), "AES");
46         Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
47         cipher.init(Cipher.ENCRYPT_MODE, skeySpec, iv);
48         byte[] cipherText = cipher.doFinal(PT.getBytes());
49
50         return cipherText;
51     } catch (Exception exp){
52
53     }
54     return null;
55 }
56
```

## 5- decrypt:

In this method we have started the decryption process where this method will accept the encrypted message, initial value and the key. The process begins by a try{} statement that will allow us to define the values that we inserted and test for any errors. We will be using the crypto tool here as we will be needing the IvParameterSpec, SecretKeySpec and Cipher.

- **IvParameterSpec:** specifies the initialization vector, where it will take the value of the initiation vector that have been given (invex).
- **SecretKeySpec:** constructs the given key (K) that will be represented as bytes (K.getBytes()) using the aes algorithm ("AES").
- **Cipher:** providing the functionality of cryptographic cipher used in encryption and decryption, It will get the Instance (.getInstance()) of the default values ("AES/CBC/PKCS5Padding").

After that the cipher will initialize (.init()) the value by taking the Cipher and using the Decrypt\_Mode tool (Cipher.DECRYPT\_MODE) , the value of the SecretKeySpec(skySpec) and the IvParameterSpec (iv).

A byte array is defined as it will contain the original text (byte[] original) as it will take the cipher value and finishes the decryption (.doFinal()) of the encrypted text that is given (encrypt).

Finally returning the cipher text value (return new String(original);). Catch exception to end the try {}, returning null if nothing (return null).

Figure 6:

```
57
58 //The Decryption Method:
59 public static String decrypt(byte[] encrypt, byte[] invex, String K){ // encrypt: encrypted message, invex: initial value, K: key.
60
61     try{
62
63         IvParameterSpec iv = new IvParameterSpec(invex);
64         SecretKeySpec skeySpec = new SecretKeySpec(K.getBytes(), "AES");
65         Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
66         cipher.init(Cipher.DECRYPT_MODE, skeySpec, iv);
67         byte[] original = cipher.doFinal(encrypt);
68
69         return new String(original);
70     } catch (Exception exp){
71
72     }
73     return null;
74 }
75
76
```



## 6- GuessTheKey:

In this method is where everything comes together, this method will call all the previous methods as well as start the final calculations for this project. The method starts by the try {} statement that will allow us to define and test for any errors. After that we used the File and gave it the .txt directory and Scanner to scan the directory to be able to read the.txt file that was given.

A while loop have been created to read through the file scanning each line and storing each vale as a String key (K). where the key will be replace the target line (\n) with a white space (" ").

If the key length was bigger than 16 then replace what is left of it as # shown bellow:

Figure 7:

```
if (K.length() < 16){  
  
    for (int i = K.length(); i < 16; i++) {  
  
        K += "#"; // Replace the rest of the key that is < 16 with hashes.  
  
    }  
}
```

After that we will begin with the inputs, specifying our plaintext "This is a top secret." As String, The initial value "aabbccddeeff00112233445566778899" converting from hexadecimal to bytes (Using HexToByte) and the encrypting it (using encrypt) taking the plaintext, initial value (iv), and key (K).

Figure 8:

```

76
77 //Guess The Key Method:
78 static public void GuessTheKey() {
79
80     try{
81
82         File file = new File(pathname: "/Users/XI/Desktop/words.txt"); // using file IO.
83         Scanner scan = new Scanner(file); // Scanner to scan document.
84
85         while (scan.hasNextLine()){ // Loop to read from file.
86             String K = scan.nextLine(); //K: key.
87             K = K.replace(target: "\n", replacement: "");
88
89             if (K.length() < 16){
90                 for (int i = K.length(); i < 16; i++) {
91                     K += "#"; // Replace the rest of the key that is < 16 with hashes.
92                 }
93             }
94
95             //The inputs for GuessTheKey:
96             String plaintext = "This is a top secret.");//PlainText
97             byte[] iv = HexToByte(hex: "aabbccddeeff00998877665544332211"); //IV
98             byte[] encryptText = encrypt(plaintext, iv, K);
99

```

Continuing we will create an if statement in the event of an encrypted text available and not null (if (encryptText != null)) We will convert the encrypted text from bytes to a hexadecimal value, if the encrypted text turned to lowercase (.toLowerCase()) is equal to “764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2” and a message will be displayed for the user letting them know the search is now done.

The encrypted message will be decrypted (decrypt()) defined as plain, where it will take the encrypted message value (encryptText), initial value (iv) and the key (K). if the decrypted text is available and not null (if (plain != null)) the program will start printing the results displaying the plaintext, Key used in ASCII, Key used in HEX and the encryption text in hex, else the key has not been found.

Finally Catch exception to end the try{ }.

Figure 9:

```

100         if (encryptText != null){
101             //Turn Bytes to Hex:
102             String encryptedHex = BytesToHex(encryptText);
103             if (encryptedHex.toLowerCase().equals(anObject: "764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2")){
104                 System.err.println(x: "Key Search Is Now Done! The Results >>>");
105
106                 String plain = decrypt(encryptText, iv, K); //Decryption of the text:
107                 if (plain != null){ //Print the final results.
108                     System.out.println("Plain Test is: " + plain);
109                     System.out.println("Key Used in ASCII: " + K);
110                     String keyHex = BytesToHex(K.getBytes()); //Key in hex
111                     System.out.println("Key Used in Hex: " + keyHex);
112                     System.out.println("Encrypted Text in Hex: " + encryptedHex);
113                 }else{
114                     System.err.println(x: "No Key Found"); // Error message if key not found.
115                 }
116             }
117         }
118     }
119 }
120
121 }
122
123 }
124 catch (Exception exp){
125     exp.printStackTrace();
126 }
127 }
128

```

## 7- Main:

Reaching the end of our program where the main method will call the GuessTheKey method as all the calculations have been done there.

Figure 10:

```

128
129
130     Run | Debug
131     public static void main(String[] args){
132         GuessTheKey();
133     }
134

```

## The Final Output:

Figure 11:

```
Key Search Is Now Done! The Results >>>
Plain Text is: This is a top secret.
Key Used in ASCII: Syracuse#####
Key Used in Hex: 537972616375736523232323232323
Encrypted Text in Hex: 764AA26B55A4DA654DF6B19E4BCE00F4ED05E09346FB0E762583CB7DA2AC93A2
```