**Prerequisite**

**1. Install OpenSSL and Development Libraries**

Run the following command in your VM to install OpenSSL and its development libraries:

*sudo apt-get update*

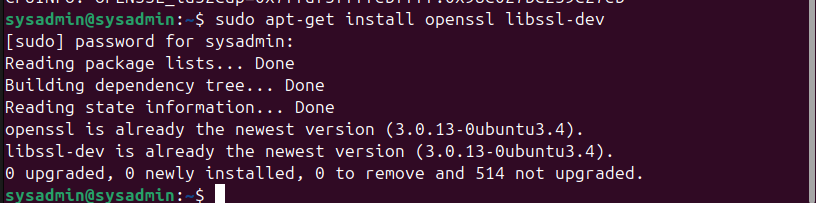
*sudo apt-get install openssl libssl-dev*

2. Verify Installation

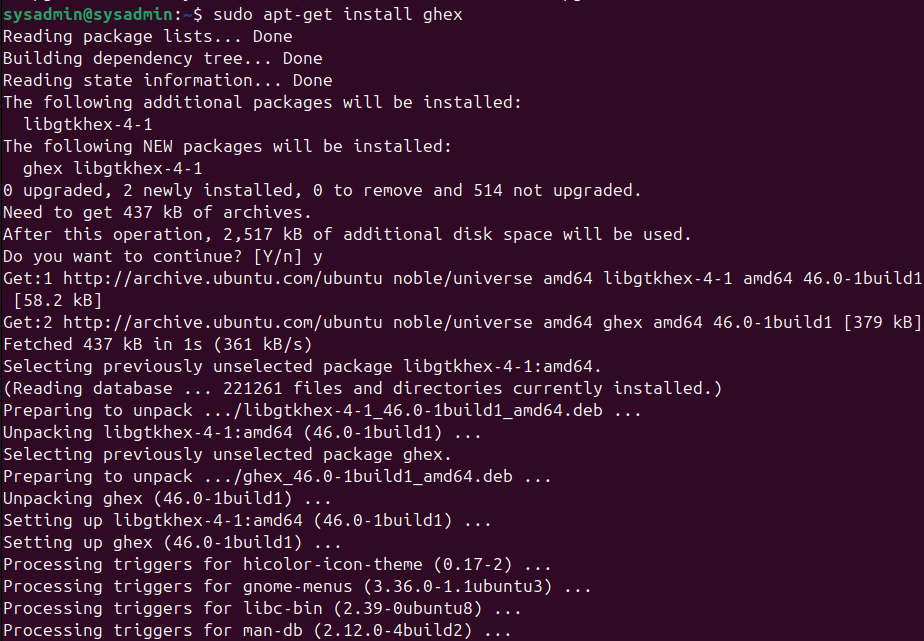
After installation, check the version to confirm it’s installed correctly:

*openssl version -a*

****

****

**For GHex, install it with:**

****

**Task 1: Encryption using different ciphers and modes**

In this task, we will play with various encryption algorithms and modes. You can use the following openssl enc command to encrypt/decrypt a file. To see the manuals, you can type man openssl and man enc.

|  |
| --- |
| % openssl enc ciphertype -e -in plain.txt -out cipher.bin \ -K 00112233445566778889aabbccddeeff \  -iv 0102030405060708 |

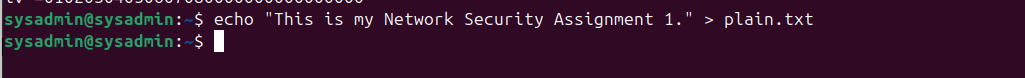
Please replace the ciphertype with a specific cipher type, such as -aes-128-cbc, -aes-128-cfb, -des-cbc, etc. In this task, you should try at least 3 different ciphers and three different modes. You can find the meaning of the command-line options and all the supported cipher types by typing "man enc". We include some common options for the openssl enc command in the following:

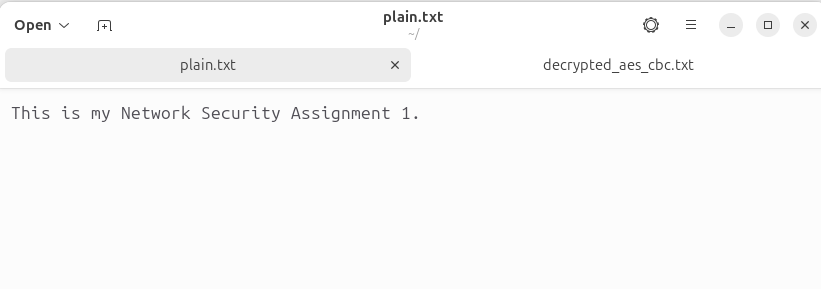
|  |
| --- |
| -in <file> input file  -out <file> output file  -e encrypt  -d decrypt  -K/-iv key/iv in hex is the next argument  -[pP] print the iv/key (then exit if -P) |

**Solution:**

**Step 1: Create a Sample Plaintext File**

First, create a sample plaintext file to encrypt:

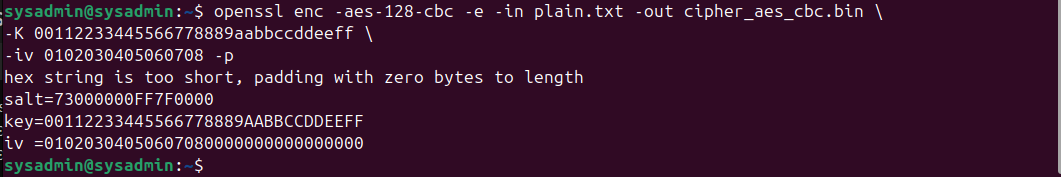




**Step 2: Encrypt Using Different Ciphers and Modes**

Try three different cipher types and modes.

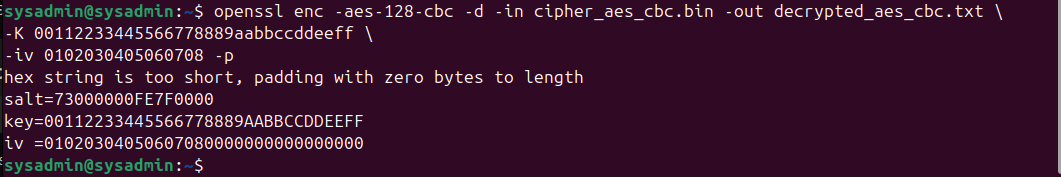
**1. AES-128-CBC Mode (Cipher Block Chaining)**



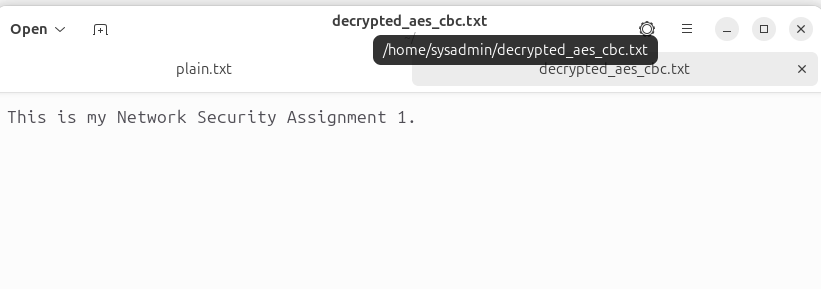
**Decrypt the Encrypted Files**

To decrypt, use the same cipher, key, and IV.

**Decrypt AES-128-CBC**

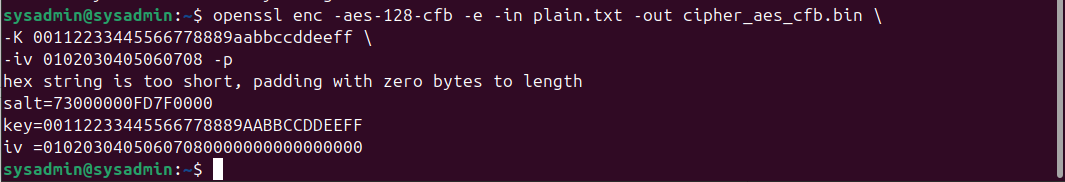






**Encrypt Using Different Ciphers and Modes**

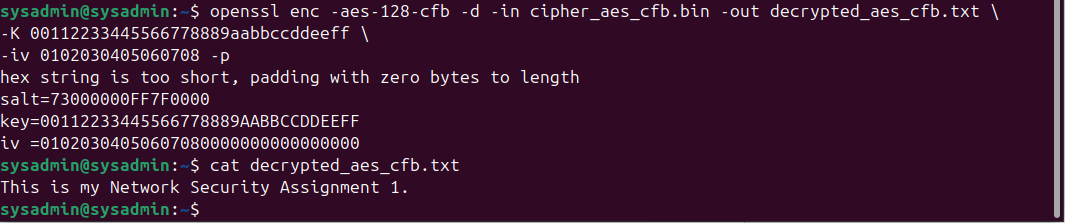
**2. AES-128-CFB Mode (Cipher Feedback)**

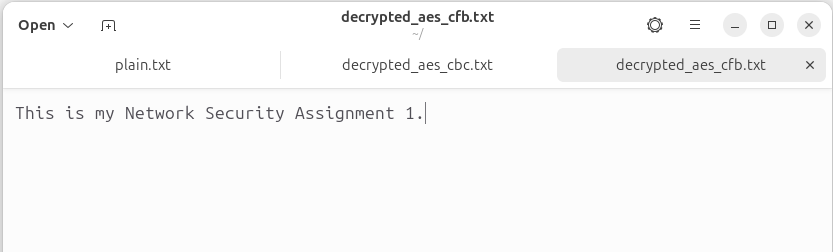


**Decrypt the Encrypted Files**

To decrypt, use the same cipher, key, and IV.

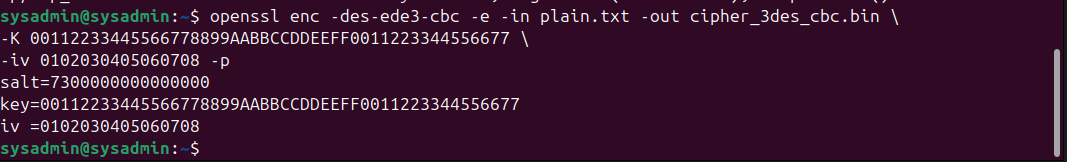
**Decrypt AES-128-CFB**





**Encrypt Using Different Ciphers and Modes**

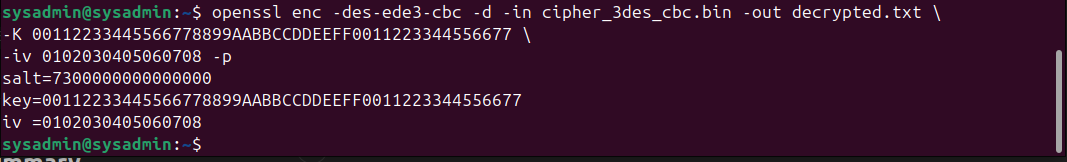
**3. DES-EDE3-CBC Mode (Data Encryption Standard)**

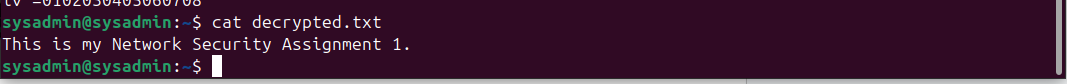


**Decrypt the Encrypted Files**

To decrypt, use the same cipher, key, and IV.

**Decrypt DES-EDE3-CBC Mode**





**Task 2: Encryption Mode – ECB vs. CBC**

The file pic original.bmp contains a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

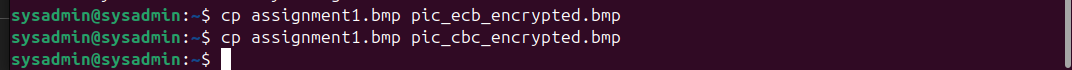
1. Let us treat the encrypted picture as a picture, and use a picture viewing software to display it. How ever, the first 54 bytes of a .bmp file contain the header information about the picture. We have to set these bytes correctly, so that the encrypted file can be treated as a legitimate .bmp file. We will replace the header of the encrypted picture with that of the original picture. You can use the ghex tool to directly modify binary files.

2. Display the encrypted picture using any picture viewing software. Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

**Solution:**

**Step 1: Make a Copy of the Original BMP**

Before encryption, make a copy of the original image for comparison:

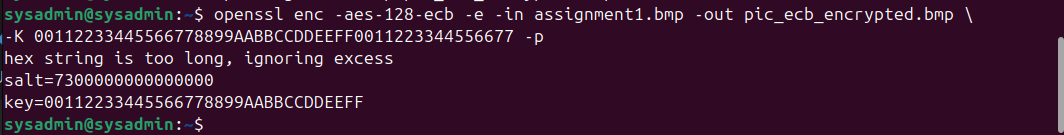


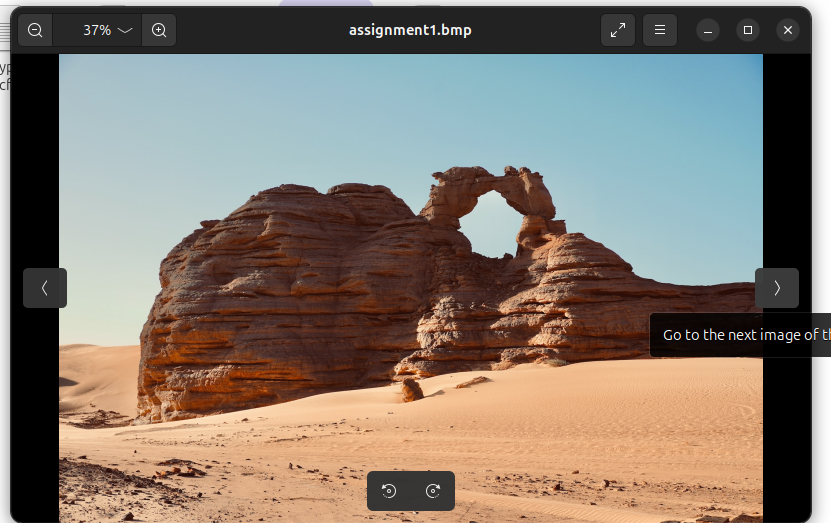
**Step 2: Encrypt the BMP File**

A BMP file consists of **a 54-byte header** followed by raw pixel data.

**Encrypt Using ECB Mode**

Use **AES-128-ECB** to encrypt the image:





**Step 3: Restore the BMP Header**

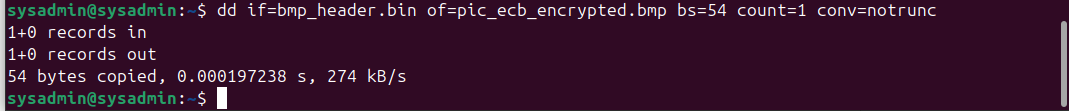
Since encryption modifies the entire file, **we must restore the first 54 bytes of the original BMP header**.

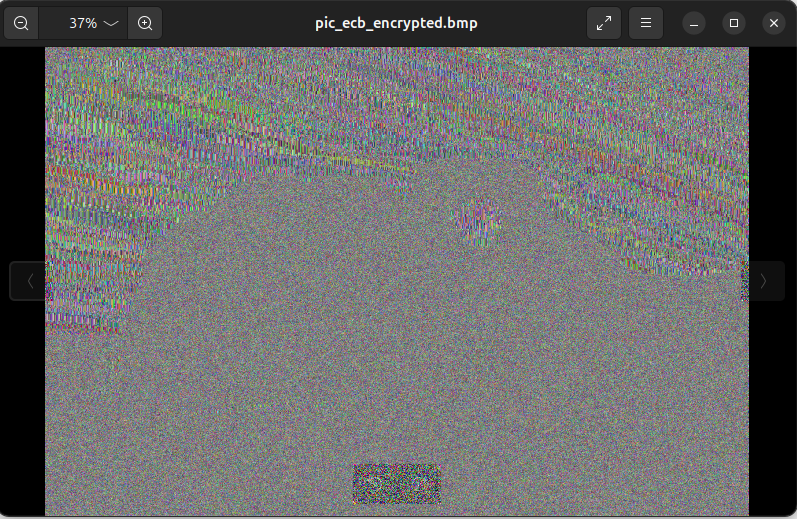
**Extract the Original Header**



**Restore the Header for the Encrypted Images**

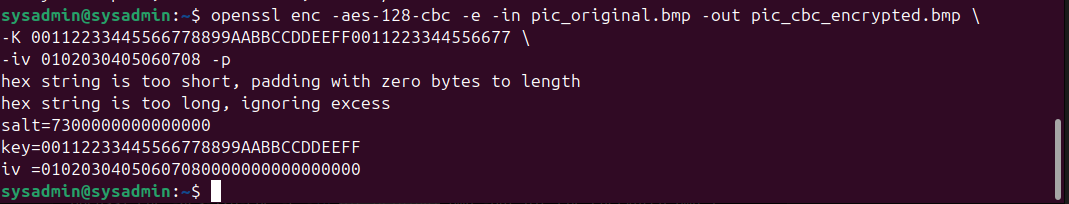
For ECB-encrypted BMP:





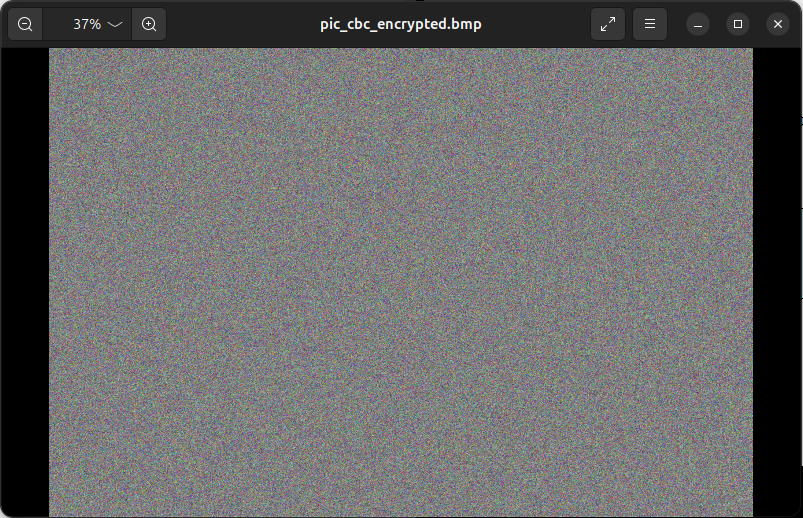
**Encrypt Using CBC Mode**

Use **AES-128-CBC** to encrypt the image:



**Restore the Header for the Encrypted Images**

For CBC-encrypted BMP:



**Task 3: Encryption Mode – Corrupted Cipher Text**

To understand the properties of various encryption modes, we would like to do the following exercise: 1. Create a text file that is at least 64 bytes long.

2. Encrypt the file using the AES-128 cipher.

3. Unfortunately, a single bit of the 30th byte in the encrypted file got corrupted. You can achieve this corruption using ghex.

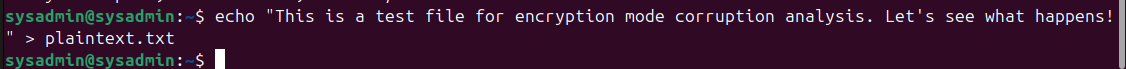
4. Decrypt the corrupted file (encrypted) using the correct key and IV.

Please answer the following questions: (1) How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively? Please answer this question before you conduct this task, and then find out whether your answer is correct or wrong after you finish this task. (2) Explain why. (3) What are the implication of these differences?

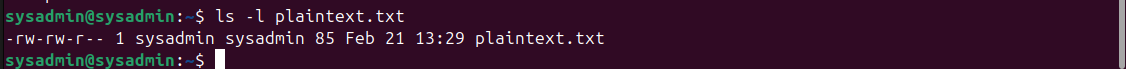
**Solution:**

**Create a Text File (At Least 64 Bytes)**

Create a simple text file (plaintext.txt) with at least **64 characters**:

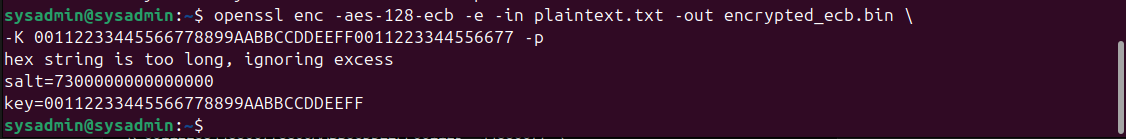


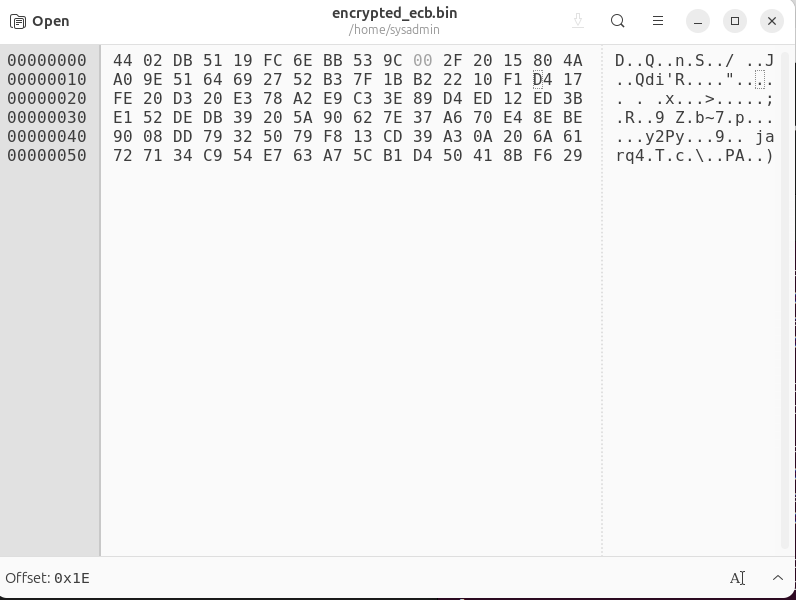
**Verify the file size:**



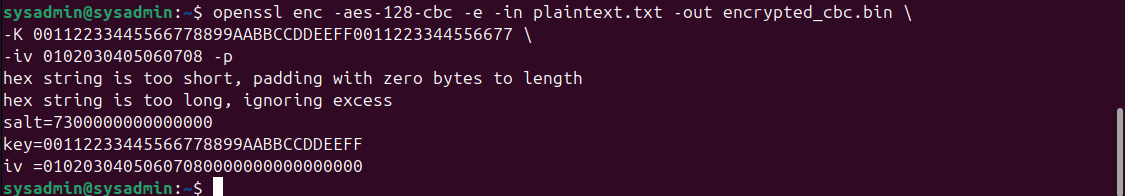
We will encrypt the file using **four different modes**: ECB, CBC, CFB, and OFB.

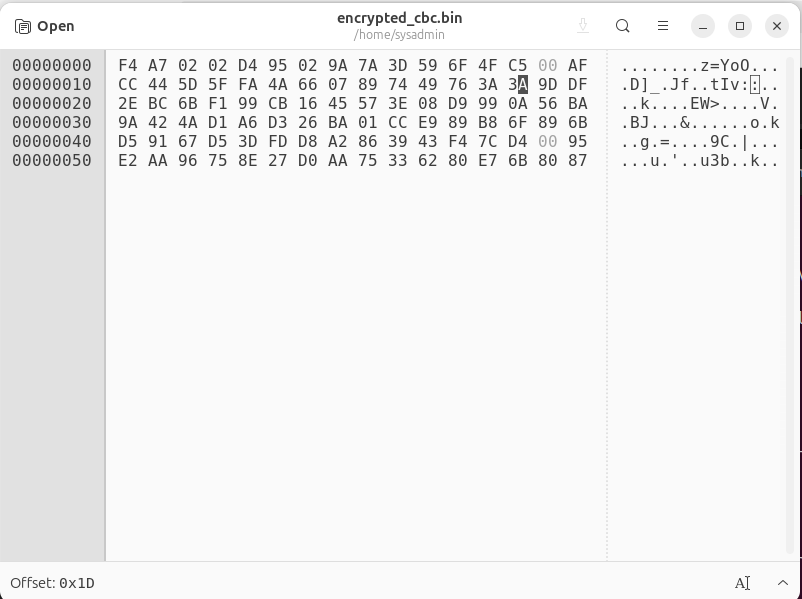
**Encrypt Using AES-128-ECB**



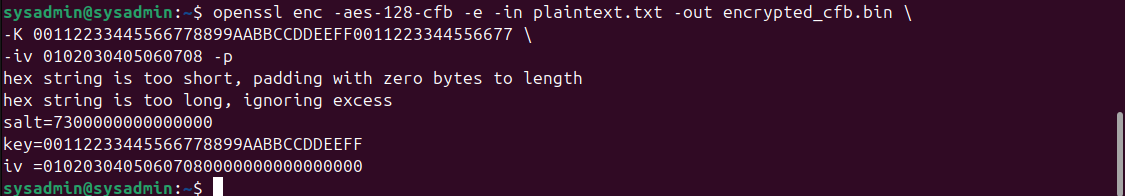


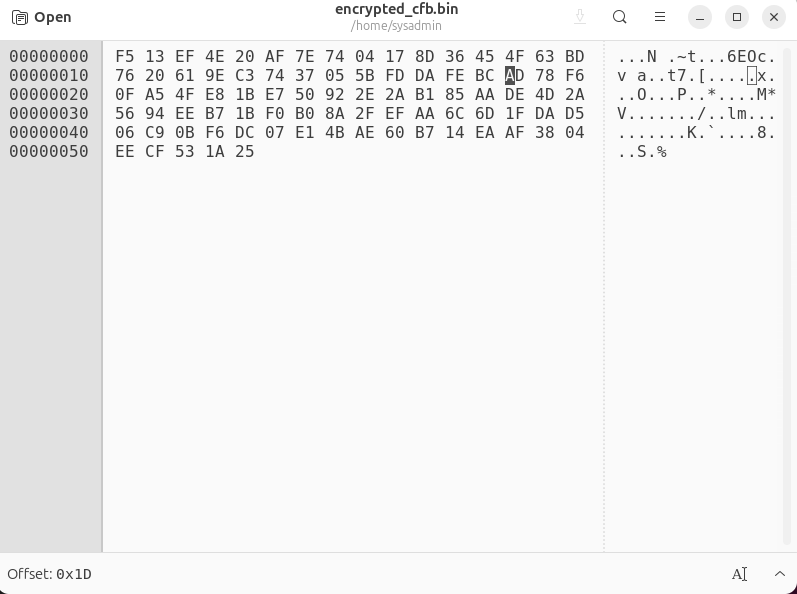
**Encrypt Using AES-128-CBC**



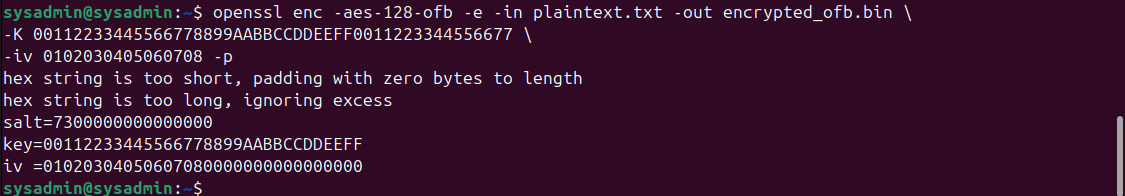


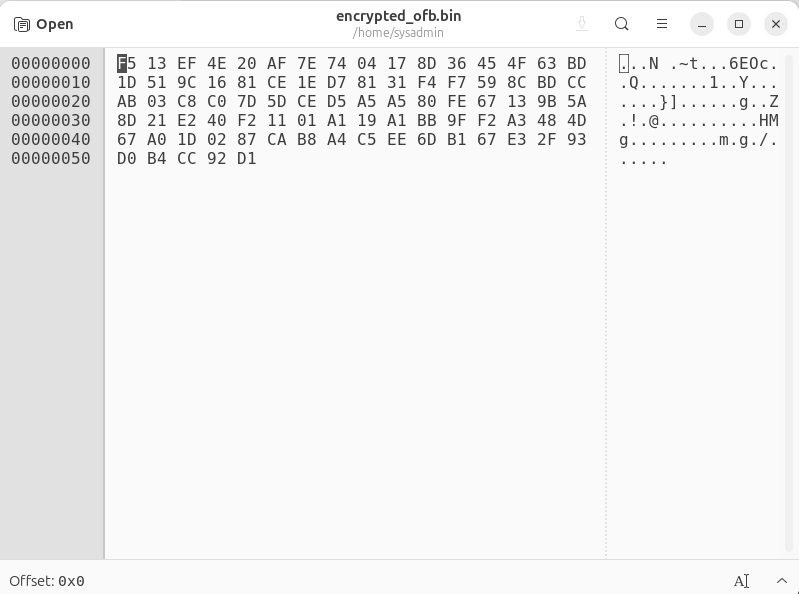
**Encrypt Using AES-128-CFB**





**Encrypt Using AES-128-OFB**





**Corrupt the 30th Byte in Each Encrypted File**

We now simulate a **bit-flip error** in each file using **GHex**.

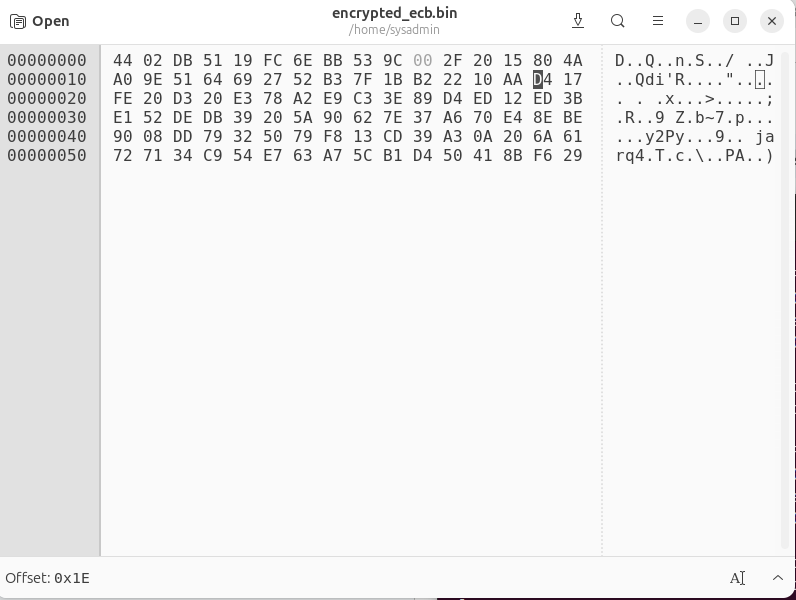
**Using GHex:**

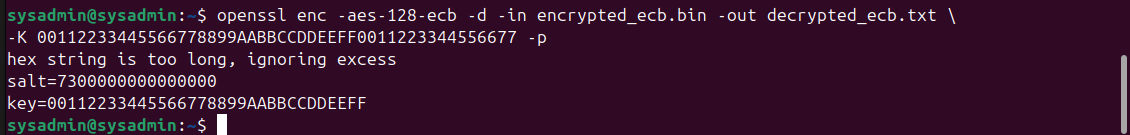
1. Open **GHex** (ghex &).
2. Open **each encrypted file (e.g., encrypted\_ecb.bin)**.
3. Navigate to the **30th byte** (count from the first byte).
4. Modify **a single bit** in that byte (e.g., change A3 to A7).
5. Save and exit.

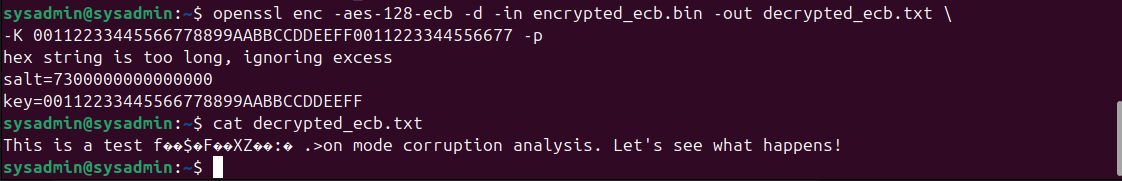
**Step 4: Decrypt the Corrupted Files**

We now attempt to decrypt the corrupted files.

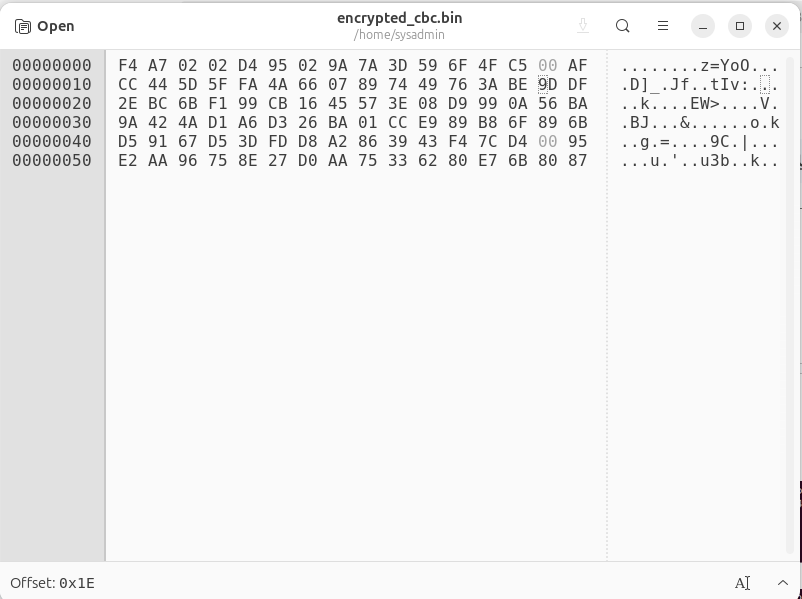
**Decrypt ECB**

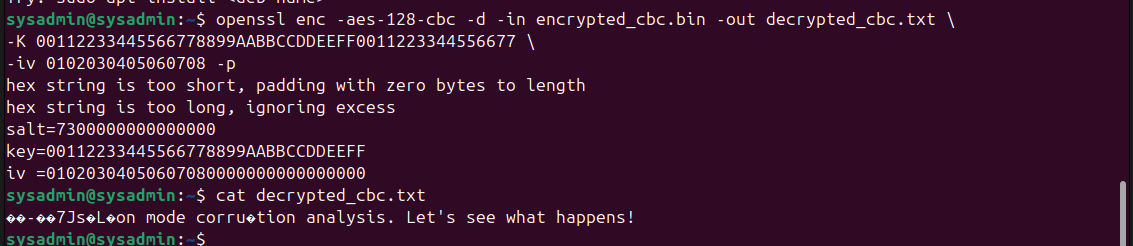




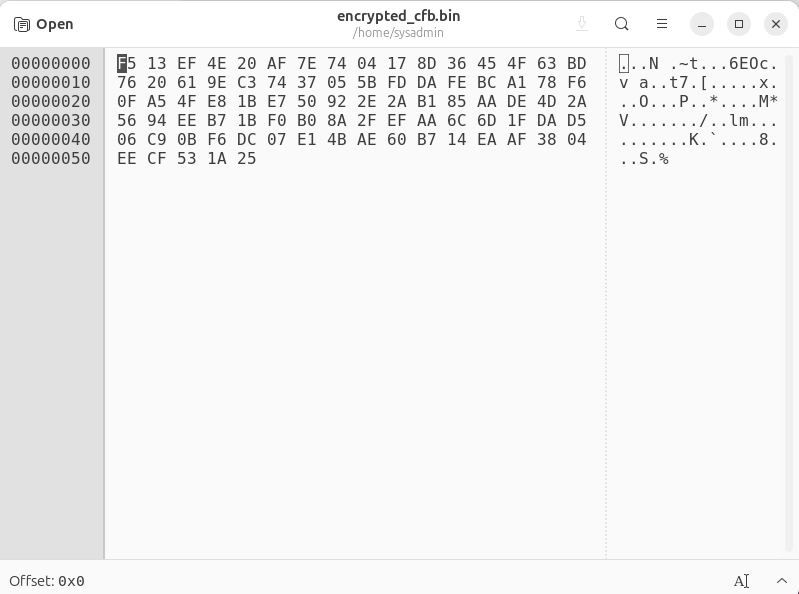


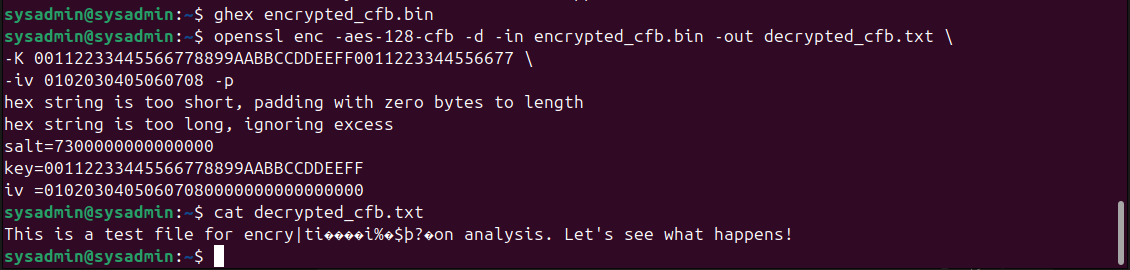
**Decrypt CBC**



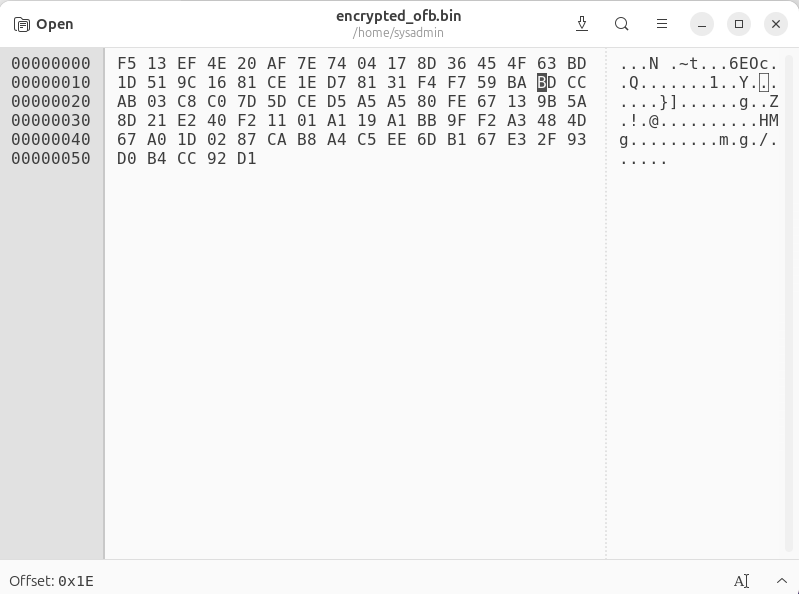


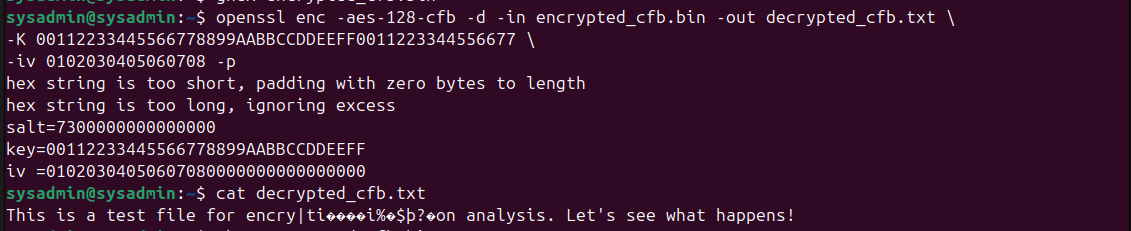
**Decrypt CFB**





**Decrypt OFB**





**Step 5: Analyze the Effects of Corruption**

Now, let's answer the key questions.

**(1) What happens when you decrypt the corrupted files?**

|  |  |
| --- | --- |
| **Mode** | **Effect of Corrupted Byte** |
| **ECB** | Only the **corrupted block (16 bytes)** is affected. The rest decrypts normally. |
| **CBC** | The **corrupted block** is fully damaged, **AND** the next block is partially corrupted. |
| **CFB** | Only the **corrupted byte** is affected. The rest of the plaintext decrypts correctly. |
| **OFB** | Only the **corrupted byte** is affected, similar to CFB. |

**(2) Why do these effects occur?**

* **ECB Mode**: Each block is encrypted independently. A corruption **only affects its corresponding block**.
* **CBC Mode**: Blocks depend on the previous block’s ciphertext (chaining effect). A corrupted byte in block **N** affects **all of block N and part of block N+1**.
* **CFB Mode**: Works like a **stream cipher**, so only **one byte is corrupted**, but the rest remains intact.
* **OFB Mode**: Also works like a stream cipher; **corruption is limited to a single byte**.

**(3) What are the implications of these differences?**

* **ECB is weak** because **patterns remain visible** and corruption is localized.
* **CBC is more secure** but corruption spreads across blocks.
* **CFB and OFB are resilient** to corruption, as only **one byte is affected**.

**Security Implications**

1. **Use CBC for security**, but be aware of error propagation.
2. **ECB should never be used** for sensitive data.
3. **CFB and OFB** are useful when error tolerance is needed.

**Conclusion**

This experiment highlights the **weaknesses of ECB mode**, the **error propagation effect of CBC**, and the **resilience of CFB/OFB**. It also shows why **choosing the right encryption mode matters** for security.

**Task 4: Padding**

For block ciphers, when the size of the plaintext is not the multiple of the block size, padding may be required. In this task, we will study the padding schemes. Please do the following exercises:

1. The openssl manual (try man enc) says that openssl uses PKCS5 (Public Key Cryptography Standard) for its padding. In this standard the value of each byte is the number of bytes that are added – for example, if padding is required for 4 bytes, the hexadecimal padding string 04 04 04 04 is used. If the input data is a multiple of the block size, then an extra block of bytes with the block size value is added – for example, if the input data is 80 bytes and the block size is 16 bytes, an extra padding block of 16 bytes, with each byte 10 in hexadecimal, is added.

Design an experiment to verify the padding scheme. In particular, use your experiment to figure out the paddings in the AES encryption when the length of the plaintext is 20 bytes and 32 bytes.

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

**Solution:**

The goal of the experiment is to verify and understand the PKCS5 padding used in OpenSSL for AES encryption.   
PKCS5 is used in block ciphers and pads the plaintext to ensure its length is a multiple of the block size.   
For AES with a 128-bit block size (16 bytes), PKCS5 padding adds padding bytes to the plaintext such that the total length of the plaintext becomes a multiple of 16 bytes.

The padding scheme is as follows:

* If the plaintext is already a multiple of the block size (16 bytes for AES), PKCS5 will add a block of 16 padding bytes (each byte set to 0x10).
* If the plaintext is shorter than a block (not a multiple of 16 bytes), PKCS5 will add n padding bytes, each of the value n, where n is the number of bytes required to make the length a multiple of the block size.

Prepare plaintext files of 20 bytes and 32 bytes

*echo -n "This is 20 bytes!!!" > plaintext20.txt  
echo -n "This is a plaintext of 32 bytes" > plaintext32.txt*

Ecrypt the files with aes-128 encryption in cbc mode

*openssl enc -aes-128-cbc -in plaintext20.txt -out ciphertext20.bin -pass pass:mysecretkey  
openssl enc -aes-128-cbc -in plaintext32.txt -out ciphertext32.bin -pass pass:mysecretkey*

Examine the lengths of the encrypted files

*ls -l ciphertext20.bin  
ls -l ciphertext32.bin*

It is observed that the length of both encrypted files is in multiple of 16bytes (block size)

For the 20-byte plaintext, since it is not a multiple of 16, the padding will be 16 - (20 % 16) = 12 bytes.   
The padding byte value will be 0x0C (12 in hexadecimal) according to the PKCS5 padding scheme.  
The padding can be verified by decrypting the ciphertext with **-**nopad option and viewing the decrypted file in GHex  
*command: openssl enc -aes-128-ecb -d -nopad -in ciphertext20.bin -out ciphertext20\_dec -pass pass:mysecretkey*

For the 32-byte plaintext, 32 is already a multiple of 16, hence a full block of padding i.e. 16 bytes with value 16 (10 hex) is added as per PKCS5 scheme

The padding can be verified by decrypting the ciphertext with -nopad option and viewing the decrypted file in GHex  
*command: openssl enc -aes-128-cbc -d -nopad -in ciphertext32.bin -out ciphertext32\_dec -pass pass:mysecretkey*

* In ECB, padding is required, since the plaintext is divided into blocks, and each block is encrypted independently.
* In CBC, each ciphertext block depends on the previous ciphertext block, with an Initialization Vector (IV) used for the first block. Despite the chaining of blocks, if the length of the plaintext isn't a multiple of the block size, padding is required to make the plaintext align to the block size (16 bytes for AES). This padding is necessary because CBC mode works on complete blocks of plaintext, and any leftover portion must be padded.
* In CFB, the encryption process uses a feedback mechanism where the previous ciphertext block (or part of it) is used to generate the key stream for the current plaintext block. The mode works by processing the plaintext in smaller segments (e.g., 8, 16, or 128 bits at a time), rather than encrypting it in full blocks. This means that CFB can process plaintext of any length without needing padding.
* In OFB, a keystream is generated independently of the plaintext and ciphertext. The keystream is XORed with the plaintext to produce the ciphertext. Because the keystream is independent of the plaintext length and generated continuously, OFB can encrypt any length of plaintext without needing padding, regardless of whether the plaintext is a multiple of the block size. Hence padding is not required.

**Task 5: Programming using the Crypto Library**

So far, we have learned how to use the tools provided by openssl to encrypt and decrypt messages. In this task, we will learn how to use openssl’s crypto library to encrypt/decrypt messages in programs. OpenSSL provides an API called EVP, which is a high-level interface to cryptographic functions. Al though OpenSSL also has direct interfaces for each individual encryption algorithm, the EVP library pro vides a common interface for various encryption algorithms. To ask EVP to use a specific algorithm, we simply need to pass our choice to the EVP interface. A sample code is given in http://www.openssl. org/docs/crypto/EVP\_EncryptInit.html. Please get yourself familiar with this program, and then do the following exercise.

You are given a plaintext and a ciphertext, and you know that aes-128-cbc is used to generate the ciphertext from the plaintext, and you also know that the numbers in the IV are all zeros (not the ASCII character ‘0’). Another clue that you have learned is that the key used to encrypt this plaintext is an English word shorter than 16 characters; the word that can be found from a typical English dictionary. Since the word has less than 16 characters (i.e. 128 bits), space characters (hexadecimal value 0x20) are appended to the end of the word to form a key of 128 bits. Your goal is to write a program to find out this key. You can download a English word list from the Internet. We have also linked one on the web page of this lab. The plaintext and ciphertext are as follows:

|  |
| --- |
| Plaintext (total 21 characters): This is a top secret. Ciphertext (in hex format): 8d20e5056a8d24d0462ce74e4904c1b5 13e10d1df4a2ef2ad4540fae1ca0aaf9 |

Note 1: If you choose to store the plaintext message in a file, and feed the file to your program, you need to check whether the file length is 21. Some editors may add a special character to the end of the file. If that happens, you can use the ghex tool to remove the special character.

Note 2: In this task, you are supposed to write your own program to invoke the crypto library. No credit will be given if you simply use the openssl commands to do this task.

Note 3: To compile your code, you may need to include the header files in openssl, and link to openssl libraries. To do that, you need to tell your compiler where those files are.

In your Makefile, you may want to specify the following:

|  |
| --- |
| IDIR=/usr/include/  LDIR=/usr/lib/x86\_64-linux-gnu  all:  gcc -I$(IDIR) -L$(LDIR) -o file file.c -lcrypto -ldl |

**Solution:**

* *Read the dictionary file to get all possible words.*
* *Pad each word with spaces (0x20) to make it 16 bytes long.*
* *Use OpenSSL's EVP API to decrypt the ciphertext using each padded key.*
* *Compare the decrypted text with the given plaintext.*
* *Print the correct key when found.*

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <string.h>*

*#include <openssl/evp.h>*

*#define KEY\_SIZE 16*

*#define BLOCK\_SIZE 16*

*#define PLAINTEXT\_LEN 21*

*#define CIPHERTEXT\_LEN 32*

*// Given plaintext and ciphertext*

*const unsigned char plaintext[PLAINTEXT\_LEN] = "This is a top secret.";*

*const unsigned char expected\_ciphertext[CIPHERTEXT\_LEN] = { 0x8d, 0x20, 0xe5, 0x05, 0x6a, 0x8d, 0x24, 0xd0, 0x46, 0x2c, 0xe7, 0x4e, 0x49, 0x04, 0xc1, 0xb5, 0x13, 0xe1, 0x0d, 0x1d, 0xf4, 0xa2, 0xef, 0x2a, 0xd4, 0x54, 0x0f, 0xae, 0x1c, 0xa0, 0xaa, 0xf9 };*

*void handleErrors() {*

*fprintf(stderr, "An error occurred\n");*

*exit(1);*

*}*

*// AES encryption function using EVP API*

*int aes\_encrypt(const unsigned char \*key, const unsigned char \*plaintext, unsigned char \*ciphertext) {*

*EVP\_CIPHER\_CTX \*ctx;*

*int len, ciphertext\_len;*

*unsigned char iv[BLOCK\_SIZE] = {0}; // IV is all zeros*

*// Create and initialize context*

*if (!(ctx = EVP\_CIPHER\_CTX\_new())) handleErrors();*

*// Initialize encryption operation*

*if (EVP\_EncryptInit\_ex(ctx, EVP\_aes\_128\_cbc(), NULL, key, iv) != 1) handleErrors();*

*// Provide the plaintext to encrypt*

*if (EVP\_EncryptUpdate(ctx, ciphertext, &len, plaintext, PLAINTEXT\_LEN) != 1) handleErrors();*

*ciphertext\_len = len;*

*// Finalize encryption*

*if (EVP\_EncryptFinal\_ex(ctx, ciphertext + len, &len) != 1) handleErrors();*

*ciphertext\_len += len;*

*// Clean up*

*EVP\_CIPHER\_CTX\_free(ctx);*

*return ciphertext\_len;*

*}*

*int main() {*

*FILE \*wordlist = fopen("/home/sysadmin/prg/wordlist.txt", "r");*

*if (!wordlist) {*

*perror("Error opening wordlist file");*

*//return 1;*

*handleErrors();*

*}*

*char word[KEY\_SIZE] = {0}; // Buffer for reading words*

*unsigned char key[KEY\_SIZE] = {0}; // Key buffer*

*unsigned char ciphertext[CIPHERTEXT\_LEN] = {0}; // Buffer for encryption output*

*while (fgets(word, KEY\_SIZE, wordlist)) {*

*word[strcspn(word, "\r\n")] = '\0'; // Remove newline*

*// Prepare the key (pad with spaces)*

*memset(key, 0x20, KEY\_SIZE); // Fill with spaces*

*memcpy(key, word, strlen(word)); // Copy word*

*//printf("trying key '%s' ...\n",word);*

*// Encrypt the plaintext*

*aes\_encrypt(key, plaintext, ciphertext);*

*// Compare encrypted output with expected ciphertext*

*if (memcmp(ciphertext, expected\_ciphertext, CIPHERTEXT\_LEN) == 0) {*

*printf("Found key: '%s' \n", word);*

*printf("Expected Cipher - ");*

*for(int c=0; c<CIPHERTEXT\_LEN; c++) printf("%x",expected\_ciphertext[c]);*

*printf("\n");*

*printf("Obtained Cipher - ");*

*for(int c=0; c<CIPHERTEXT\_LEN; c++) printf("%x",ciphertext[c]);*

*printf("\n");*

*fclose(wordlist);*

*return 0;*

*}*

*}*

*printf("Key not found in wordlist\n");*

*fclose(wordlist);*

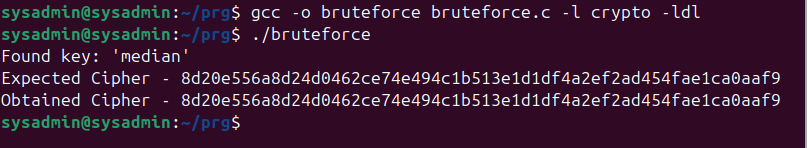
*return 1;*

*}*

1. Save the program as bruteforce.c.
2. Download a wordlist (e.g., from /usr/share/dict/words or another online source) and save it as wordlist.txt.
3. Compile using: gcc -o bruteforce bruteforce.c -l crypto -ldl
4. Run it: ./bruteforce

If the correct key exists in the wordlist, the program will print

Found key: 'your\_key\_here'

****

***Wordlist*** [***https://seedsecuritylabs.org/Labs\_16.04/Crypto/Crypto\_Encryption/files/words.txt***](https://seedsecuritylabs.org/Labs_16.04/Crypto/Crypto_Encryption/files/words.txt)