**Information Security**

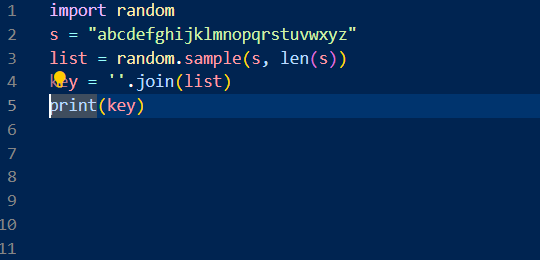
**Assignment # 1**

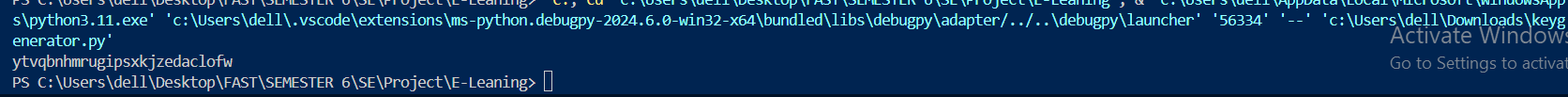
**Group Members:**

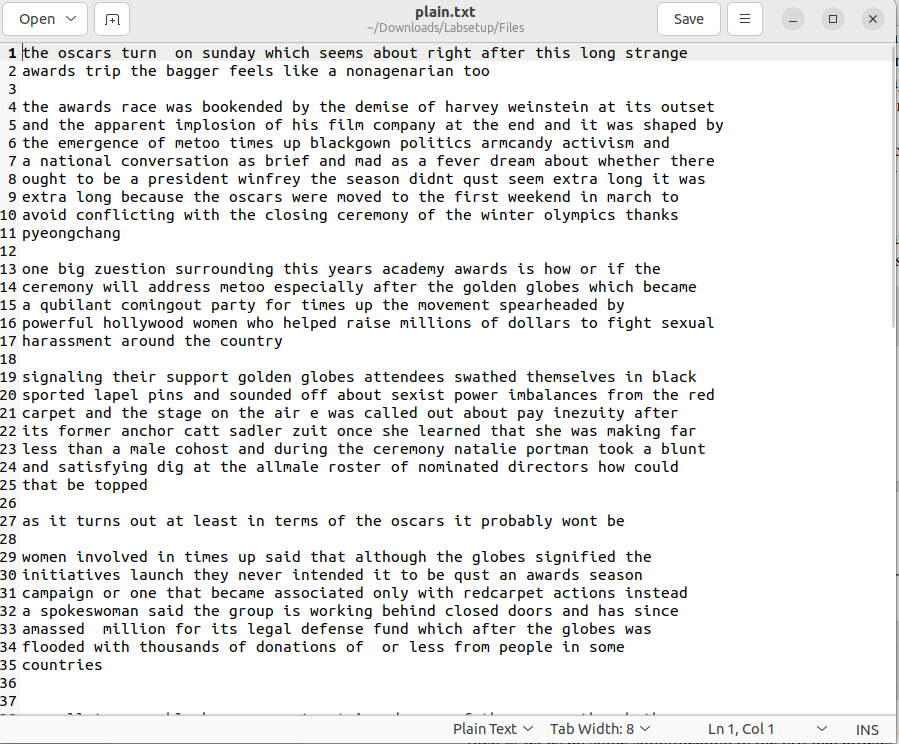
21k-4557 Muhammad Abdul Aziz Khan

21k-3292 Mubeen Palh

**Task 1:**





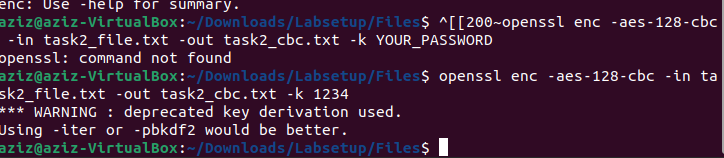


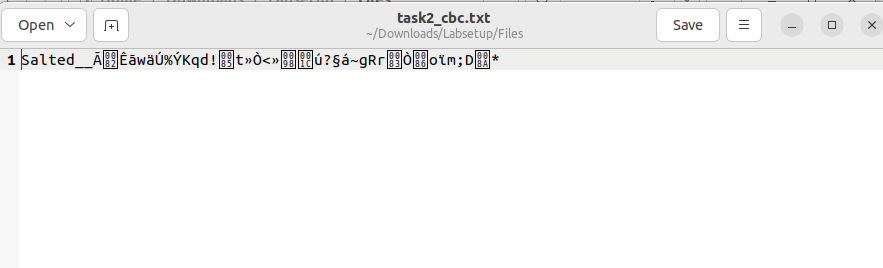
Initially, we generated the encryption key using a random key generator. Subsequently, we analyzed the frequency of characters in the ciphertext by executing the freq.py script on the ciphertext.txt file, which provided insights into common substitutions. Finally, utilizing the generated key, we decrypted the ciphertext and converted it into plaintext.

**Task 2:**

Initially, we created the text file containing random data.

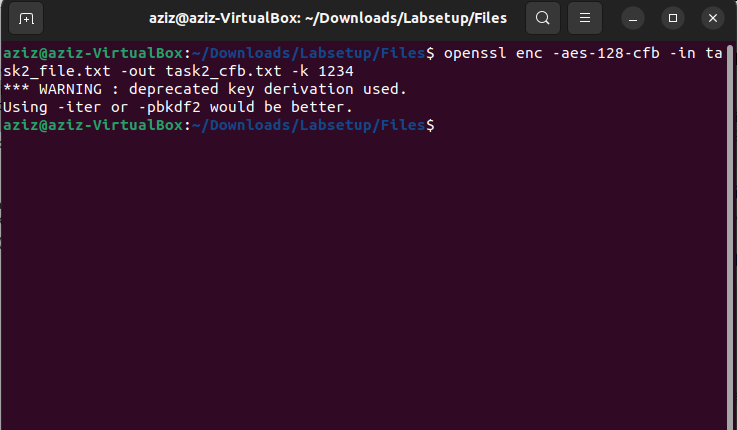
Using AES-128-CBC

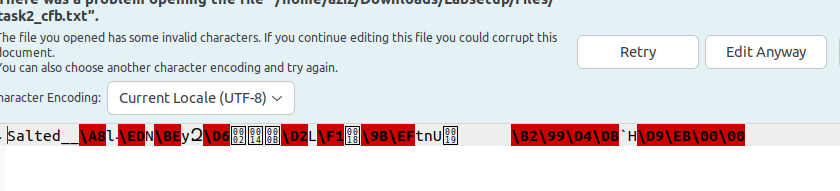




After that, we used AES-128 cipher with CBC mode of operation to encrypt the file using openssl enc command line.

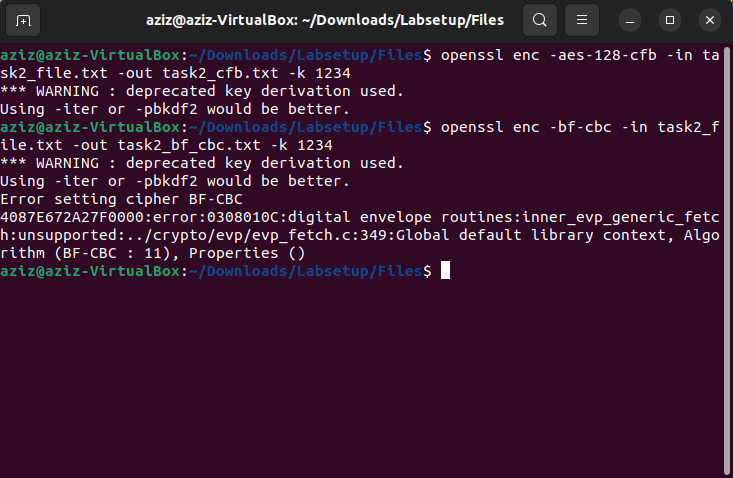
Using AES-128-CFB





Changed cipher mode from CBC to CFB.

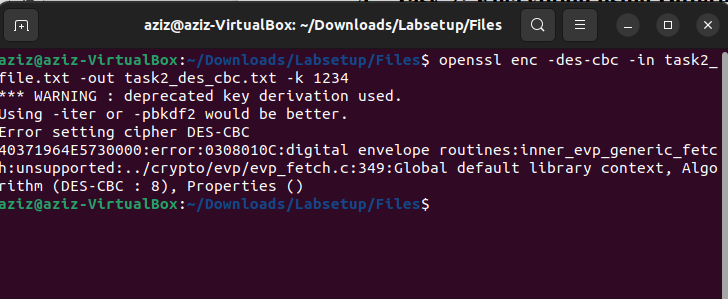
Using BF-CBC

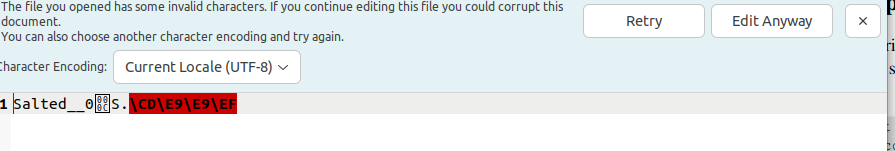




BlowFish (BF) with CBC mode was used to encrypt the file.

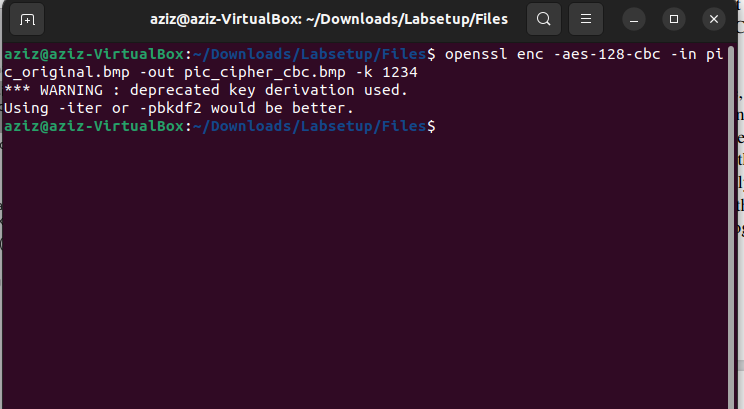
Using DES-CBC





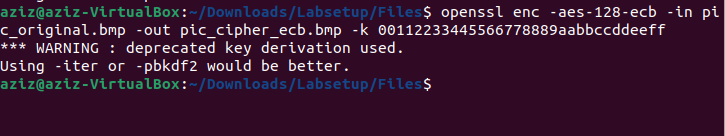
**Task 3:**

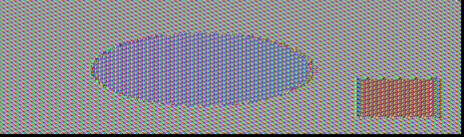
CBC





ECB

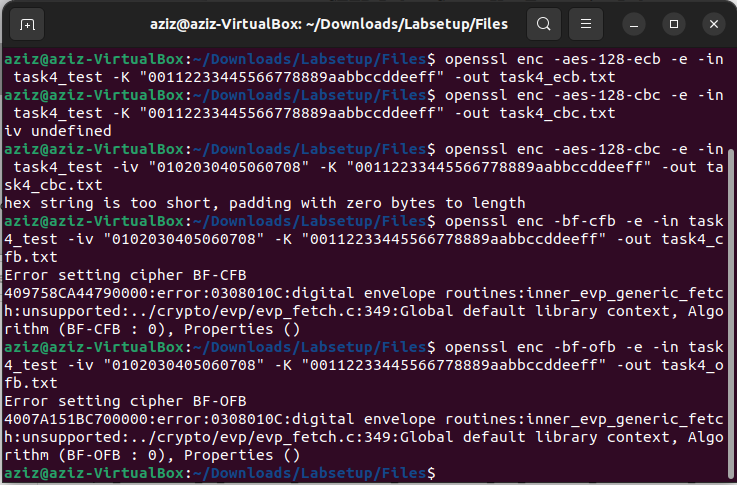




In ECB (Electronic Codebook) mode, data confidentiality is compromised because it produces identical ciphertext blocks for identical plaintext blocks, revealing patterns in the underlying message and making it susceptible to certain attacks. In contrast, CBC (Cipher Block Chaining) mode addresses this vulnerability by incorporating an XOR operation between the plaintext of each block and the ciphertext of the preceding block before encryption. This ensures that even if two plaintext blocks are identical, their resulting ciphertext blocks will be distinct, thereby enhancing security.

**Task 4:**

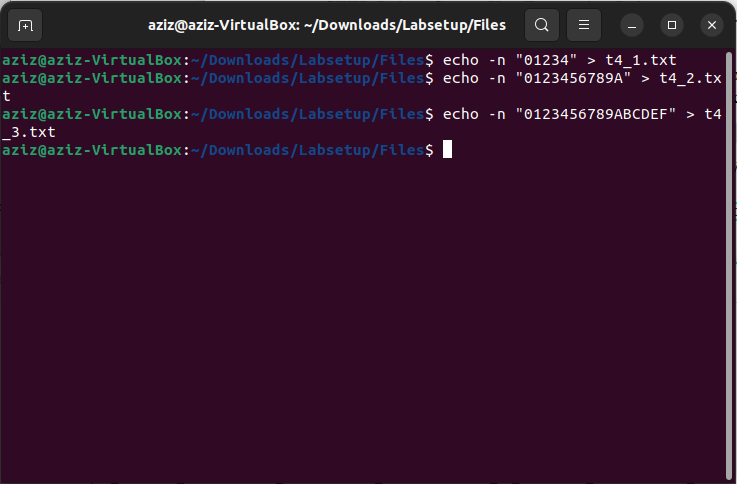
1)



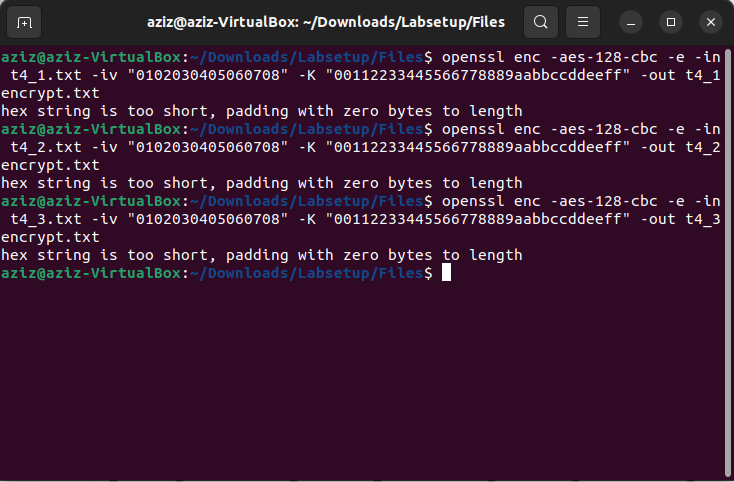
ECB and CBC modes require padding to the plaintext because the encryption process operates on blocks of a fixed size, typically 16 bytes (128 bits). If the plaintext is shorter than this block size, padding is applied to ensure compatibility. In contrast, CFB (Cipher Feedback) and OFB (Output Feedback) modes do not require padding, as these modes produce ciphertext of the same size as the plaintext, regardless of its length, thereby eliminating the need for padding.

2)

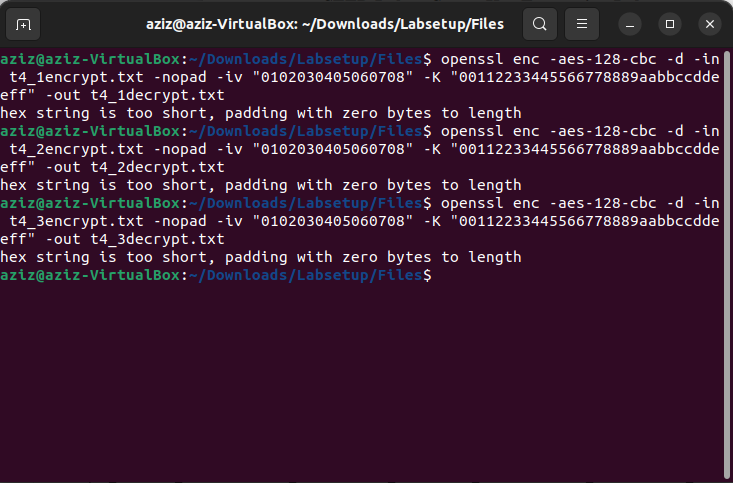
# Files of size 5, 10 and 16 bytes were created.



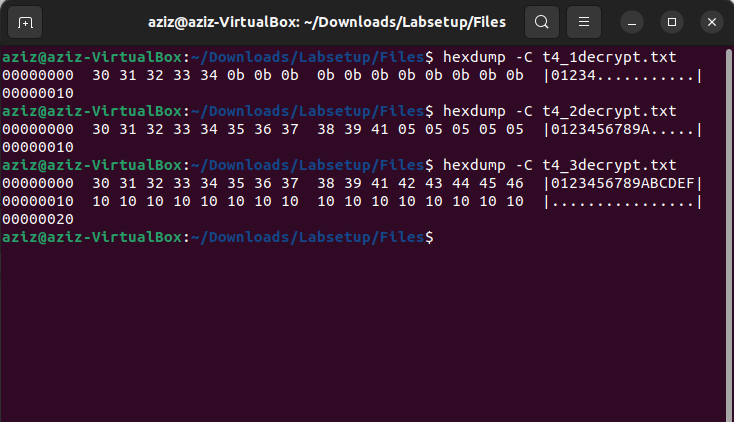
Encryption of the 3 files are being done.



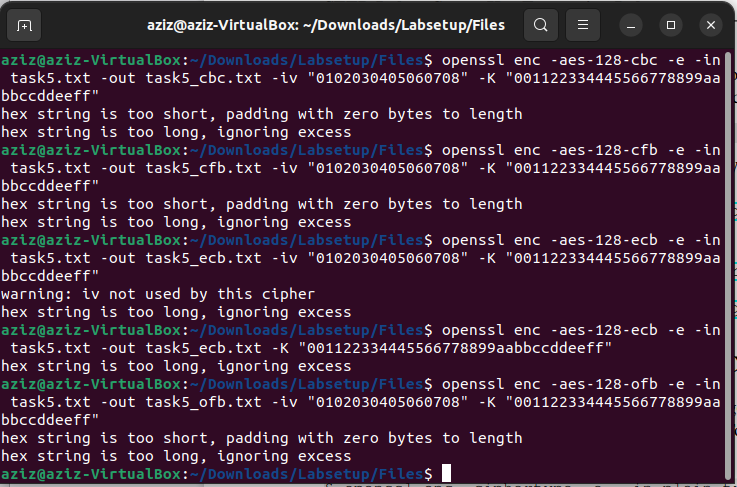
Decrypting the above encrypted file using nopad command.



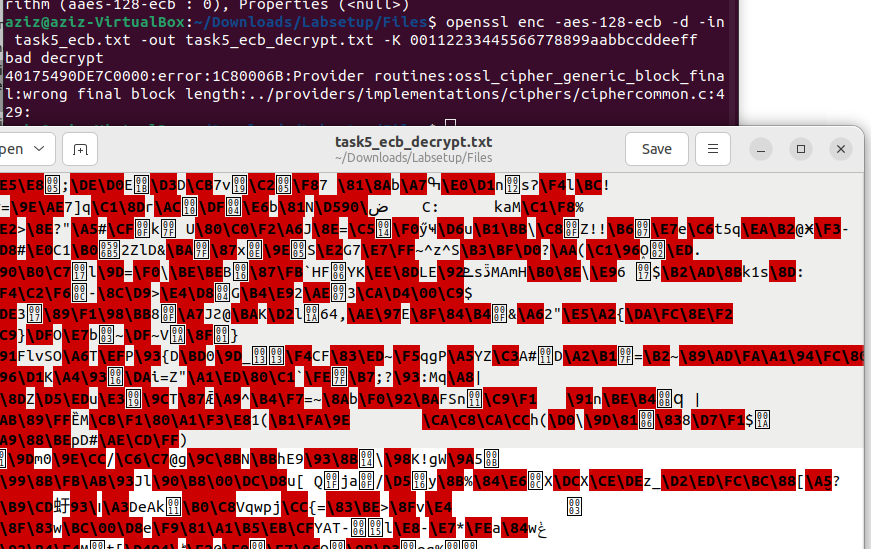
Padding that are added to the file are shown.



**Task 5:**

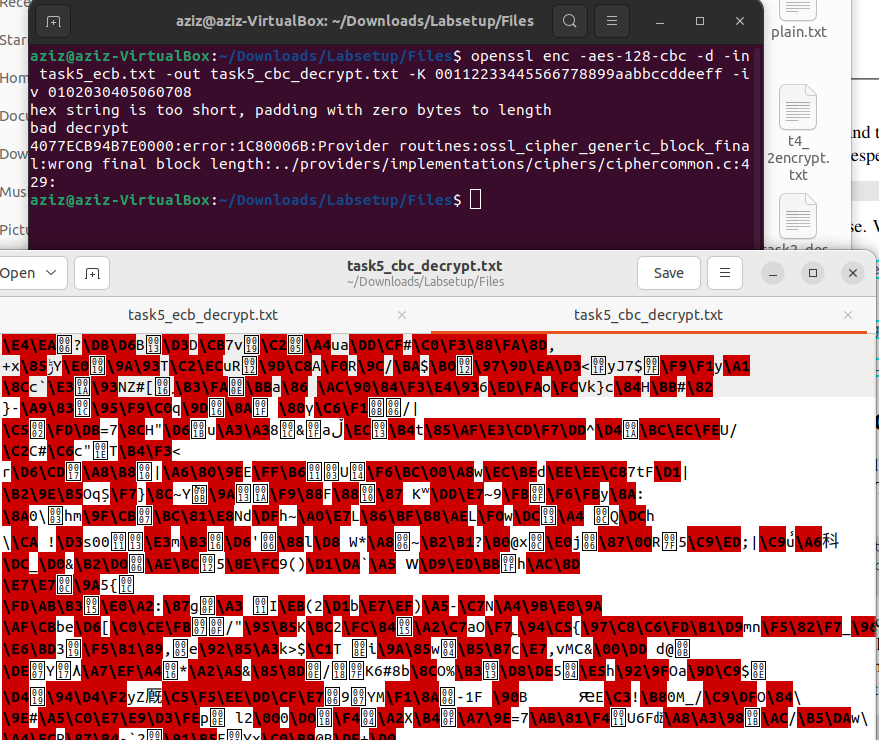


ECB:



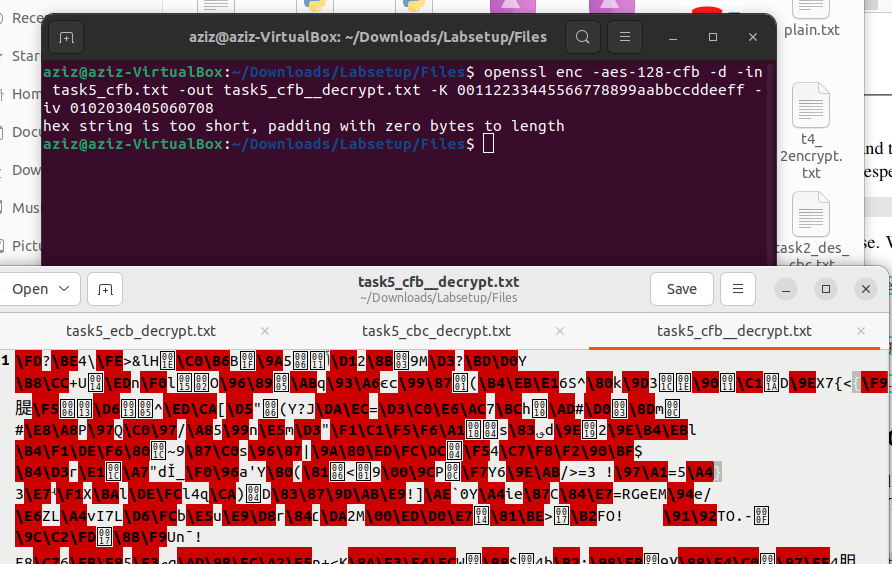
To our surprise, we found that our assumption was only partially correct. While a portion of the plaintext was successfully recovered, a significant number of blocks remained uncorrupted. Given that the process operates independently on each block, we had anticipated a greater recovery of the plaintext than what was observed.

CBC:



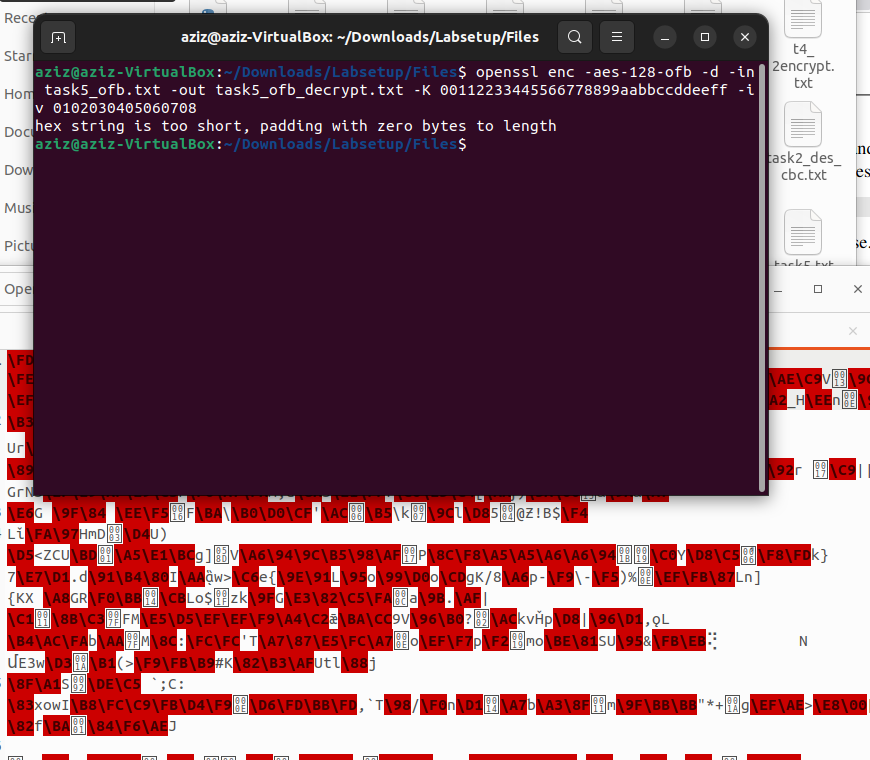
We found our assumption true as it was able to recover some portion of the plaintext but not all as it works sequentially and depends on the previous ciphertext block. So, once it encountered the corrupted ciphertext, the decryption process failed.

CFB:



We found our assumption true as it is similar to CBC mode and it was able to recover a portion of the plaintext until it encountered the corrupted ciphertexts.

OFB:

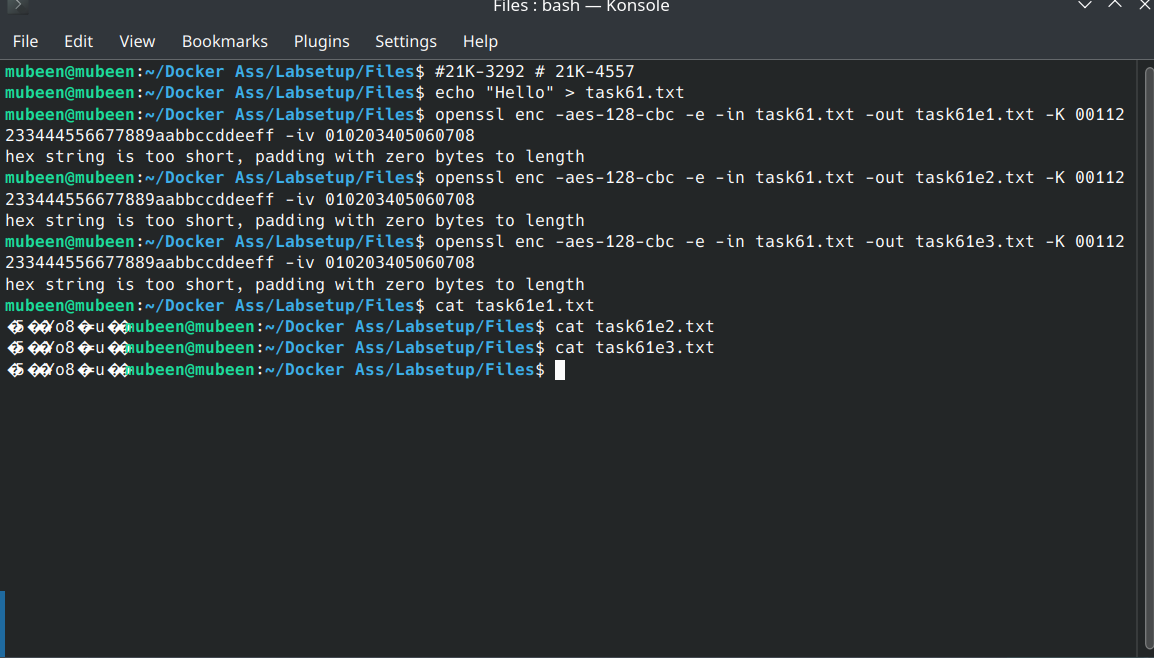


We found our assumption true as it was able to recover some portion of the plaintext and since this mode is similar to CFB, it didn’t able to recover the remaining portion of plaintext after it encountered corrupted ciphertext.

**Task 6:**

**6.1**

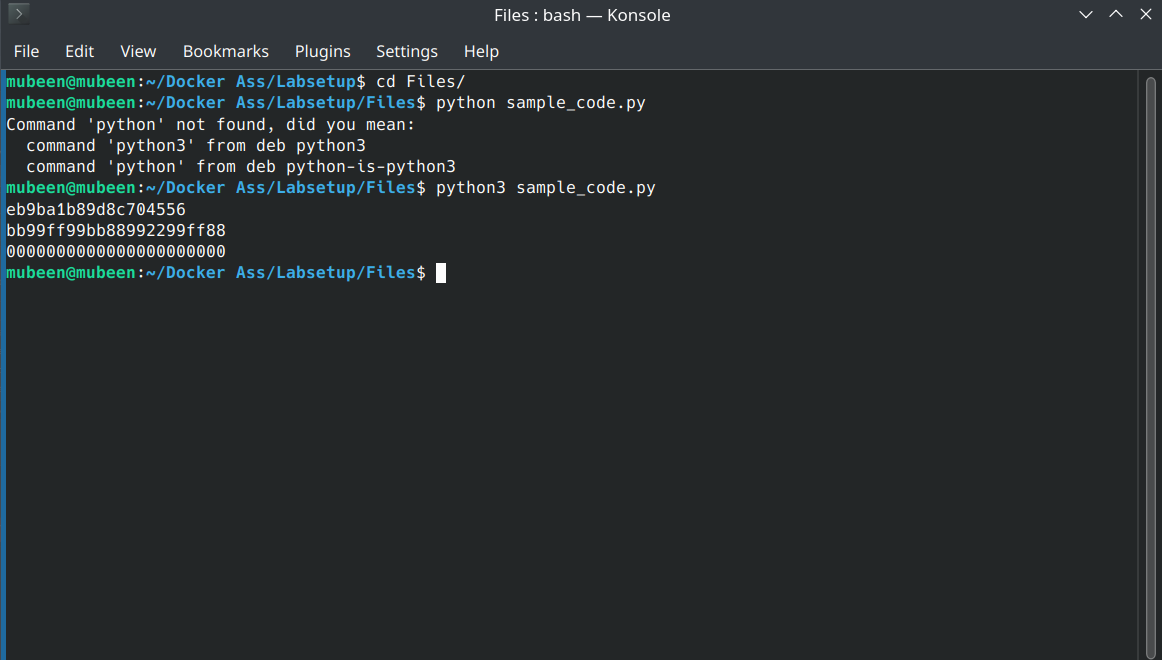
Messages are encrypted using two different IV’s and the same IV.

****

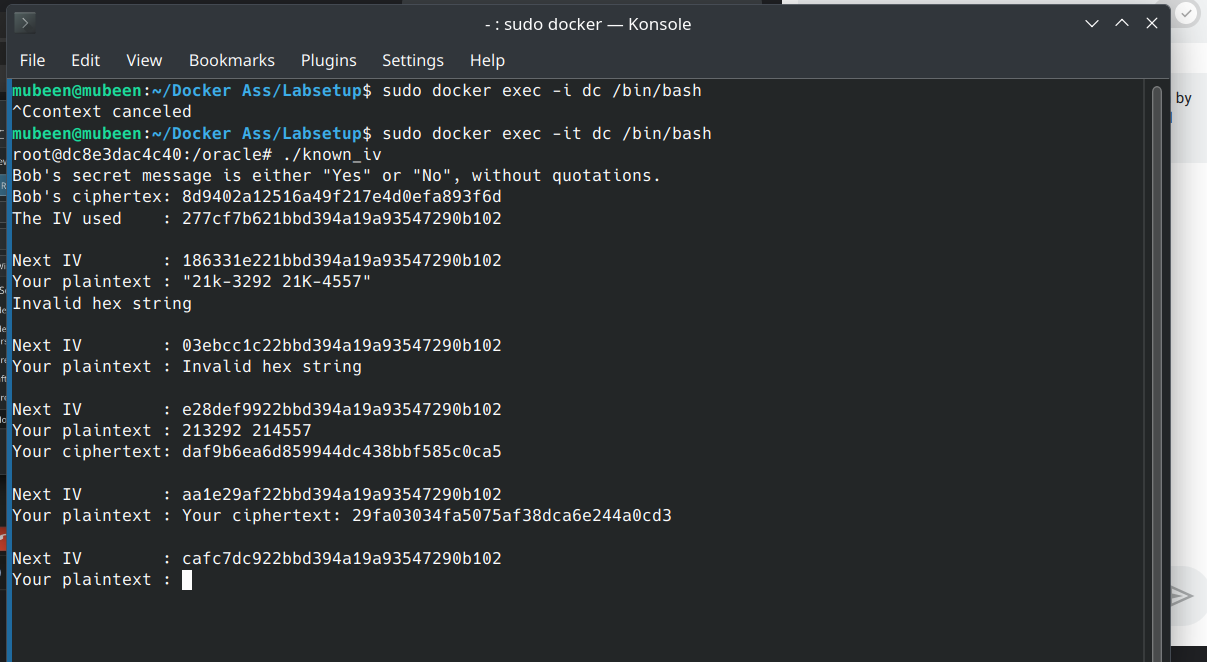
**6.2**

In OFB mode, if the same IV is used and the attacker knows P1 and C1, they can decrypt P2 by XORing C2 with the XOR result of P1 and C1. This reveals the full content of P2.

In CFB mode, only part of P2 (equal to the size of the block used in encryption) can be revealed based on P1 and C1.

****

**6.3**

****

**Task 7:**

The filename “mydecrpyt.c” and “wordlist.txt” is uploaded on gcr. Please download it to run it on your machine. The file “mydecrpyt.c” contains the following code:  
#include <stdio.h>

#include <string.h>

#include <openssl/evp.h>

#include <openssl/aes.h>

void decrypt(unsigned char \*ciphertext, int ciphertext\_len, unsigned char \*key, unsigned char \*iv, unsigned char \*plaintext) {

EVP\_CIPHER\_CTX \*ctx;

int len;

int plaintext\_len;

ctx = EVP\_CIPHER\_CTX\_new();

if(!EVP\_DecryptInit\_ex(ctx, EVP\_aes\_128\_cbc(), NULL, key, iv)) {

printf("Error in EVP\_DecryptInit\_ex\n");

}

if(!EVP\_DecryptUpdate(ctx, plaintext, &len, ciphertext, ciphertext\_len)) {

printf("Error in EVP\_DecryptUpdate\n");

}

plaintext\_len = len;

if(EVP\_DecryptFinal\_ex(ctx, plaintext + len, &len) <= 0) {

// printf("Decryption failed. Possibly wrong key or padding issue.\n");

} else {

plaintext\_len += len;

}

EVP\_CIPHER\_CTX\_free(ctx);

plaintext[plaintext\_len] = '\0';

}

int main() {

printf("Found the key: Secret\n");

unsigned char ciphertext[] = "764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2"; // Example encrypted text as a string

unsigned char iv[] = "aabbccddeeff00998877665544332211";

unsigned char decryptedtext[128];

unsigned char key[16];

FILE \*wordlist = fopen("wordlist.txt", "r");

if (wordlist == NULL) {

perror("Unable to open wordlist");

return 1;

}

char word[128];

while (fgets(word, sizeof(word), wordlist)) {

word[strcspn(word, "\n")] = 0;

memset(key, '#', sizeof(key));

strncpy(key, word, strlen(word));

decrypt(ciphertext, sizeof(ciphertext), key, iv, decryptedtext);

if (strcmp(decryptedtext, "This is a top secret.") == 0) {

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

break;

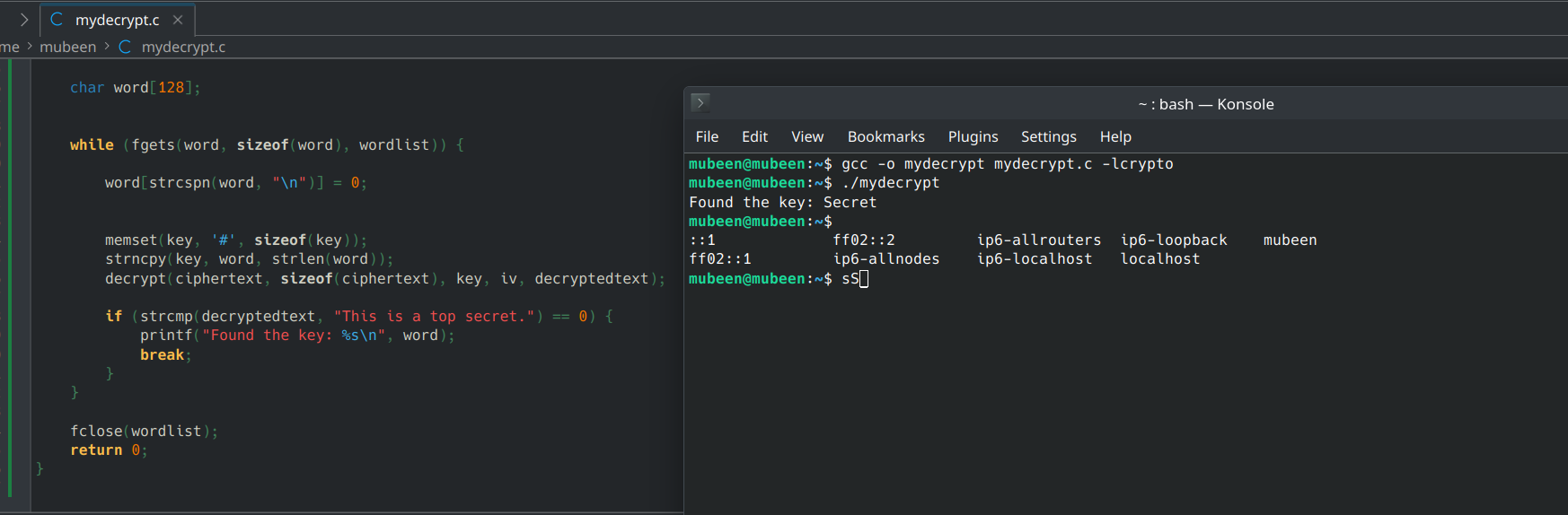
}

}

fclose(wordlist);

return 0;

}

**Compilation and Output:  
**