Jonathan Smith

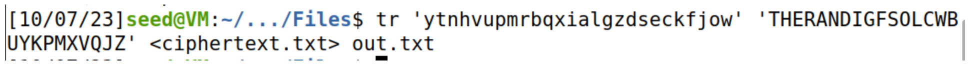
Prof Nhut Nguyen

CS 4393.001

12 October 2023

Secret Key Encryption Lab

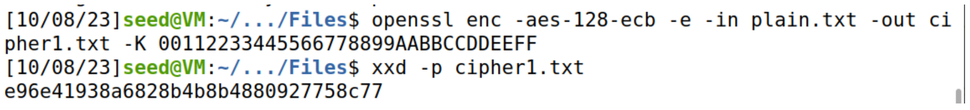
Task 1



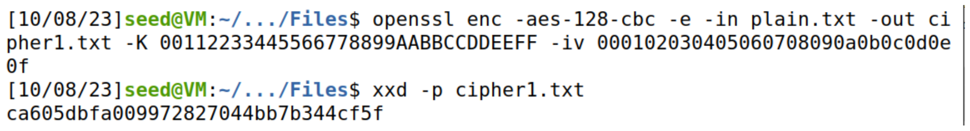
```
[10/07/23]seed@VM:~/.../Files$ tr 'ytnhvupmrbqxialgzdseckfjow' 'THERANDIGFSOLCWB
UYKPMXVQJZ' <ciphertext.txt> out.txt
```

Above is the command I derived by analyzing the frequency analysis on ciphertext.txt. I've included a text file called "Task1.txt" that contains the completed cipher text of ciphertext.txt. According to frequency analysis the most common trigrams in ciphertext.txt are 'ytn' and 'vup,' where they represent 'the' and 'and' respectively, gotten from the trigram Wikipedia page given in the lab. All other subsequent letters afterwards were derived from filling the blanks in words I recognized in the file, like replacing the 'h' in "thehe," as I recognized it should be "there."

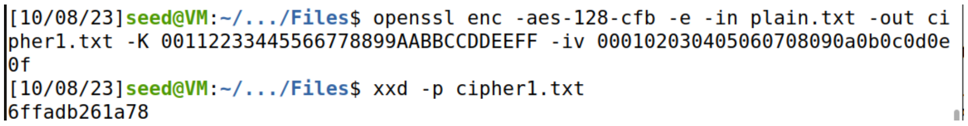
Task 2



```
[10/08/23]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e -in plain.txt -out ci
pher1.txt -K 00112233445566778899AABBCCDDEEFF
[10/08/23]seed@VM:~/.../Files$ xxd -p cipher1.txt
e96e41938a6828b4b8b4880927758c77
```



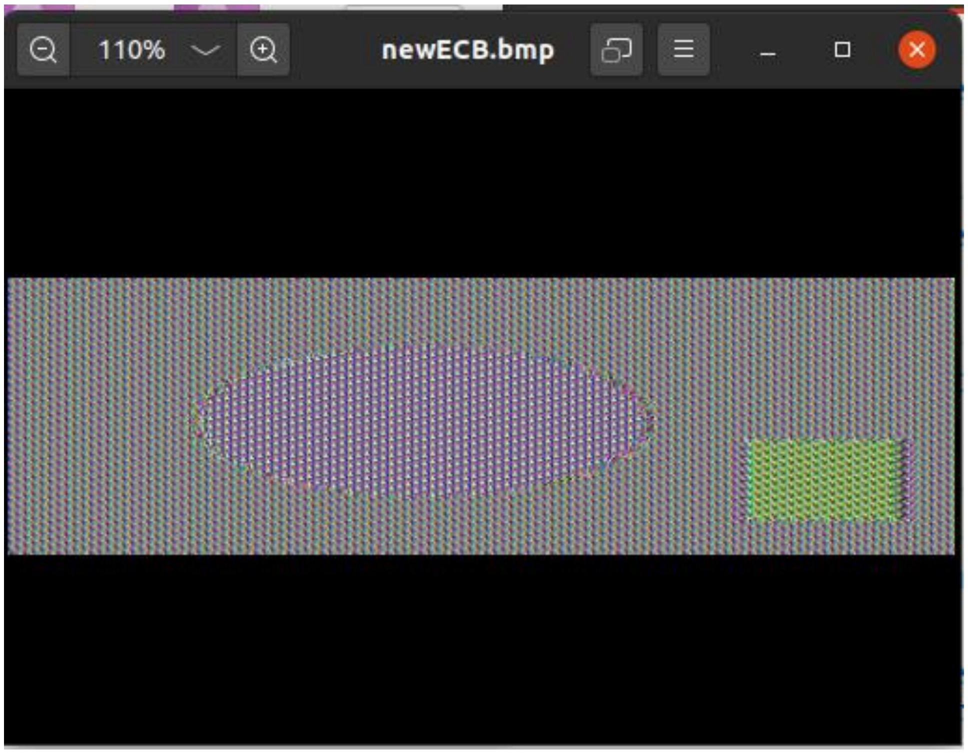
```
[10/08/23]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in plain.txt -out ci
pher1.txt -K 00112233445566778899AABBCCDDEEFF -iv 000102030405060708090a0b0c0d0e
0f
[10/08/23]seed@VM:~/.../Files$ xxd -p cipher1.txt
ca605dbfa009972827044bb7b344cf5f
```

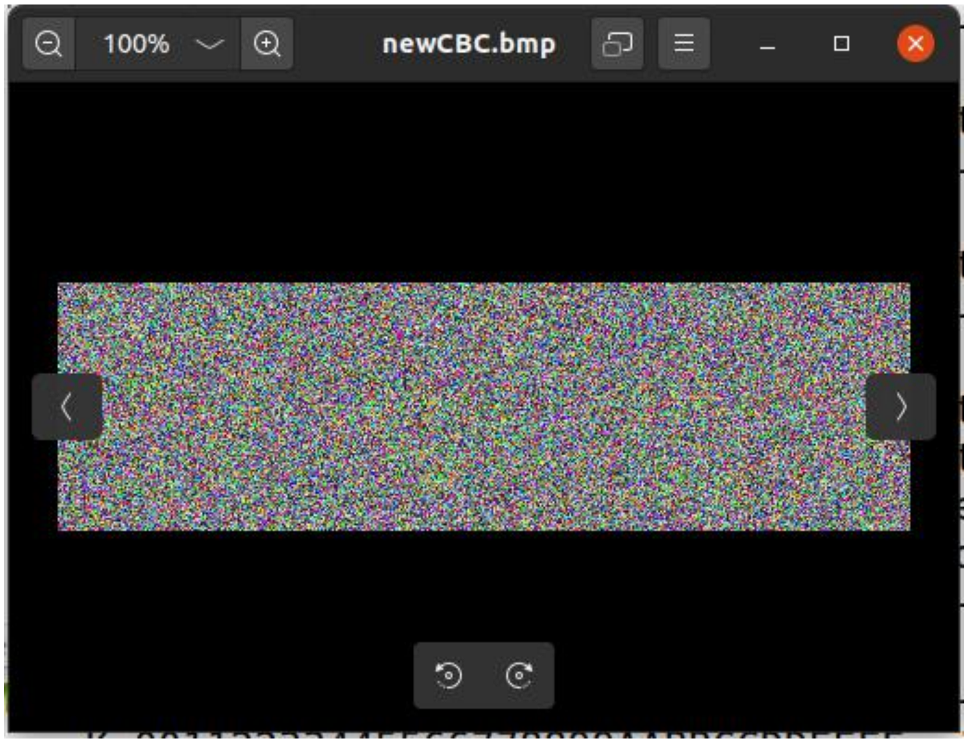


```
[10/08/23]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -e -in plain.txt -out ci
pher1.txt -K 00112233445566778899AABBCCDDEEFF -iv 000102030405060708090a0b0c0d0e
0f
[10/08/23]seed@VM:~/.../Files$ xxd -p cipher1.txt
6ffadb261a78
```

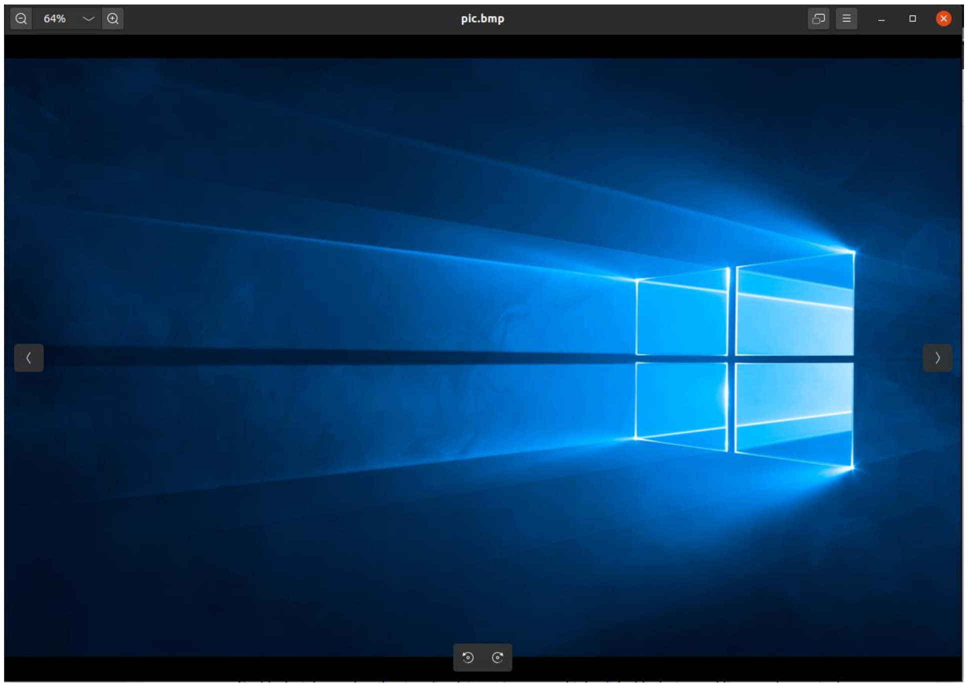
Above shows the basics and experimentation with the openssl command present within Linux. Each picture is a different encryption mode applied to the same text file represented by the flag after -aes-128. The results of each mode of encryption are then printed out to display different ciphertexts showing the difference in strategy with each mode of encryption.

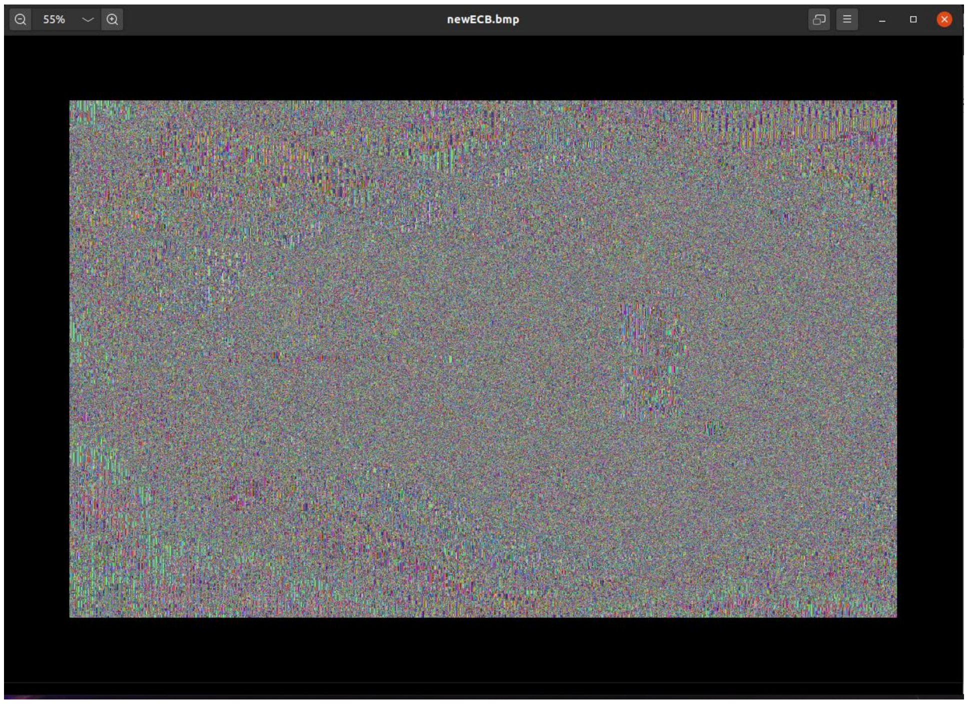
Task 3

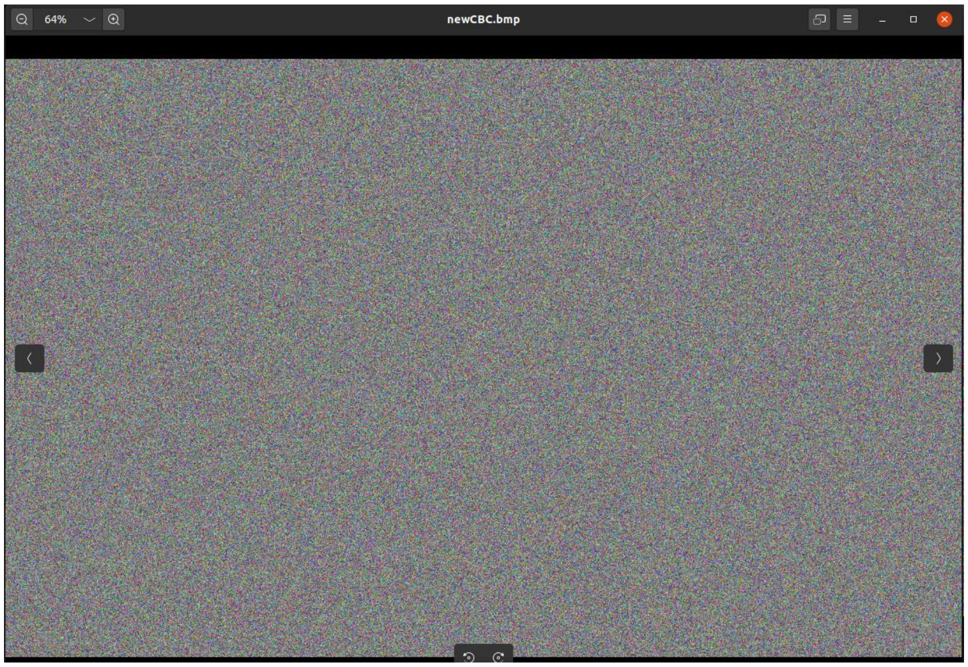




The above two images are the effects of ECB and CBC encryption on the image given in the lab, pic_original.bmp. In this instance, ECB seems to not distort the image comparatively to CBC, which makes sense due to the complexity comparisons between the two modes. A rough outline of the original image can still be seen within the first image, which is a large vulnerability when it comes to encryption.

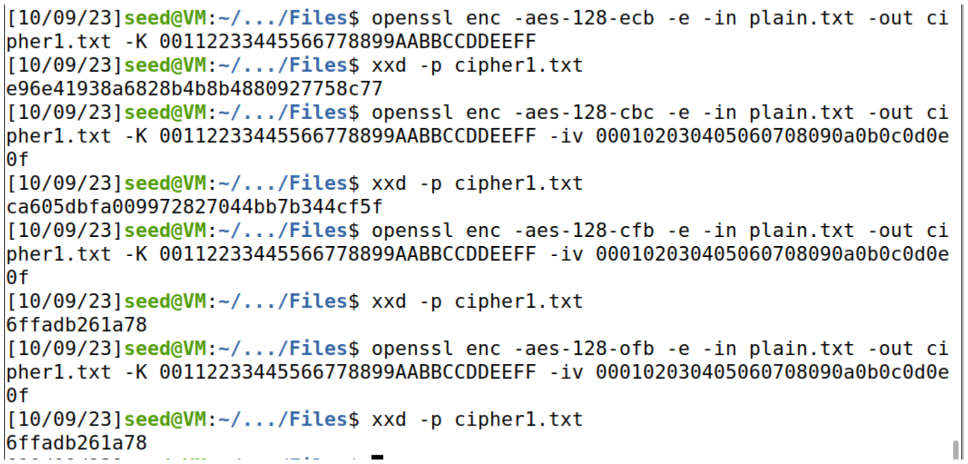

64%
pic.bmp


55%
newECB.bmp



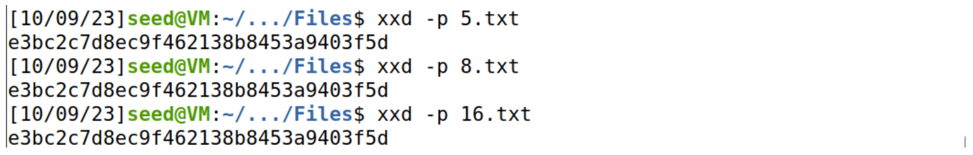
The above images are the same encryptions used in the first part of task 3, ECB and CBC, on an image picked by the user. The first image is the image that was the original image that was used, and the following two images are the results of the ECB and CBC ciphers. Again, rough shapes can still be made out in the first image, reflecting the lack of complexity in ECB compared to the CBC mode of encryption. Although the original image can't be seen in either encryption, CBC provides a more secure encryption as seen in both parts of task 3.

Task 4

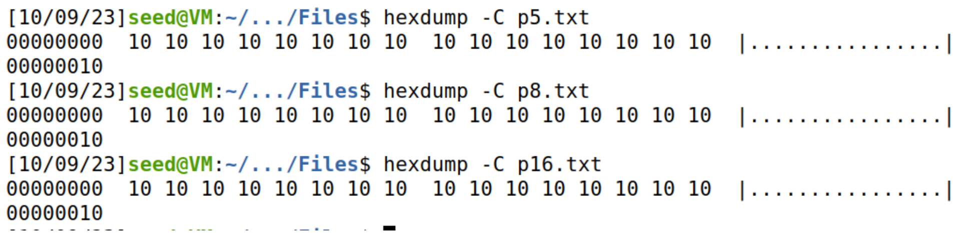


```
[10/09/23]seed@VM:~/.../Files$ openssl enc -aes-128-ecb -e -in plain.txt -out ci
pher1.txt -K 00112233445566778899AABBCCDDEEFF
[10/09/23]seed@VM:~/.../Files$ xxd -p cipher1.txt
e96e41938a6828b4b8b4880927758c77
[10/09/23]seed@VM:~/.../Files$ openssl enc -aes-128-cbc -e -in plain.txt -out ci
pher1.txt -K 00112233445566778899AABBCCDDEEFF -iv 000102030405060708090a0b0c0d0e
0f
[10/09/23]seed@VM:~/.../Files$ xxd -p cipher1.txt
ca605dbfa009972827044bb7b344cf5f
[10/09/23]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -e -in plain.txt -out ci
pher1.txt -K 00112233445566778899AABBCCDDEEFF -iv 000102030405060708090a0b0c0d0e
0f
[10/09/23]seed@VM:~/.../Files$ xxd -p cipher1.txt
6ffadb261a78
[10/09/23]seed@VM:~/.../Files$ openssl enc -aes-128-ofb -e -in plain.txt -out ci
pher1.txt -K 00112233445566778899AABBCCDDEEFF -iv 000102030405060708090a0b0c0d0e
0f
[10/09/23]seed@VM:~/.../Files$ xxd -p cipher1.txt
6ffadb261a78
```

The above picture shows the encryption of the same text file, plain.txt, under different modes of encryption, ECB, CBC, CFB, and OFB. Shown by the picture, modes like ECB and CBC require padding shown by the longer text produced by xxd –p cipher1.txt, as they operate more like block ciphers and require messages to be a certain length before encrypting. Shown by the shorter text from xxd –p cipher1.txt, CFB and OFB do not need padding, as they operate more like stream ciphers, operating bit by but rather than blocks.


```
[10/09/23]seed@VM:~/.../Files$ xxd -p 5.txt
e3bc2c7d8ec9f462138b8453a9403f5d
[10/09/23]seed@VM:~/.../Files$ xxd -p 8.txt
e3bc2c7d8ec9f462138b8453a9403f5d
[10/09/23]seed@VM:~/.../Files$ xxd -p 16.txt
e3bc2c7d8ec9f462138b8453a9403f5d
```

The above image shows the size of the encrypted files. All of them are the same length, 32 bytes, because of padding from the CBC mode encryption requiring messages to be of a certain length to function.


```
[10/09/23]seed@VM:~/.../Files$ hexdump -C p5.txt
00000000  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000010
[10/09/23]seed@VM:~/.../Files$ hexdump -C p8.txt
00000000  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000010
[10/09/23]seed@VM:~/.../Files$ hexdump -C p16.txt
00000000  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000010
```

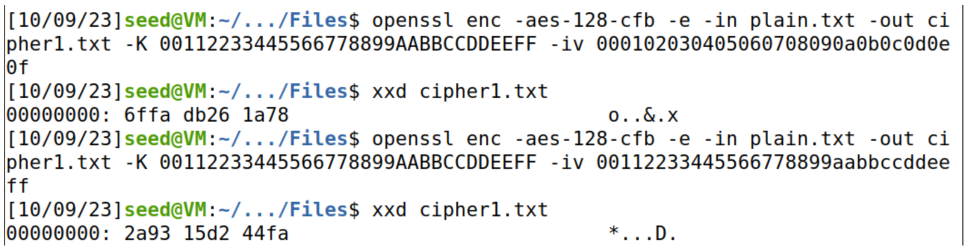
The above image shows the decryption of the previous files with the padding still attached, revealing the padding scheme to be 10.

Task 5

Included separately, the text file, Task5.txt contains the contents of the original one-thousand bit file and the results of encryption modes, EBC, CBC, CFB, and OFB, with a corrupted 5th byte. Looking at the results of corruption with different modes of encryption, you can conclude that depending on the mode of encryption different amounts of code get corrupted.

EBC has 1 block corrupted, CBC has 2 blocks corrupted, CFB has 2 blocks corrupted, and OFB has 1 block corrupted. This is caused by how the mode of encryption works, as both CBC and CFB use the previous block to encrypt the current block, corrupting two blocks with 1 corrupted bit.

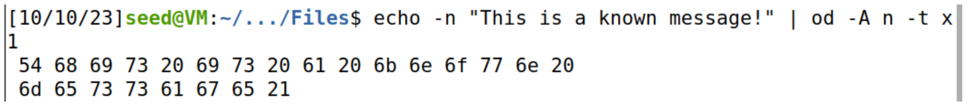
Task 6.1



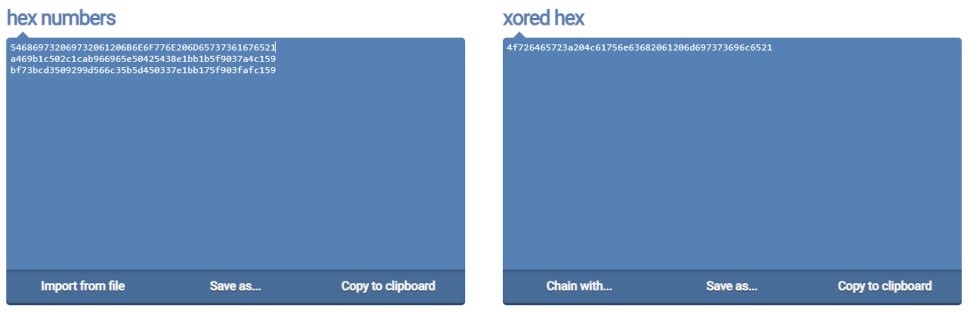
```
[10/09/23]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -e -in plain.txt -out ci
pher1.txt -K 00112233445566778899AABBCCDDEEFF -iv 000102030405060708090a0b0c0d0e
0f
[10/09/23]seed@VM:~/.../Files$ xxd cipher1.txt
00000000: 6ffa db26 1a78                           o..&.x
[10/09/23]seed@VM:~/.../Files$ openssl enc -aes-128-cfb -e -in plain.txt -out ci
pher1.txt -K 00112233445566778899AABBCCDDEEFF -iv 00112233445566778899aabbccddee
ff
[10/09/23]seed@VM:~/.../Files$ xxd cipher1.txt
00000000: 2a93 15d2 44fa                           *...D.
```

Above is the effect of different IV's on a plain text. This shows the importance of uniqueness in IV's as using the same IV would produce the same ciphertext, but different IV's produce different ciphertext, making it harder for attackers to crack.

Task 6.2

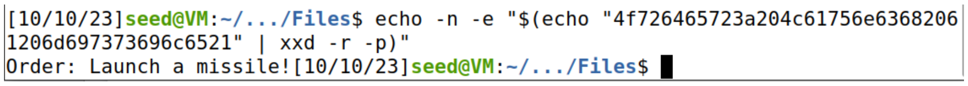


```
[10/10/23]seed@VM:~/.../Files$ echo -n "This is a known message!" | od -A n -t x
1
 54 68 69 73 20 69 73 20 61 20 6b 6e 6f 77 6e 20
 6d 65 73 73 61 67 65 21
```



```
546869732069732061206B6E6F776E206D65737361676521
a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159
bf73bcd3509299d566c35b5d450337e1bb175f903fafc159
```

```
4f726465723a204c61756e63682061206d697373696c6521
```



```
[10/10/23]seed@VM:~/.../Files$ echo -n -e "$(echo "4f726465723a204c61756e6368206
1206d697373696c6521" | xxd -r -p)"
Order: Launch a missile![10/10/23]seed@VM:~/.../Files$ 
```

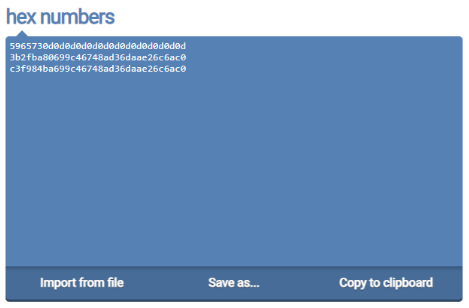
The above images show the process of deciphering what the ciphertext, C2, was saying with the given two plain texts, K1 and P2, and given cipher text, C1. The first image is turning the initial message “This is a known message!” into hexadecimal to be later XOR with C1 and P2 to derive C2 later. This works because of how OFB works deriving

$C1=P1\oplus K1$

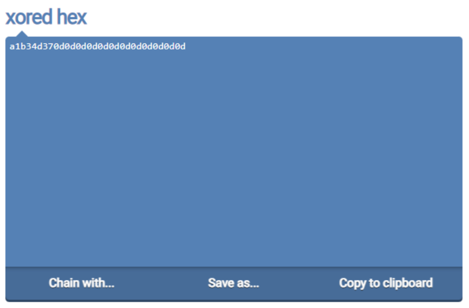
$C2=P2\oplus K2$

With the above equations, the second image shows P1, K1, and K2 being XORed producing C2 with the last step being to convert the hexadecimal back into letter format, so the message can be read, which is depicted in the third image above.

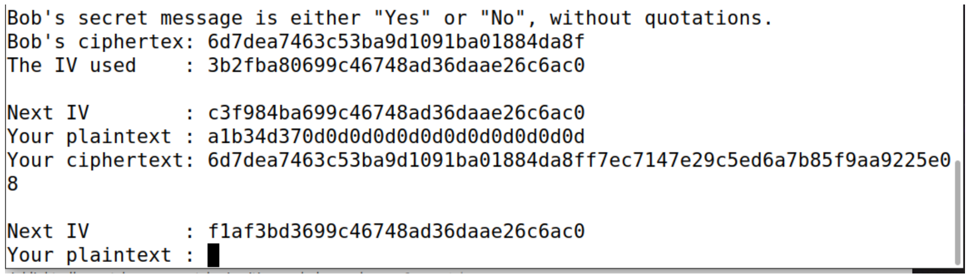
Task 6.3



```
5965730d0d0d0d0d0d0d0d0d0d0d0d0d
3b2fba80699c46748ad36daae26c6ac0
c3f984ba699c46748ad36daae26c6ac0
```



```
a1b34d370d0d0d0d0d0d0d0d0d0d0d0d
```



```
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 6d7dea7463c53ba9d1091ba01884da8f
The IV used     : 3b2fba80699c46748ad36daae26c6ac0

Next IV         : c3f984ba699c46748ad36daae26c6ac0
Your plaintext : a1b34d370d0d0d0d0d0d0d0d0d0d0d0d
Your ciphertext: 6d7dea7463c53ba9d1091ba01884da8ff7ec7147e29c5ed6a7b85f9aa9225e0
8

Next IV         : f1af3bd3699c46748ad36daae26c6ac0
Your plaintext : 
```

Using knowledge accumulated throughout the lab we can confirm Bob’s message is either “Yes” or “No” by feeding “Yes” back to Bob and see if the message that comes back resembles the message that Bob sent and if not, Bob sent No. To find this out you convert “Yes” to hexadecimal, padding it with ‘0d’ until the length reaches 16 bytes. Using similar equations to the ones derived in task 6.2 we can XOR “Yes,” Bob’s IV, and the next IV and feed it back to him to get to get the same ciphertext Bob sent to us along with padding values, which can be seen in the image above.

Task 7

Knowing that the key is a word in the English dictionary, the best way to approach the problem is to brute force it using the file provided, words.txt. You must write a code that iterates through the entire words.txt file, appending # to the ends of the words until they reach 16 characters, skipping words that are over 16 characters. Then you would run the aes-128-cbc cipher to decipher the original plaintext given with key created with the word from the dictionary with # appended to it. Finally, compare the results of the deciphered text form the aes-128-cbc and the original given plaintext. If they match then you have found the key, print it and terminate the program, if they do not match then continue through words.txt until you either reach the end of the list or you find the key that was used to create the given ciphertext with the given plaintext.