Q1 (5 pts): Random J. has been told to design a scheme to prevent messages from being modified by an intruder. Random J. decided to append to each message a hash of that message. Does this solve the problem? Why?

This is not a valid solution since anyone can append a hash to a message. Moreover, an attacker can just modify the original contents of the message and append a new hash to it. Another note, an appended hash does not alert the receiver if the original contents of the message have been modified.

Q2 (5 pts): Suppose Alice, Bob, and Carol want to use secret key technology to authenticate each other. If they all used the same secret key K, then Bob could impersonate Carol to Alicia (actually any of three can impersonate the other to the third). Suppose instead that each had their own secret key, so Alice uses KA, Bob uses KB, and Carol uses KC . This means that each one, to prove his/her identity, responds to a challenge with a function of his/her secret key and the challenge. Is this more secure than having them all use the same secret key K? (Hint: what does Alice need to know in order to verify Carol's answer to Alice's challenge?)

No, it is not more secure. For example, if Alice wants to communicate with Carol, she would need to know Carol's secret key. As a result, Alice can now impersonate herself as Carol as she is communicating to Bob since Alice now has KC and can respond to the challenge using said key. Moreover, this situation still requires all three members to know each other's secret key. Thus, anyone in the group can impersonate each other.

Q3 (5 pts): It is common, for performance reasons, to sign a message digest of a message rather than the message itself. Why is it so important that it be difficult to find two messages with the same message digest?

It is important that no two messages have the same message digest because that allows the attacker to potentially replace m1 with m2 if they have result in the same message digest. As a result, the receiver will not be able to identify if the message has been altered; thus, data integrity is lost.

Q4 (7 pts): Token cards display a number that changes periodically, perhaps every minute. Each such device has a unique secret key. A human can prove possession of a particular such device by entering the displayed number into a computer system. The computer system knows the secret keys of each authorized device. How would you design such a device?

I will design the device to have "calculator" abilities with the unique secret key stored on it. When I enter a key, a special one time password will be generated. I will then enter the password with my user-name to prove I am the owner said device.

Q5 (7 pts): How many DES keys, on the average, encrypt a particular plaintext block to a particular ciphertext block? Please explain.

As we learned in class, DES encrypts a 64 bit plaintext to 64 bit ciphertext using a 56 bit key.

•For a 64 bit plaintext blocks, there are 2^64 possible resultant ciphertext blocks.

•By this logic, each 56 bit key maps 2^64 plaintext blocks to 2^64 ciphertext blocks.

•As a result, each key has a probability of 1/(2^64) in mapping a particular plaintext block to a particular ciphertext block

•As there are 2^56 possible keys, the final resultant is 1/(2^8) as the average.

Q6 (7 pts): Suppose the DES mangler function mapped every 32-bit value to zero, regardless of the value of its input. What function would DES then compute?

•Since the mangler function maps every 32 bit value to zero, the left and right halves will just get swapped without any modifications to their values. This is because any binary string that is xor with a binary string of zeros does not change the value of the first string.

•First there would be the initial permutation of the 64 bit plain text box.

•Then it would be split into of left half and a right half with each half being 32 bits.

•Then with each des round, the left and right halves would be swapped without any modifications to their values.

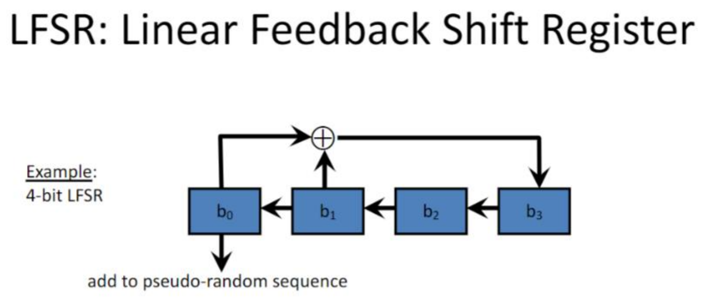
•After the 16 rounds, the final permutation will be created.

•Please note that the final permutation interchanges the consecutive even and odd bits.

Q7 (7 pts): The pseudo-random stream of blocks generated by 64-bit OFB must eventually repeat (since at most 264 different blocks can be generated). Will K {IV} necessarily be the first block to be repeated?

•It will be the first to repeat if the sequence

•Let us reference the LFSR as a general example


LFSR: Linear Feedback Shift Register
Example:
4-bit LFSR
b_0
b_1
b_2
b_3
add to pseudo-random sequence

•As a generic symbol, let bi represent the ith block of encryption.

•We also know that for each bi (where i is 0,1,2….), bi-1 is the decryption of bi and bi is the encryption of bi-1.

•Let ba be the first repeated occurrence of bz where z<a.

•If z = 0 (a.k.a it is the first block in the sequence), then we have satisfied the previous bullet point as z < a and ba==bz.

•If z > 0, then this is a contradiction which will be explained below.

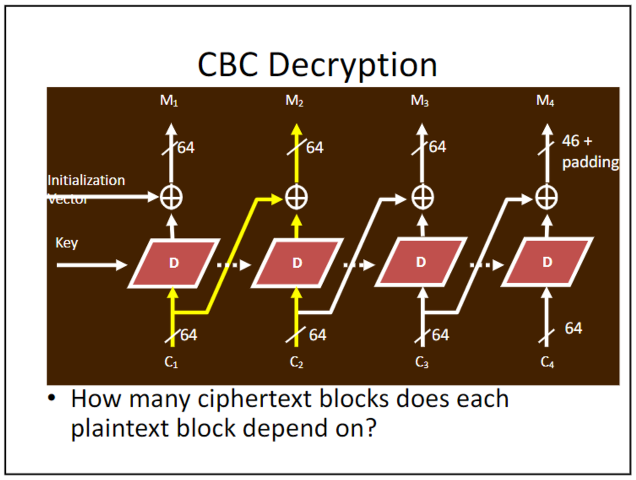
•The previous above statement implies that bz-1 = ba-1 since we have stated that ba==bz.

•By the above statement, we are saying that ba is not the first repeated occurrence of bz.

•Thus ba == b0 with proof by contradiction.

Q8 (7 pts): Consider the following alternative method of encrypting a message. To encrypt a message, use the algorithm for doing a CBC decrypt. To decrypt a message, use the algorithm for doing a CBC encrypt. Would this work? What are the security implications of this, if any, as contrasted with the "normal" CBC?

Yes, this would work. However, there are two security concerns. Since CBC decrypt will be used to encrypt a message, only the XOR portion will be the unpredictable encryption. The next security concern will be explained along with the picture below.



CBC Decryption

- How many ciphertext blocks does each plaintext block depend on?

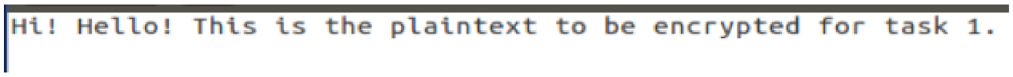
•To restate, we are using CBC decryption to encrypt a message. Thus, when looking at the diagram, just switch the positions of the Cs and Ms

•Note that all the decryption blocks perform the same calculation.

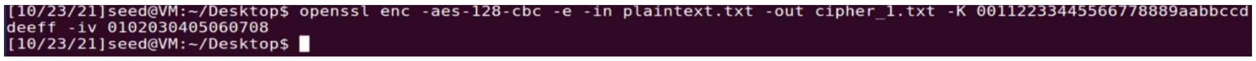
•The security concern is that there will be information leakage.

•If all the Ms contain the same plaintext, the resulting Cs will be exactly the same except for C1 since the D(M1) will be XOR with IV.

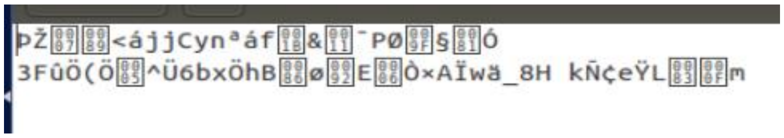
1.1 Task 1: Encryption using different ciphers and modes [10 pts]



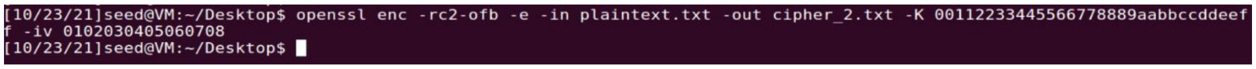
```
Hi! Hello! This is the plaintext to be encrypted for task 1.
```



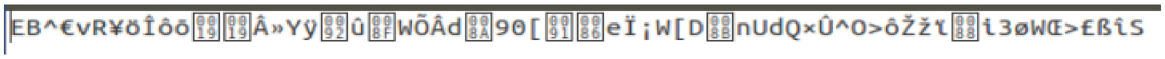
```
[10/23/21]seed@VM:~/Desktop$ openssl enc -aes-128-cbc -e -in plaintext.txt -out cipher_1.txt -K 00112233445566778889aabbccd
deeff -iv 0102030405060708
[10/23/21]seed@VM:~/Desktop$ █
```



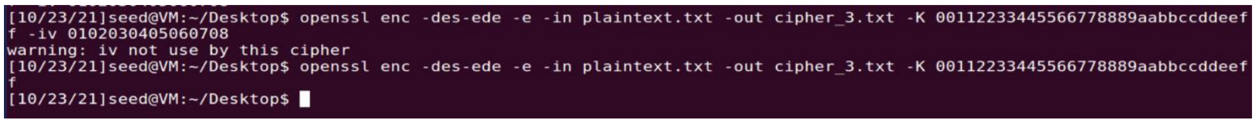
```
þŽ[0087][0089]<ájjCynªáf[001B]&[0011]¯PØ[009F]§[0081]Ó
3FûÖ(Ö[0085]^Ü6bxÖhB[0086]ø[0092]E[0086]Ò×AÏwă_8H kÑçeŸL[0083][008F]m
```



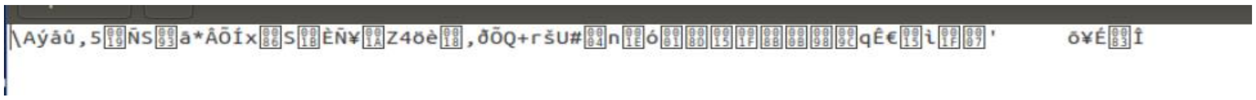
```
[10/23/21]seed@VM:~/Desktop$ openssl enc -rc2-ofb -e -in plaintext.txt -out cipher_2.txt -K 00112233445566778889aabbccddeef
f -iv 0102030405060708
[10/23/21]seed@VM:~/Desktop$ █
```



```
EB^€vR¥öÎôõ[0019][0019]Â»Yÿ[0092]û[008F]WÕÂd[008A]90[[0091][0086]eÏ¡W[D[0088]nUdQ×Û^O>ôŽžť[0088]ĭ3øWŒ>£ßĭS
```



```
[10/23/21]seed@VM:~/Desktop$ openssl enc -des-ede -e -in plaintext.txt -out cipher_3.txt -K 00112233445566778889aabbccddeef
f -iv 0102030405060708
warning: iv not use by this cipher
[10/23/21]seed@VM:~/Desktop$ openssl enc -des-ede -e -in plaintext.txt -out cipher_3.txt -K 00112233445566778889aabbccddeef
f
[10/23/21]seed@VM:~/Desktop$ █
```



```
\Aýâû,5[0019]ÑS[0093]ā*ÂÕÍx[0086]S[001B]ÈÑ¥[001A]Z4öè[001B],ðÕQ+ršU#[0084]n[001E]ó[0081][008D][0015][001F][0088][0088][0098][009C]qÊ€[0015]ì[001F][0087]'      õ¥É[0083]Î
```

I have encrypted the plaintext (first image) in three different encryptions with different modes:

- aes-128-cbc
- rc2-ofb
- des-ede

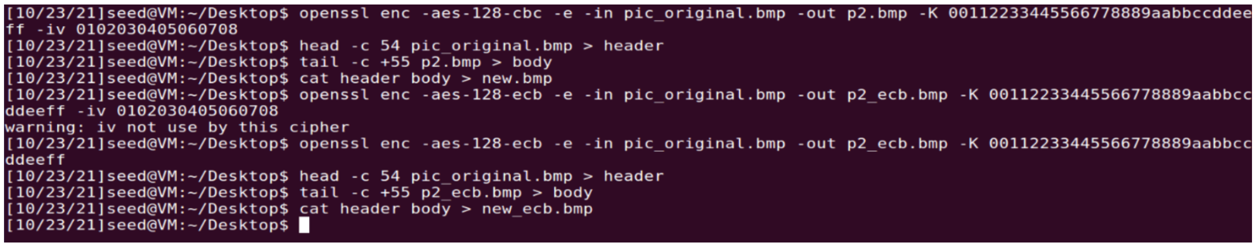
Also, the key and initialization vector used for the encryptions are:

- key = 00112233445566778889aabbccddeeff
- initialization vector = 0102030405060708

As expected, the plaintext has been completely scrambled in all three resultant ciphers. Moreover, all three ciphers use a lot of foreign characters that are not common in the English language as well. I have also noted some interesting results from each of the ciphers.

- The aes cipher in cbc mode produces two lines of cipher text.
- The rc2 cipher in ofb mode cipher produces one line of cipher text.
- The des cipher in ede mode produces one line of cipher text, but there is a noticeable whitespace near the end of the line.

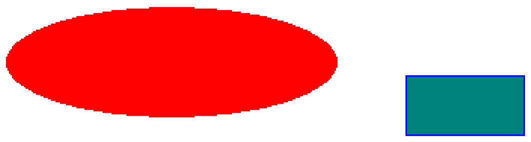
Task 2: Encryption Mode – ECB vs. CBC [10 pts]

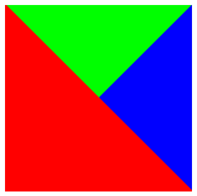


```
[10/23/21]seed@VM:~/Desktop$ openssl enc -aes-128-cbc -e -in pic_original.bmp -out p2.bmp -K 00112233445566778889aabbccddee
ff -iv 0102030405060708
[10/23/21]seed@VM:~/Desktop$ head -c 54 pic_original.bmp > header
[10/23/21]seed@VM:~/Desktop$ tail -c +55 p2.bmp > body
[10/23/21]seed@VM:~/Desktop$ cat header body > new.bmp
[10/23/21]seed@VM:~/Desktop$ openssl enc -aes-128-ecb -e -in pic_original.bmp -out p2_ecb.bmp -K 00112233445566778889aabbcc
ddeeff -iv 0102030405060708
warning: iv not use by this cipher
[10/23/21]seed@VM:~/Desktop$ openssl enc -aes-128-ecb -e -in pic_original.bmp -out p2_ecb.bmp -K 00112233445566778889aabbcc
ddeeff
[10/23/21]seed@VM:~/Desktop$ head -c 54 pic_original.bmp > header
[10/23/21]seed@VM:~/Desktop$ tail -c +55 p2_ecb.bmp > body
[10/23/21]seed@VM:~/Desktop$ cat header body > new_ecb.bmp
[10/23/21]seed@VM:~/Desktop$ 
```

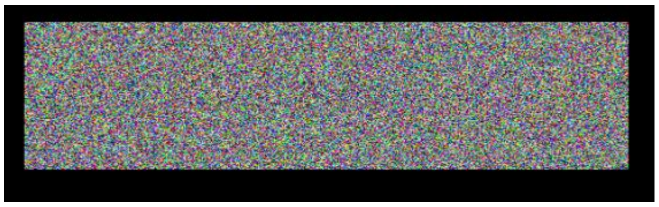
In this task, I encrypted two bmp files in aes-128-cbc and aes-128-ecb. The pictures to be encrypted are shown below.

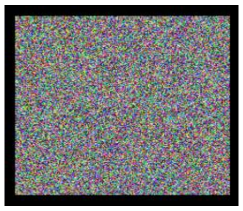
Note: I ran the same commands for the second bmp file as well.



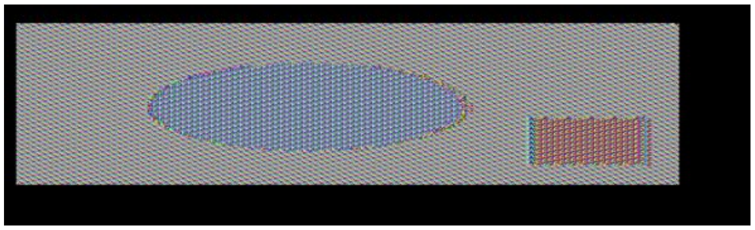


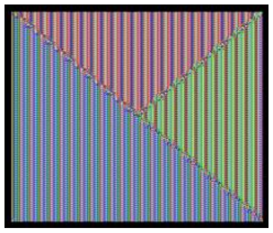
In the following image below, this is the result of the aes-128-cbc encryption.





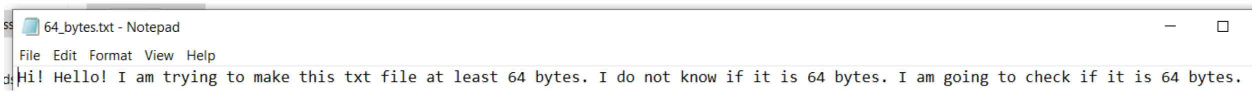
As can be shown in the above image, the picture is completely unrecognizable. There is no way to visually make out the first image from this image. However, this is completely different when it comes to the aes-128-ebc encryption, which is shown below.





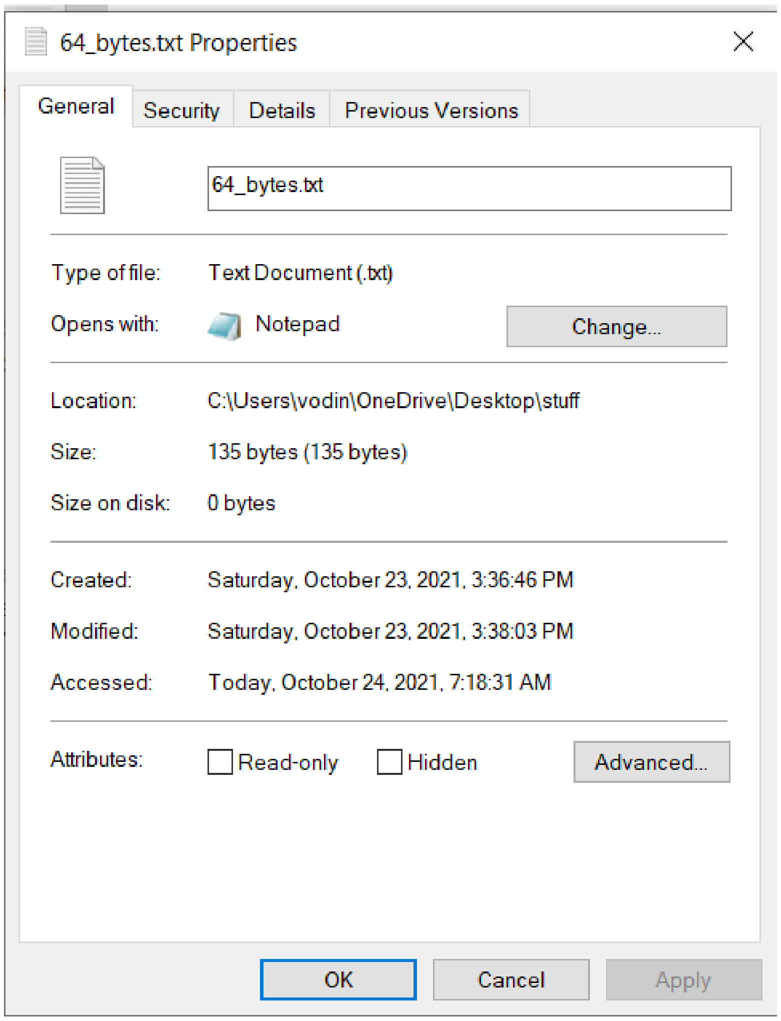
As can be shown in the above image, the first image is scrambled but not as much as the aes-128-cbc. In the following aes-128-ebc encryption image, it is still possible to visually make out the first image from this image. The silhouettes of the shapes are still there, and the colors of the ellipse and rectangle have been swapped. The colors of the triangles inside the square have been swapped as well.

Task 3: Encryption Mode – Corrupted Cipher Text [10 pts]



```
64_bytes.txt - Notepad
File  Edit  Format  View  Help
Hi! Hello! I am trying to make this txt file at least 64 bytes. I do not know if it is 64 bytes. I am going to check if it is 64 bytes.
```

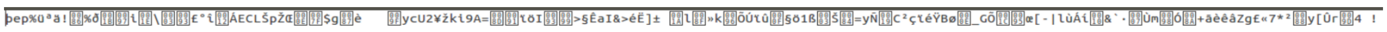
In this task, I created the following txt file that is at least 64 bytes. In order to ensure that it is at least 64 bytes, I checked the properties of the file.



```
64_bytes.txt Properties
General | Security | Details | Previous Versions

64_bytes.txt

Type of file:   Text Document (.txt)
Opens with:     Notepad                       Change...

Location:       C:\Users\vodin\OneDrive\Desktop\stuff
Size:           135 bytes (135 bytes)
Size on disk:   0 bytes

Created:        Saturday, October 23, 2021, 3:36:46 PM
Modified:       Saturday, October 23, 2021, 3:38:03 PM
Accessed:       Today, October 24, 2021, 7:18:31 AM

Attributes:     [ ] Read-only   [ ] Hidden    Advanced...

                 OK        Cancel        Apply
```

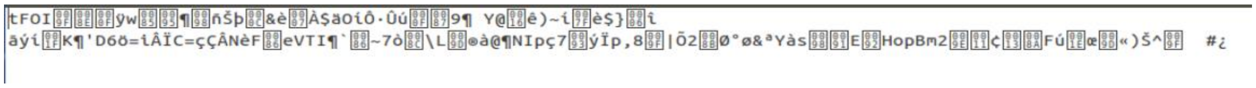
I then encrypted the text file using the aes cipher in the following modes:

- aes-128-ecb



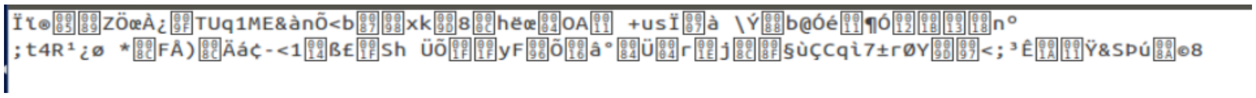
```
þep%ûªä!□%ð□□ĭ□\□□£°î□ÁECLŠpŽŒ□□$g□è    □ycU2¥žki9A=□□ŧöI□□>§ÊaI&>éË]± □l□»k□ÕÚŧû□§ö1ß□Š□=yÑ□C²çŧéŸBø□_GÕ□□œ[-|lùÁí□&`·□Ùm□Ó□+ãèêâZg£«7*²□y[Ûr□4 !
```

- aes-128-cbc



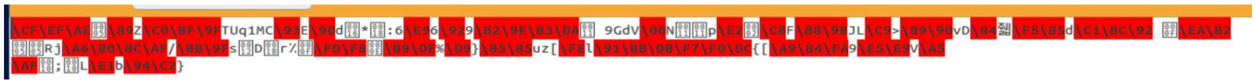
```
ŧFOI□□□ÿw□□¶□ńŠþ□&è□À$äOíÔ·Ûú□□9¶ Y@□ê)~í□è$}□î
ãýí□K¶'D6ő=íÂÏC=çÇÂNèF□eVTI¶`□~7ò□\L□®à@¶NIpç7□ýÏp,8□|Õ2□Ø°ø&ªYàs□□E□HopBm2□□¢□□Fú□œ□«)Š^□   #¿
```

- aes-128-ofb



```
Ïť®ZÖœÀ¿TUq1ME&ànÕ<bxk8hëœOA +usÏà \Ýb@Óé¶Ón°
;t4R¹¿ø *FÅ)Ääç-<1ß£Sh ÜÕyFÕâ°ÜrjşùÇCqì7±rØY<;³ÊŸ&SÞú©8
```

- aes-128-cfb



```
\CF\EF\AE\89Z\C0\BF\9FTUq1MC\93E\9Dd*:6\E96\929\B2\9E\B3\DA 9GdV\00NpE2\C8F\88\9BJL\C9>\89\90vD\84\F5\85d\C1\BC\92 \EA\B2
Rj\A6\B0\8C\AF/\BB\9Fs Dr%\F0\F8\B9\DE%\D9}\85\85uz[\FEl\91\BB\DB\F7\F0\DC{[\A9\84\FA9\E5\E9V\A5
\AF;L\E1b\94\C2}
```

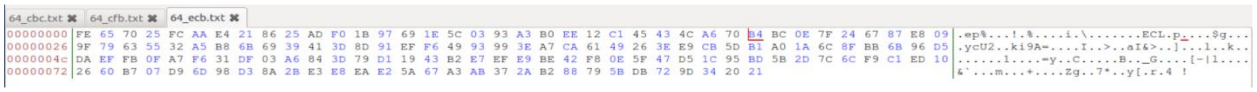
Note: For this image, there are highlights because linux states the following encryption is using unrecognizable characters.

I then proceeded to type bless in the terminal to open up the hex editor. I then opened each of the encrypted files and change the 30th byte. Here is a list of before and after changing the 30th byte.

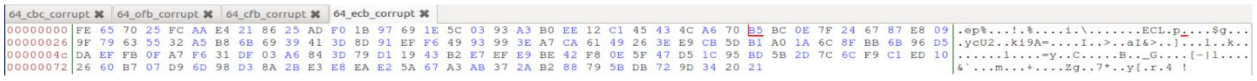
Note: The first image is the "before" and the second image is the "after".

Note: The red square highlights the 30th byte.

- aes-128-ecb

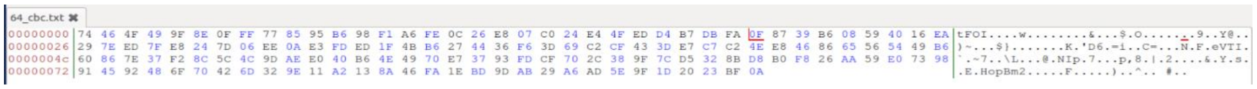


```
00000000 FE 65 70 25 FC AA E4 21 86 25 AD F0 1B 97 69 1E 5C 03 93 A3 B0 EE 12 C1 45 43 4C A6 70 B4 BC 0E 7F 24 67 87 E8 09 .ep%...!.%....i.\.......ECL.p....$g...
00000026 9F 79 63 55 32 A5 B8 6B 69 39 41 3D 8D 91 EF F6 49 93 99 3E A7 CA 61 49 26 3E E9 CB 5D B1 A0 1A 6C 8F BB 6B 96 D5 .ycU2..ki9A=....I..>..aI&>..]...l..k..
0000004c DA EF FB 0F A7 F6 31 DF 03 A6 84 3D 79 D1 19 43 B2 E7 EF E9 BE 42 F8 0E 5F 47 D5 1C 95 BD 5B 2D 7C 6C F9 C1 ED 10 ......1....=y..C.....B.._G....[-|l....
00000072 26 60 B7 07 D9 6D 98 D3 8A 2B E3 E8 EA E2 5A 67 A3 AB 37 2A B2 88 79 5B DB 72 9D 34 20 21                         &`...m...+....Zg..7*..y[.r.4 !
```

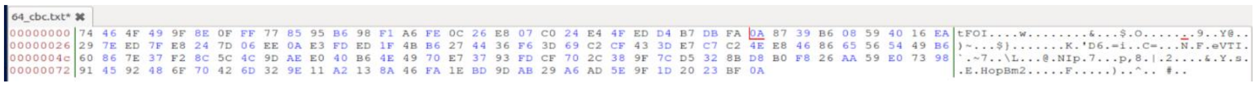


```
00000000 FE 65 70 25 FC AA E4 21 86 25 AD F0 1B 97 69 1E 5C 03 93 A3 B0 EE 12 C1 45 43 4C A6 70 B5 BC 0E 7F 24 67 87 E8 09 .ep%...!.%....i.\.......ECL.p....$g...
00000026 9F 79 63 55 32 A5 B8 6B 69 39 41 3D 8D 91 EF F6 49 93 99 3E A7 CA 61 49 26 3E E9 CB 5D B1 A0 1A 6C 8F BB 6B 96 D5 .ycU2..ki9A=....I..>..aI&>..]...l..k..
0000004c DA EF FB 0F A7 F6 31 DF 03 A6 84 3D 79 D1 19 43 B2 E7 EF E9 BE 42 F8 0E 5F 47 D5 1C 95 BD 5B 2D 7C 6C F9 C1 ED 10 ......1....=y..C.....B.._G....[-|l....
00000072 26 60 B7 07 D9 6D 98 D3 8A 2B E3 E8 EA E2 5A 67 A3 AB 37 2A B2 88 79 5B DB 72 9D 34 20 21                         &`...m...+....Zg..7*..y[.r.4 !
```

- aes-128-cbc

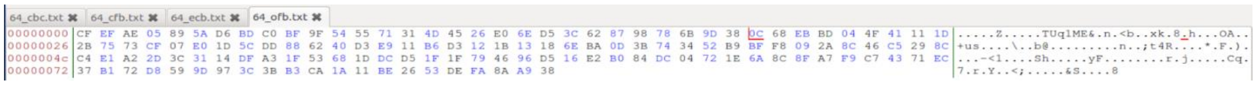


```
00000000 74 46 4F 49 9F 8E 0F FF 77 85 95 B6 98 F1 A6 FE 0C 26 E8 07 C0 24 E4 4F ED D4 B7 DB FA 0F 87 39 B6 08 59 40 16 EA tFOI....w........&...$.O.........9..Y@..
00000026 29 7E ED 7F E8 24 7D 06 EE 0A E3 FD ED 1F 4B B6 27 44 36 F6 3D 69 C2 CF 43 3D E7 C7 C2 4E E8 46 86 65 56 54 49 B6 )~...$}.......K.'D6.=i..C=...N.F.eVTI.
0000004c 60 86 7E 37 F2 8C 5C 4C 9D AE E0 40 B6 4E 49 70 E7 37 93 FD CF 70 2C 38 9F 7C D5 32 8B D8 B0 F8 26 AA 59 E0 73 98 `.~7..\L...@.NIp.7...p,8.|.2....&.Y.s.
00000072 91 45 92 48 6F 70 42 6D 32 9E 11 A2 13 8A 46 FA 1E BD 9D AB 29 A6 AD 5E 9F 1D 20 23 BF 0A                         .E.HopBm2.....F.....)..^.. #..
```

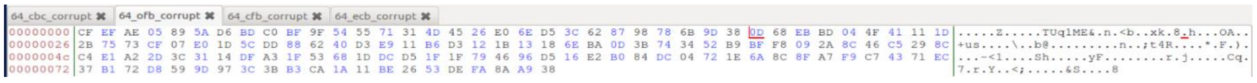


```
00000000 74 46 4F 49 9F 8E 0F FF 77 85 95 B6 98 F1 A6 FE 0C 26 E8 07 C0 24 E4 4F ED D4 B7 DB FA 0A 87 39 B6 08 59 40 16 EA tFOI....w........&...$.O.........9..Y@..
00000026 29 7E ED 7F E8 24 7D 06 EE 0A E3 FD ED 1F 4B B6 27 44 36 F6 3D 69 C2 CF 43 3D E7 C7 C2 4E E8 46 86 65 56 54 49 B6 )~...$}.......K.'D6.=i..C=...N.F.eVTI.
0000004c 60 86 7E 37 F2 8C 5C 4C 9D AE E0 40 B6 4E 49 70 E7 37 93 FD CF 70 2C 38 9F 7C D5 32 8B D8 B0 F8 26 AA 59 E0 73 98 `.~7..\L...@.NIp.7...p,8.|.2....&.Y.s.
00000072 91 45 92 48 6F 70 42 6D 32 9E 11 A2 13 8A 46 FA 1E BD 9D AB 29 A6 AD 5E 9F 1D 20 23 BF 0A                         .E.HopBm2.....F.....)..^.. #..
```

- aes-128-ofb

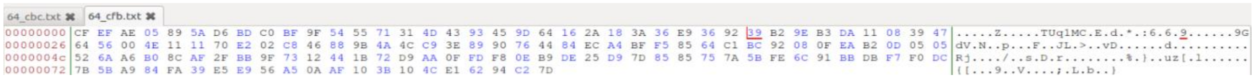


```
00000000 CF EF AE 05 89 5A D6 BD C0 BF 9F 54 55 71 31 4D 45 26 E0 6E D5 3C 62 87 98 78 6B 9D 38 0C 68 EB BD 04 4F 41 11 1D .....Z.....TUq1ME&.n.<b..xk.8.h...OA..
00000026 2B 75 73 CF 07 E0 1D 5C DD 88 62 40 D3 E9 11 B6 D3 12 1B 13 18 6E BA 0D 3B 74 34 52 B9 BF F8 09 2A 8C 46 C5 29 8C +us....\..b@.........n..;t4R....*.F.).
0000004c C4 E1 A2 2D 3C 31 14 DF A3 1F 53 68 1D DC D5 1F 1F 79 46 96 D5 16 E2 B0 84 DC 04 72 1E 6A 8C 8F A7 F9 C7 43 71 EC ...-<1....Sh.....yF........r.j.....Cq.
00000072 37 B1 72 D8 59 9D 97 3C 3B B3 CA 1A 11 BE 26 53 DE FA 8A A9 38                                                       7.r.Y..<;.....&S....8
```

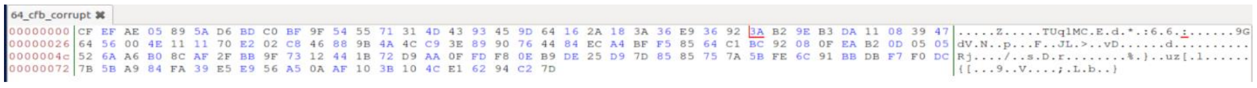


```
00000000 CF EF AE 05 89 5A D6 BD C0 BF 9F 54 55 71 31 4D 45 26 E0 6E D5 3C 62 87 98 78 6B 9D 38 0D 68 EB BD 04 4F 41 11 1D .....Z.....TUq1ME&.n.<b..xk.8.h...OA..
00000026 2B 75 73 CF 07 E0 1D 5C DD 88 62 40 D3 E9 11 B6 D3 12 1B 13 18 6E BA 0D 3B 74 34 52 B9 BF F8 09 2A 8C 46 C5 29 8C +us....\..b@.........n..;t4R....*.F.).
0000004c C4 E1 A2 2D 3C 31 14 DF A3 1F 53 68 1D DC D5 1F 1F 79 46 96 D5 16 E2 B0 84 DC 04 72 1E 6A 8C 8F A7 F9 C7 43 71 EC ...-<1....Sh.....yF........r.j.....Cq.
00000072 37 B1 72 D8 59 9D 97 3C 3B B3 CA 1A 11 BE 26 53 DE FA 8A A9 38                                                       7.r.Y..<;.....&S....8
```

- aes-128-cfb



```
00000000 CF EF AE 05 89 5A D6 BD C0 BF 9F 54 55 71 31 4D 43 93 45 9D 64 16 2A 18 3A 36 E9 36 92 39 B2 9E B3 DA 11 08 39 47 .....Z.....TUq1MC.E.d.*.:6.6.9......9G
00000026 64 56 00 4E 11 11 70 E2 02 C8 46 88 9B 4A 4C C9 3E 89 90 76 44 84 EC A4 BF F5 85 64 C1 BC 92 08 0F EA B2 0D 05 05 dV.N..p...F..JL.>..vD......d.........
0000004c 52 6A A6 B0 8C AF 2F BB 9F 73 12 44 1B 72 D9 AA 0F FD F8 0E B9 DE 25 D9 7D 85 85 75 7A 5B FE 6C 91 BB DB F7 F0 DC Rj..../..s.D.r........%.}..uz[.l......
00000072 7B 5B A9 84 FA 39 E5 E9 56 A5 0A AF 10 3B 10 4C E1 62 94 C2 7D                                                       {[...9..V....;.L.b..}
```



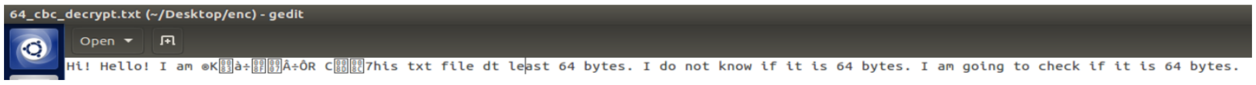
```
00000000 CF EF AE 05 89 5A D6 BD C0 BF 9F 54 55 71 31 4D 43 93 45 9D 64 16 2A 18 3A 36 E9 36 92 3A B2 9E B3 DA 11 08 39 47 .....Z.....TUq1MC.E.d.*.:6.6.:......9G
00000026 64 56 00 4E 11 11 70 E2 02 C8 46 88 9B 4A 4C C9 3E 89 90 76 44 84 EC A4 BF F5 85 64 C1 BC 92 08 0F EA B2 0D 05 05 dV.N..p...F..JL.>..vD......d.........
0000004c 52 6A A6 B0 8C AF 2F BB 9F 73 12 44 1B 72 D9 AA 0F FD F8 0E B9 DE 25 D9 7D 85 85 75 7A 5B FE 6C 91 BB DB F7 F0 DC Rj..../..s.D.r........%.}..uz[.l......
00000072 7B 5B A9 84 FA 39 E5 E9 56 A5 0A AF 10 3B 10 4C E1 62 94 C2 7D                                                       {[...9..V....;.L.b..}
```

Before decrypting the corrupted files with the same key and initialization vector, I will answer the following question below.

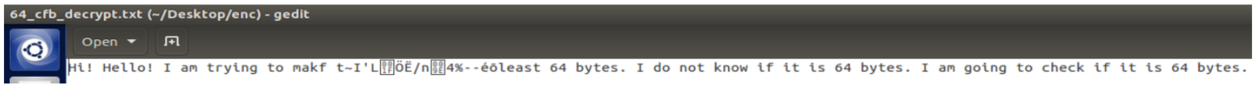
How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively?

- CBC: I should be able to recover all information except for 2 blocks.
- CFB: I should be able to recover all information except for 2 blocks.
- ECB: I should be able to recover all information except for 1 block.
- OFB: I should be able to recover all information except for 1 block.

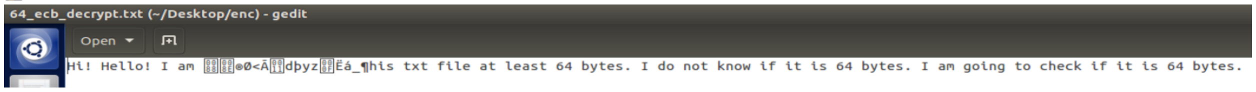
I then proceeded to decrypt all of the corrupted files (in the same order as above).



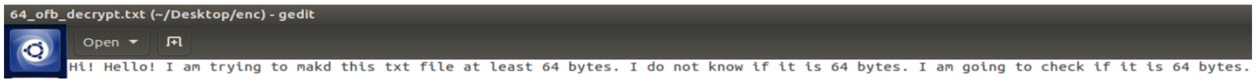
```
Hi! Hello! I am ⊛K▯▯à÷▯▯▯▯Â÷ÔR C▯▯▯▯7his txt file dt least 64 bytes. I do not know if it is 64 bytes. I am going to check if it is 64 bytes.
```



```
Hi! Hello! I am trying to makf t~I'L▯ÖË/n▯4%--éõleast 64 bytes. I do not know if it is 64 bytes. I am going to check if it is 64 bytes.
```



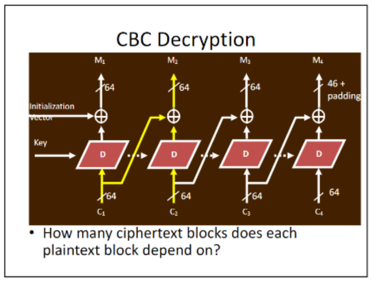
```
Hi! Hello! I am ▯▯▯▯⊛Ø<Ã▯dþyz▯Ëá_¶his txt file at least 64 bytes. I do not know if it is 64 bytes. I am going to check if it is 64 bytes.
```



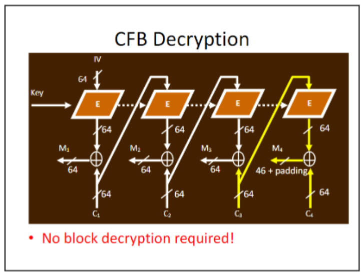
```
Hi! Hello! I am trying to makd this txt file at least 64 bytes. I do not know if it is 64 bytes. I am going to check if it is 64 bytes.
```

As can be seen, my deductions are correct and the answers will be provided below.

- CBC and CFB: As shown in the images below, the decryption of a cipher block is dependent on the cipher block that is adjacent to it.

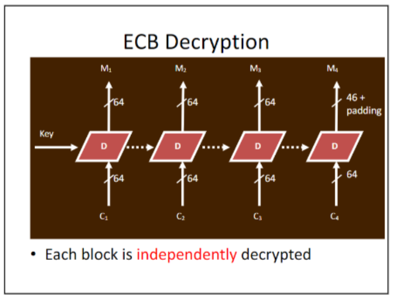


- How many ciphertext blocks does each plaintext block depend on?

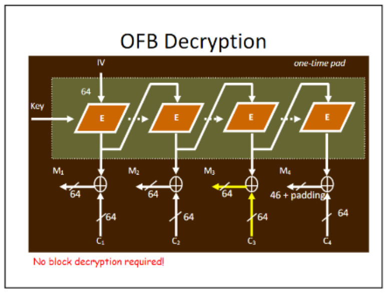


- No block decryption required!

- ECB and OFB: As shown in the images below, the decryption of a cipher block is independent of all other cipher blocks.



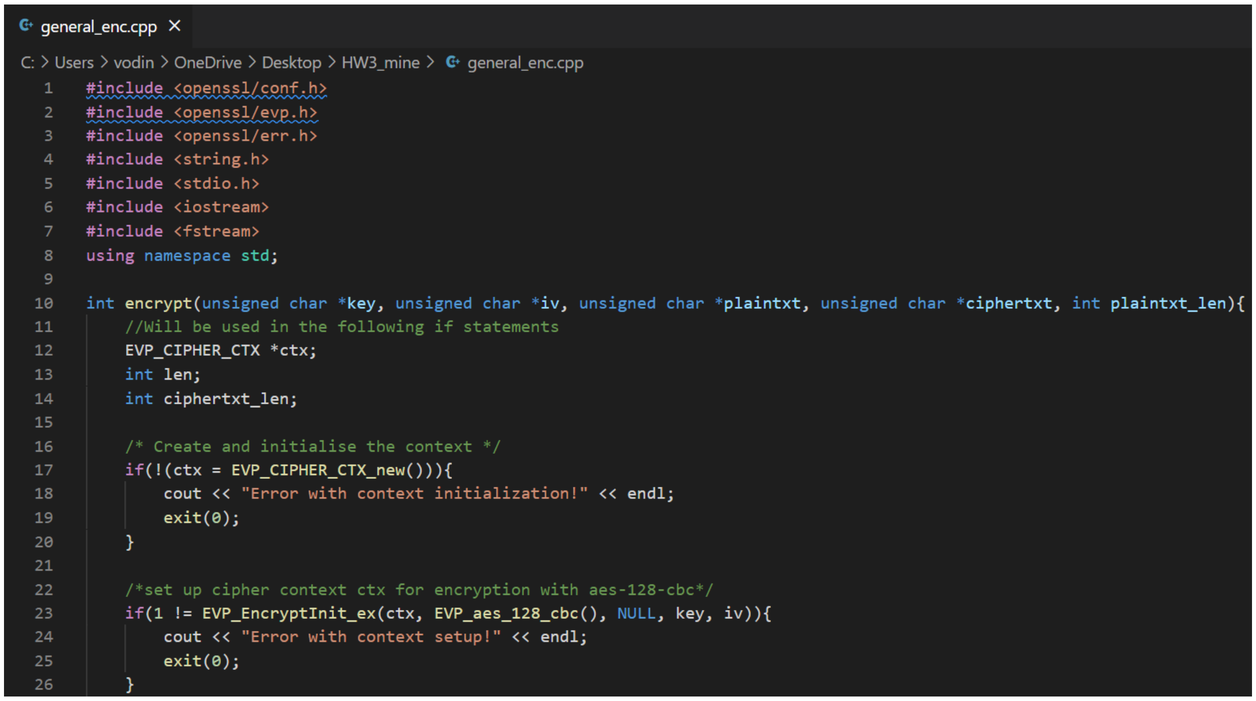
- Each block is independently decrypted



No block decryption required!

Based upon the implications of these differences, it is easy to predict the changes.

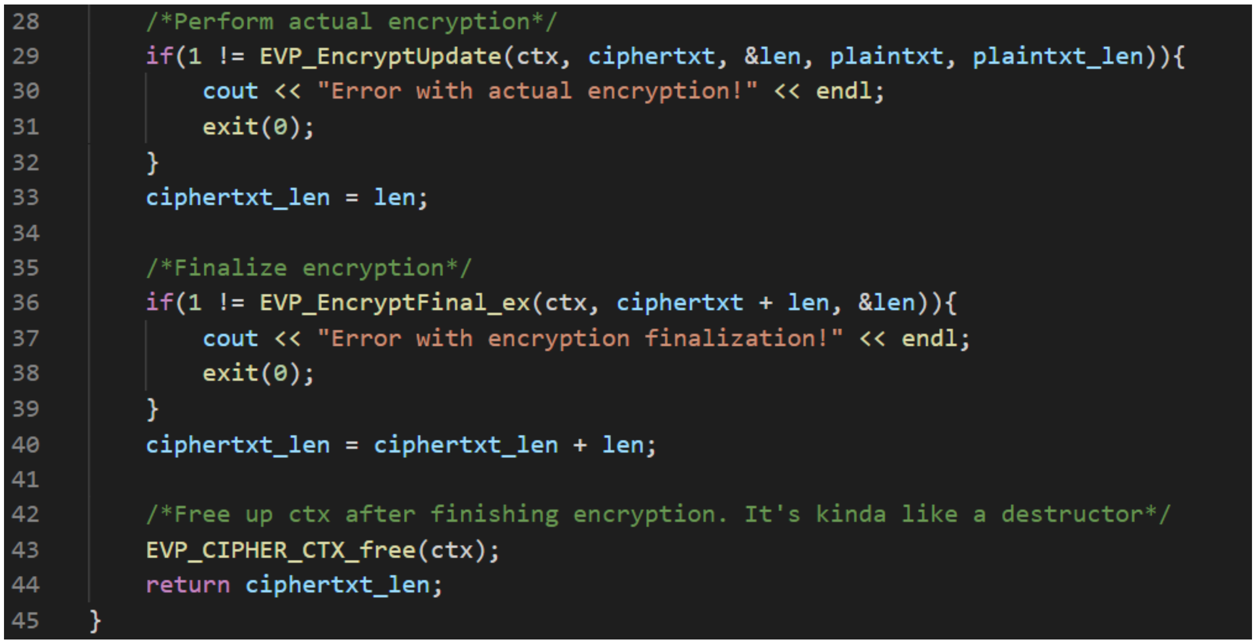
Task 4: Programming using the Crypto Library [20 pts]



general_enc.cpp ×

C: > Users > vodin > OneDrive > Desktop > HW3_mine > general_enc.cpp

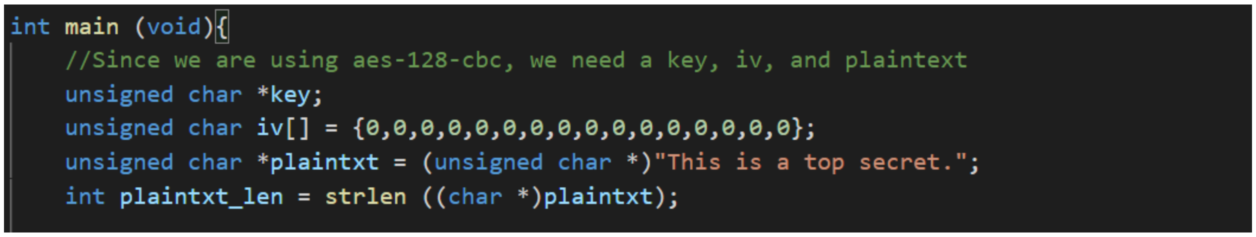
```cpp
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <string.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
using namespace std;

int encrypt(unsigned char *key, unsigned char *iv, unsigned char *plaintxt, unsigned char *ciphertxt, int plaintxt_len){
    //Will be used in the following if statements
    EVP_CIPHER_CTX *ctx;
    int len;
    int ciphertxt_len;

    /* Create and initialise the context */
    if(!(ctx = EVP_CIPHER_CTX_new())){
        cout << "Error with context initialization!" << endl;
        exit(0);
    }

    /*set up cipher context ctx for encryption with aes-128-cbc*/
    if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv)){
        cout << "Error with context setup!" << endl;
        exit(0);
    }
```



```cpp
    /*Perform actual encryption*/
    if(1 != EVP_EncryptUpdate(ctx, ciphertxt, &len, plaintxt, plaintxt_len)){
        cout << "Error with actual encryption!" << endl;
        exit(0);
    }
    ciphertxt_len = len;

    /*Finalize encryption*/
    if(1 != EVP_EncryptFinal_ex(ctx, ciphertxt + len, &len)){
        cout << "Error with encryption finalization!" << endl;
        exit(0);
    }
    ciphertxt_len = ciphertxt_len + len;

    /*Free up ctx after finishing encryption. It's kinda like a destructor*/
    EVP_CIPHER_CTX_free(ctx);
    return ciphertxt_len;
}
```

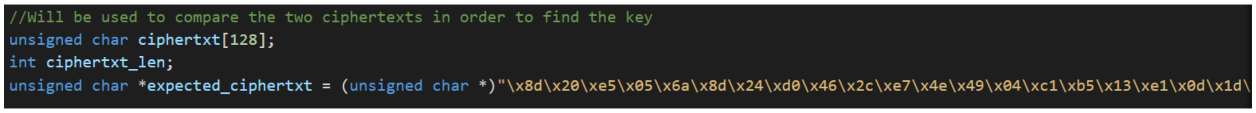
Before even trying to figure out what the key was, I knew I needed to create an encryption function if I were to perform a brute force attack. The encryption function will be explained in simplified bullet points.

- Since the encryption function performs aes-128-cbc encryption, it needs the key, iv, plaintext, ciphertext, and the length of the plaintext.
- Note: that ciphertext is a buffer which will store the encrypted plaintext.
- It then initializes the "context" for aes encryption.
- It then sets up the "context" to perform the aes encryption.
- It then performs the "actual encryption" of the plaintext.
- When the "actual encryption" portion of the code is done, the length of the ciphertext buffer will be updated.
- It then finalizes the encryption and updates the length of the ciphertext buffer one last time.
- After the encryption is complete, the program frees up the context so that no important memory is left lying around, and the length of the ciphertext buffer is returned.

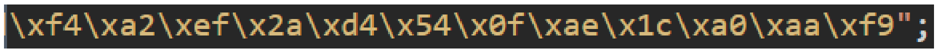


```
int main (void){
    //Since we are using aes-128-cbc, we need a key, iv, and plaintext
    unsigned char *key;
    unsigned char iv[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    unsigned char *plaintxt = (unsigned char *)"This is a top secret.";
    int plaintxt_len = strlen ((char *)plaintxt);
```

I then proceeded to work on the main function of my program. The first part of this code is to set up the initial variables which will be used for the aes-128-cbc encryption. Please note that the iv and plaintext variables are initialized with the values that are documented in the homework assignment.

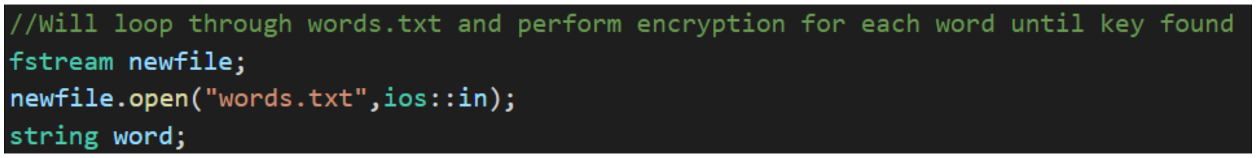


```
//Will be used to compare the two ciphertexts in order to find the key
unsigned char ciphertxt[128];
int ciphertxt_len;
unsigned char *expected_ciphertxt = (unsigned char *)"\x8d\x20\xe5\x05\x6a\x8d\x24\xd0\x46\x2c\xe7\x4e\x49\x04\xc1\xb5\x13\xe1\x0d\x1d\
```



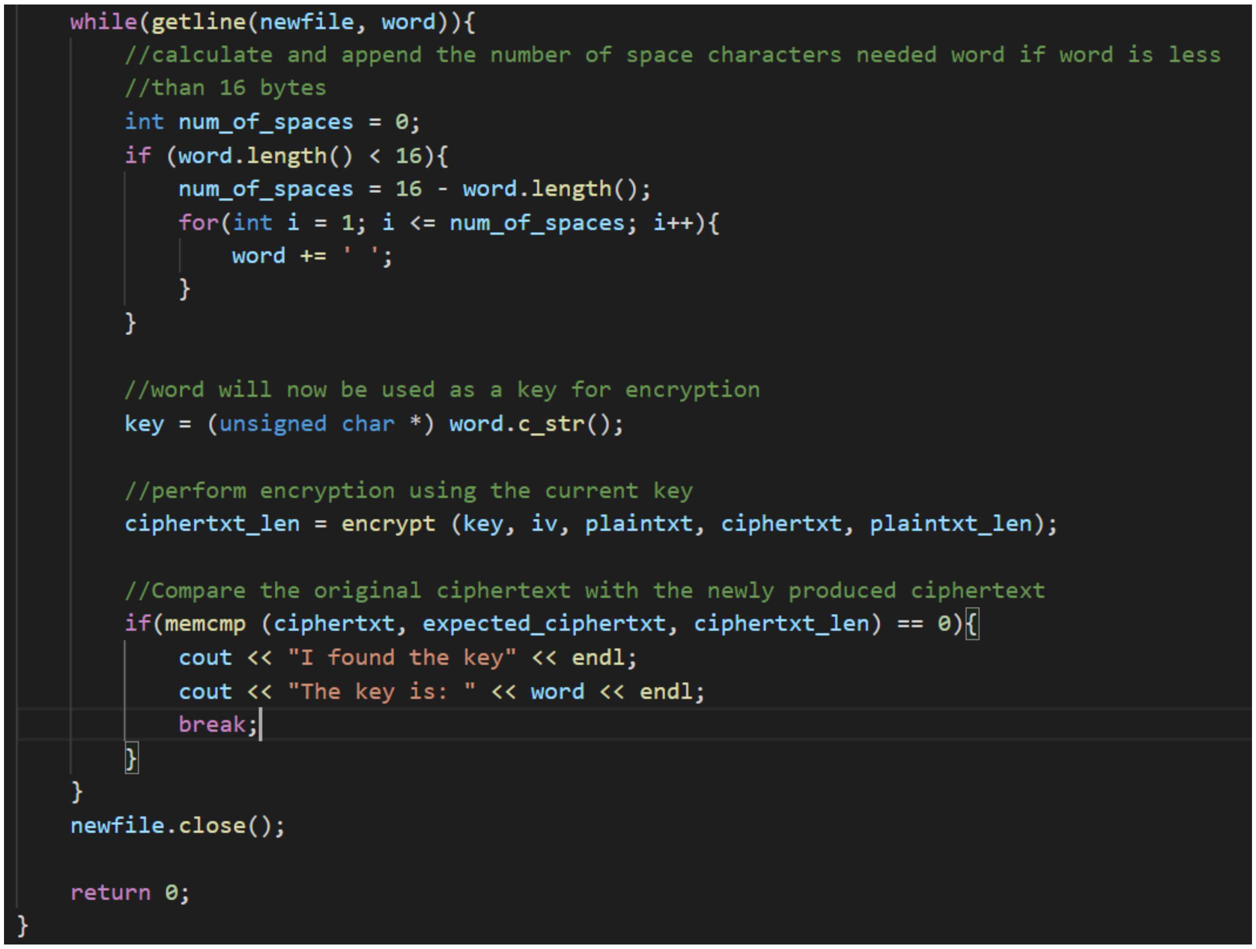
```
\xf4\xa2\xef\x2a\xd4\x54\x0f\xae\x1c\xa0\xaa\xf9";
```

The second part of the code will be used to compare the expected ciphertext with the produced cipher text. Please note that expected_ciphertext has the value as per documented in the homework assignment. Also note, I needed to add in the "\x" since the expected cipher is a hex and not a string.



```
//Will loop through words.txt and perform encryption for each word until key found
fstream newfile;
newfile.open("words.txt",ios::in);
string word;
```

The fourth part of the code will open the provided words.txt file in order to make it traversable so that it can find the correct key. The "word" variable will store the current word that the program is at in the words.txt file.

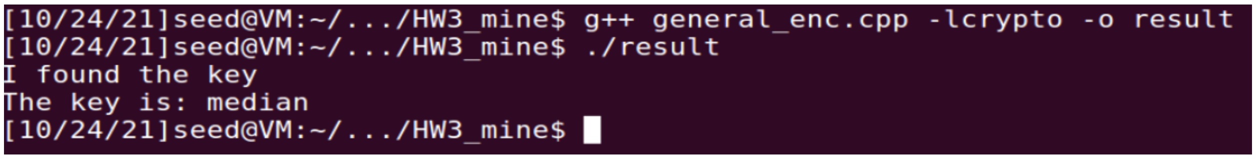


```
    while(getline(newfile, word)){
        //calculate and append the number of space characters needed word if word is less
        //than 16 bytes
        int num_of_spaces = 0;
        if (word.length() < 16){
            num_of_spaces = 16 - word.length();
            for(int i = 1; i <= num_of_spaces; i++){
                word += ' ';
            }
        }

        //word will now be used as a key for encryption
        key = (unsigned char *) word.c_str();

        //perform encryption using the current key
        ciphertxt_len = encrypt (key, iv, plaintxt, ciphertxt, plaintxt_len);

        //Compare the original ciphertext with the newly produced ciphertext
        if(memcmp (ciphertxt, expected_ciphertxt, ciphertxt_len) == 0){
            cout << "I found the key" << endl;
            cout << "The key is: " << word << endl;
            break;
        }
    }
    newfile.close();

    return 0;
}
```

The final portion of the code will perform the actual traversal of the words.txt file. This portion will be explained in simplified bullet points.

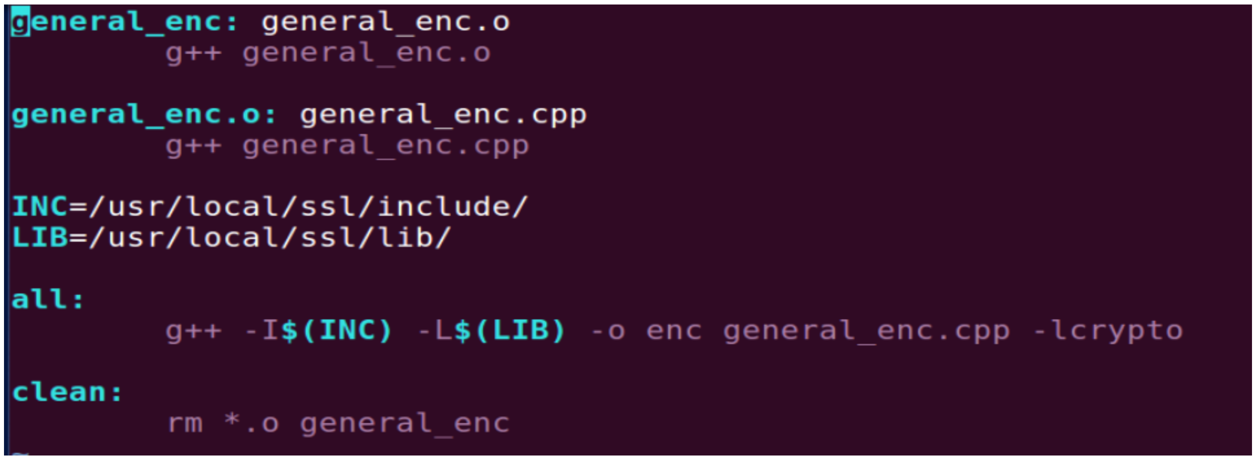
- The code traverses through the words.txt file using the getline() function as all of the words are on their own line.
- If the current word is less than 16 bytes, the code will calculate the amount of blank space characters needed and append them to the end of said word.
- After the process of appending is complete, the word will be converted to an unsigned char pointer to be used for the next aes-128-cbc encrytion.
- The code then calls the encryption function to encrypt the same plaintext using the same iv and current key. Please note that the encryption function will also return the length of the newly produced ciphertext.
- Finally, the program will compare the newly produced ciphertext with the expected ciphertext.
- If they are the same, then the code prints out the key that was used and breaks out of the loop.
- If they are not the same, then the code will continue to run the loop until it traverses to the correct word that produces the expected ciphertext.

- Please note if the while loop ends and it did not find the correct key, then either the words.txt file does not contain the correct key or there is something wrong with my code.
- Once the while loop is done running, then the program will close the words.txt file and return 0.



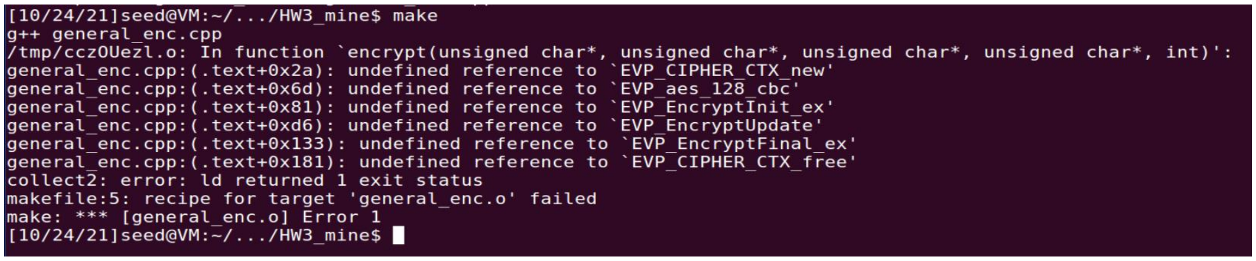
```
[10/24/21]seed@VM:~/.../HW3_mine$ g++ general_enc.cpp -lcrypto -o result
[10/24/21]seed@VM:~/.../HW3_mine$ ./result
I found the key
The key is: median
[10/24/21]seed@VM:~/.../HW3_mine$ 
```

After completing the writing of my code. I proceeded to compile the code with the -lcrypto flag as it is using the openssl library. I then proceeded to run the code, and luckily I was able to find the correct key which is “median”. Thus, the key that was used to produce the expected ciphertext, documented in the homework assignment, is “median”. Of course, since median is less than 16 bytes, it was appended with blank space characters.



```
general_enc: general_enc.o
        g++ general_enc.o

general_enc.o: general_enc.cpp
        g++ general_enc.cpp

INC=/usr/local/ssl/include/
LIB=/usr/local/ssl/lib/

all:
        g++ -I$(INC) -L$(LIB) -o enc general_enc.cpp -lcrypto

clean:
        rm *.o general_enc
```

I would also like to note that I tried creating a makefile for my program. I followed the advice that was documented in the homework assignment, but I could not get it to work for some reason. Every time I ran the make command, I would get the following errors below.



```
[10/24/21]seed@VM:~/.../HW3_mine$ make
g++ general_enc.cpp
/tmp/cczOUezl.o: In function `encrypt(unsigned char*, unsigned char*, unsigned char*, unsigned char*, int)':
general_enc.cpp:(.text+0x2a): undefined reference to `EVP_CIPHER_CTX_new'
general_enc.cpp:(.text+0x6d): undefined reference to `EVP_aes_128_cbc'
general_enc.cpp:(.text+0x81): undefined reference to `EVP_EncryptInit_ex'
general_enc.cpp:(.text+0xd6): undefined reference to `EVP_EncryptUpdate'
general_enc.cpp:(.text+0x133): undefined reference to `EVP_EncryptFinal_ex'
general_enc.cpp:(.text+0x181): undefined reference to `EVP_CIPHER_CTX_free'
collect2: error: ld returned 1 exit status
makefile:5: recipe for target 'general_enc.o' failed
make: *** [general_enc.o] Error 1
[10/24/21]seed@VM:~/.../HW3_mine$ 
```

Obviously, this is due to my lack of experience in creating makefiles. However, I could not resolve this issue. Thus, I disregarded the makefile and I tried compiling the code with g++ with the -lcrypto flag (as shown three images above) and it worked.