References
https://searchsecurity.techtarget.com/definition/security-token
https://wiki.openssl.org/index.php/Enc
https://stackoverflow.com/questions/5403103/hex-to-ascii-string-conversion

HW3

Paper and Pencil

- Q1:
  - It doesn't solve the problem because if the message was appended to the hash, then the intruder could change the hash and the message if they were able to get through. A hash message can be created and appended by anybody.
- Q2:
  - I don't think it is more secure because Alice would still need both Bob and Carol's secret keys to verify if a message was sent from either of them. This logic would be applied to both Bob and Carol as well. So basically everyone would have access to all the different keys. Overall, it is more complicated to do it this way, because any of the people could impersonate the other two.
- Q3:
  - It is common because it is a fixed-size numerical representation of a message that is cannot be reversed. This allows users (integrity checks) - >checksum
  - It is difficult to find two messages with the same digest because if not, the attacker could then modify the message without the receiver knowing there was any tampering, to begin with.
- Q4:
  - This question is describing 2 factor authentication. I have something like this on my phone where an app generates a code every minute then asks me to enter it on my computer to sign in. The way I would design this is by saying that each token card has a unique secret key. A computer used to prove human possession of the device also has access to the secret keys of each authorized device. So I would use those secret keys to encrypt the numbers from the token card using an appropriate cipher type and mode.
- Q5:
  - DES has 56-bit keys, this is because every 8th bit is a parity. There are 2^64 plaintext blocks aligned to 2^64 ciphertext block, so the average would be 2^56/(2^64) = 1/256
- Q6:
  - So because the step for a DES Encryption are Initial Perm, 16 DES rounds, swap halves, then final perm. Since mangler function always output 0, the 16 rounds are insignificant to the message. So I am only looking at step 1, 3, 4.
  - Initial Perm
    -

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

- Swapped halves
    -

57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8

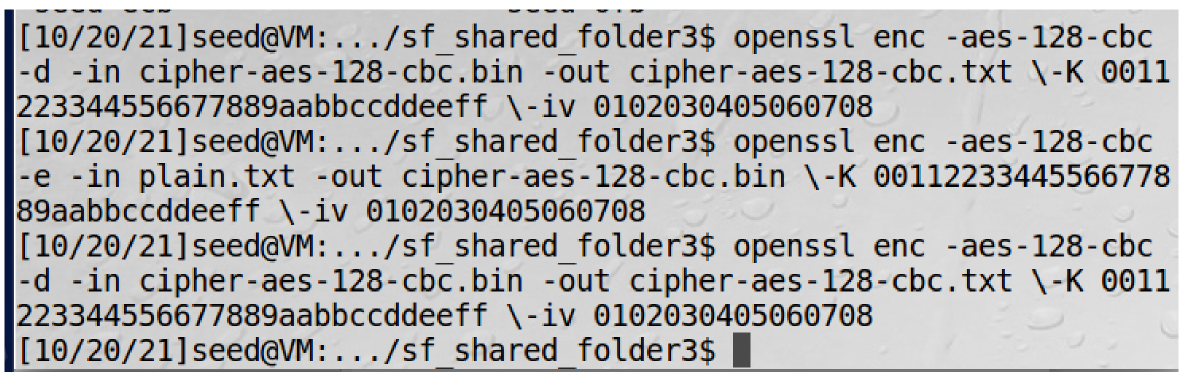
- The final perm would be swapping between the consecutive even and odd numbers

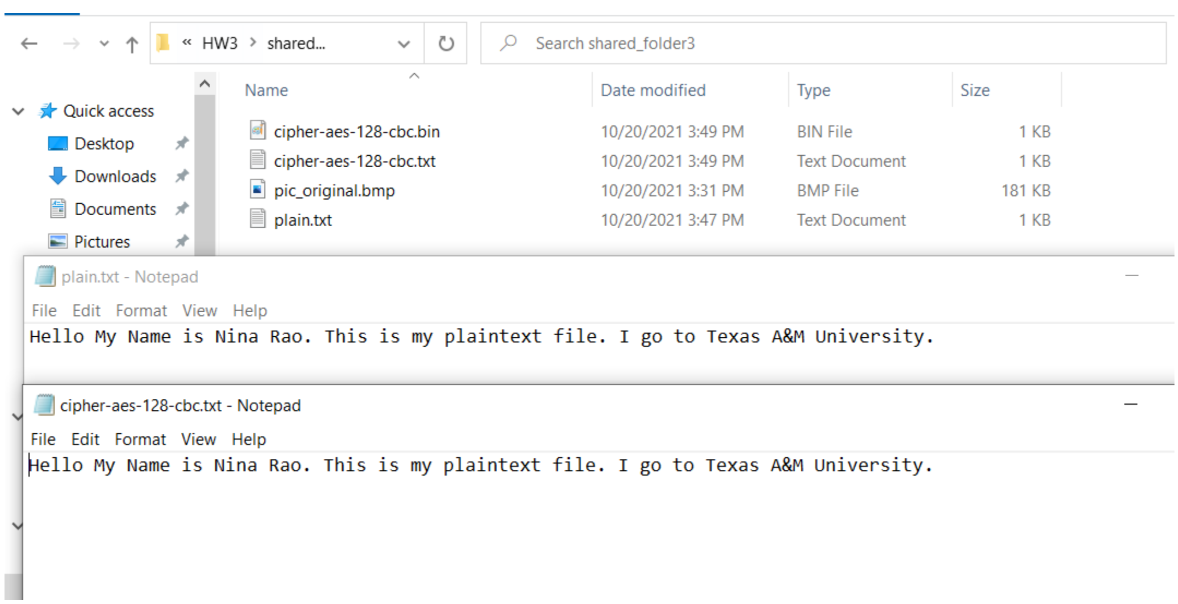
2	1	4	3	6	5	8	7
10	9	12	11	14	13	16	15
18	17	20	19	22	21	24	23
26	25	28	27	30	29	32	31
34	33	36	35	38	37	40	39
42	41	44	43	46	45	48	47
50	49	52	51	54	53	56	55

58	57	60	59	62	61	64	63

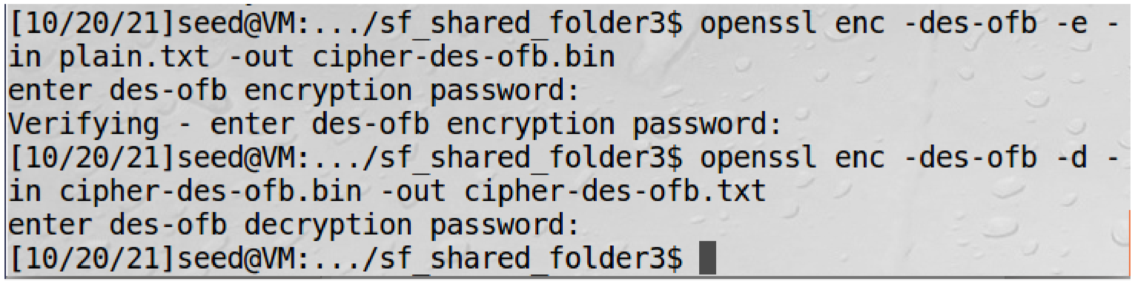
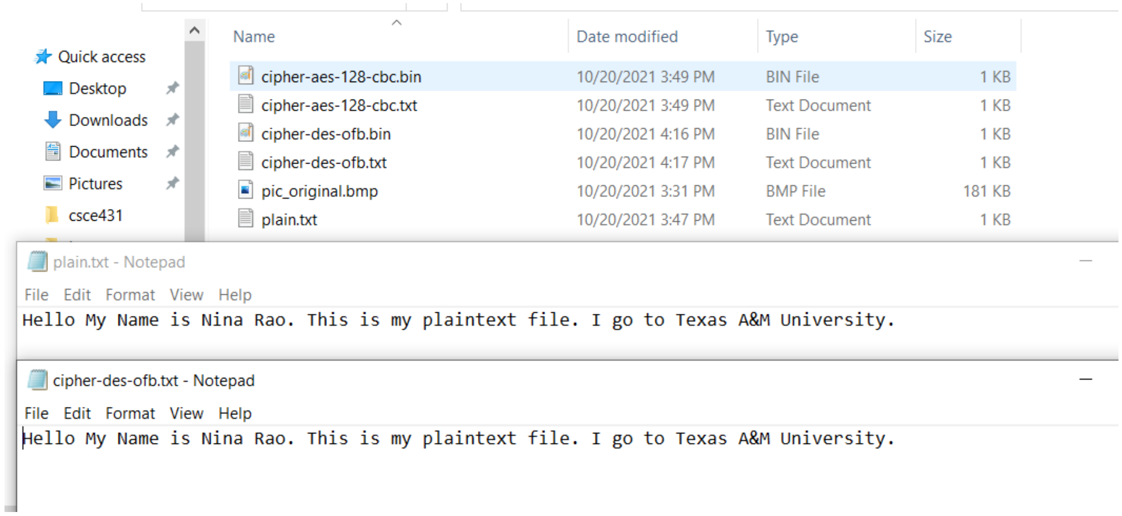
- Q7:
  - K{IV} will be the first block to repeat because encryption is reversible for OFB. So what this tells me is that however many encryptions occur, their is a chain, of K{K{K{K{n{IV} <- this chain occurs n amount of times. The first repeat should be K{IV} because it wouldn't skip to have K{K{IV}} to repeated first or any block from the middle..
- Q8:
  - It would work because CBC encryption and decryption uses XOR. The decryption is just the reverse of the encryption. Security implications would be that error wouldn't propagate as much and it would work in paralled for encryption because that is a property of CBC decryption. So it could be corrupted more.

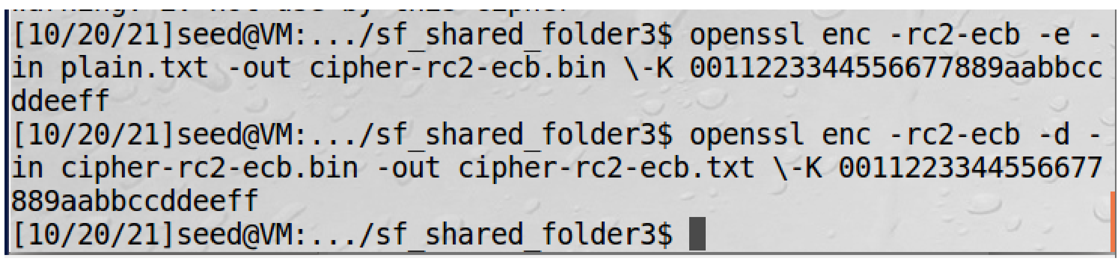
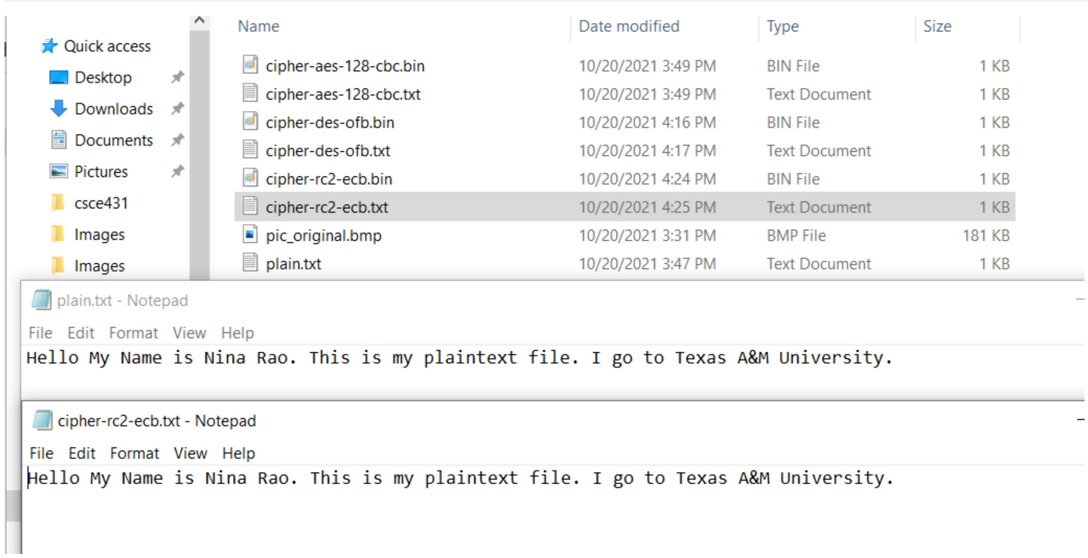
Task 1

- -aes-128-cbc
  - 

  - 

- -des-ofb

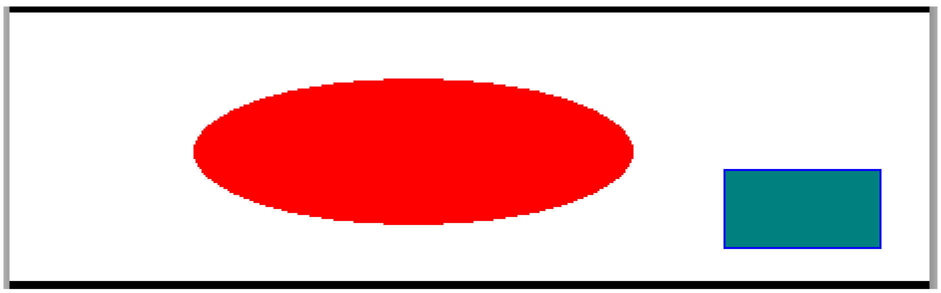
- 
[10/20/21]seed@VM:.../sf_shared_folder3$ openssl enc -des-ofb -e -in plain.txt -out cipher-des-ofb.bin
enter des-ofb encryption password:
Verifying - enter des-ofb encryption password:
[10/20/21]seed@VM:.../sf_shared_folder3$ openssl enc -des-ofb -d -in cipher-des-ofb.bin -out cipher-des-ofb.txt
enter des-ofb decryption password:
[10/20/21]seed@VM:.../sf_shared_folder3$
- 
Quick access
Desktop
Downloads
Documents
Pictures
csce431
Name
Date modified
Type
Size
cipher-aes-128-cbc.bin 10/20/2021 3:49 PM BIN File 1 KB
cipher-aes-128-cbc.txt 10/20/2021 3:49 PM Text Document 1 KB
cipher-des-ofb.bin 10/20/2021 4:16 PM BIN File 1 KB
cipher-des-ofb.txt 10/20/2021 4:17 PM Text Document 1 KB
pic_original.bmp 10/20/2021 3:31 PM BMP File 181 KB
plain.txt 10/20/2021 3:47 PM Text Document 1 KB
plain.txt - Notepad
File Edit Format View Help
Hello My Name is Nina Rao. This is my plaintext file. I go to Texas A&M University.
cipher-des-ofb.txt - Notepad
File Edit Format View Help
Hello My Name is Nina Rao. This is my plaintext file. I go to Texas A&M University.

- -rc2-ecb
    - 
[10/20/21]seed@VM:.../sf_shared_folder3$ openssl enc -rc2-ecb -e -in plain.txt -out cipher-rc2-ecb.bin \-K 00112233445566778899aabbccddeeff
[10/20/21]seed@VM:.../sf_shared_folder3$ openssl enc -rc2-ecb -d -in cipher-rc2-ecb.bin -out cipher-rc2-ecb.txt \-K 00112233445566778899aabbccddeeff
[10/20/21]seed@VM:.../sf_shared_folder3$
    - 
Quick access
Desktop
Downloads
Documents
Pictures
csce431
Images
Images
Name
Date modified
Type
Size
cipher-aes-128-cbc.bin 10/20/2021 3:49 PM BIN File 1 KB
cipher-aes-128-cbc.txt 10/20/2021 3:49 PM Text Document 1 KB
cipher-des-ofb.bin 10/20/2021 4:16 PM BIN File 1 KB
cipher-des-ofb.txt 10/20/2021 4:17 PM Text Document 1 KB
cipher-rc2-ecb.bin 10/20/2021 4:24 PM BIN File 1 KB
cipher-rc2-ecb.txt 10/20/2021 4:25 PM Text Document 1 KB
pic_original.bmp 10/20/2021 3:31 PM BMP File 181 KB
plain.txt 10/20/2021 3:47 PM Text Document 1 KB
plain.txt - Notepad
File Edit Format View Help
Hello My Name is Nina Rao. This is my plaintext file. I go to Texas A&M University.
cipher-rc2-ecb.txt - Notepad
File Edit Format View Help
Hello My Name is Nina Rao. This is my plaintext file. I go to Texas A&M University.

- Explanation:
  - The purpose of this task was to play around with different encryption/decryption cipher types and modes. For each example I encrtyped with a different cipher type and mode. The first one uses --aes-128-cbc, -des-ofb, -rc2-ecb. As you can see, from the screenshots above, each encryption and decryption process worked accordingly because the decrypted output files all match the original plain.txt contents following the respective encryption and decryption commands.

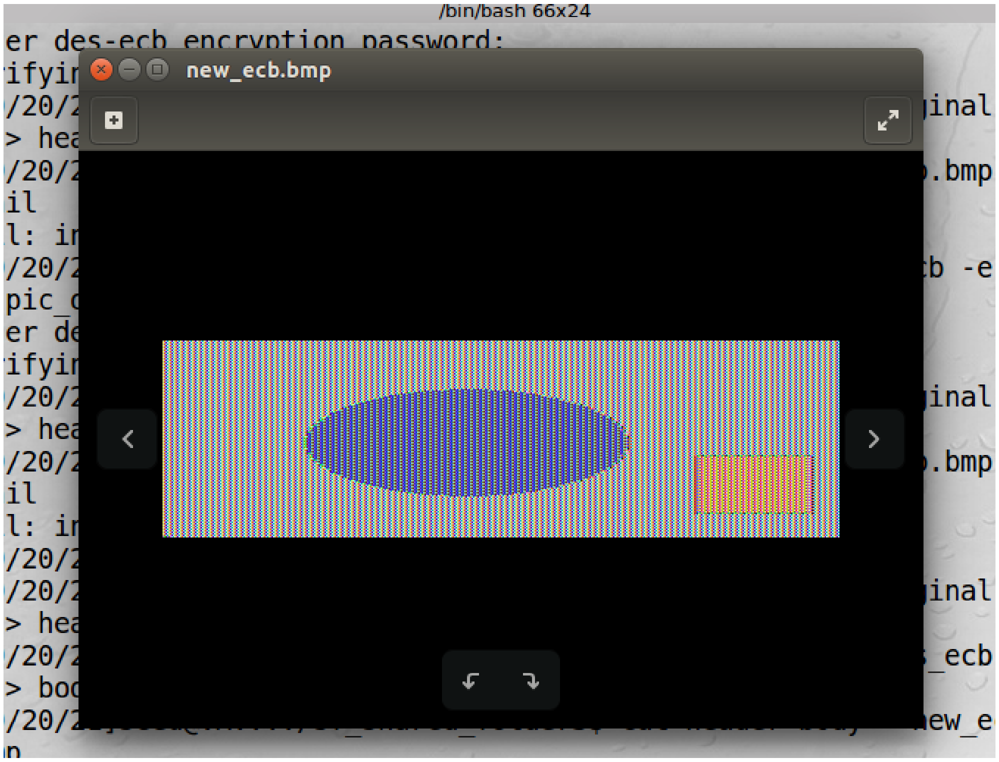
Task 2:

Original Image:

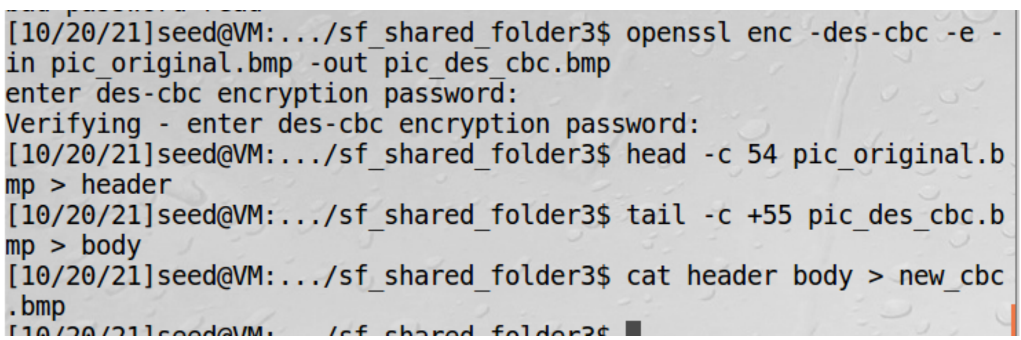


Given Picture:

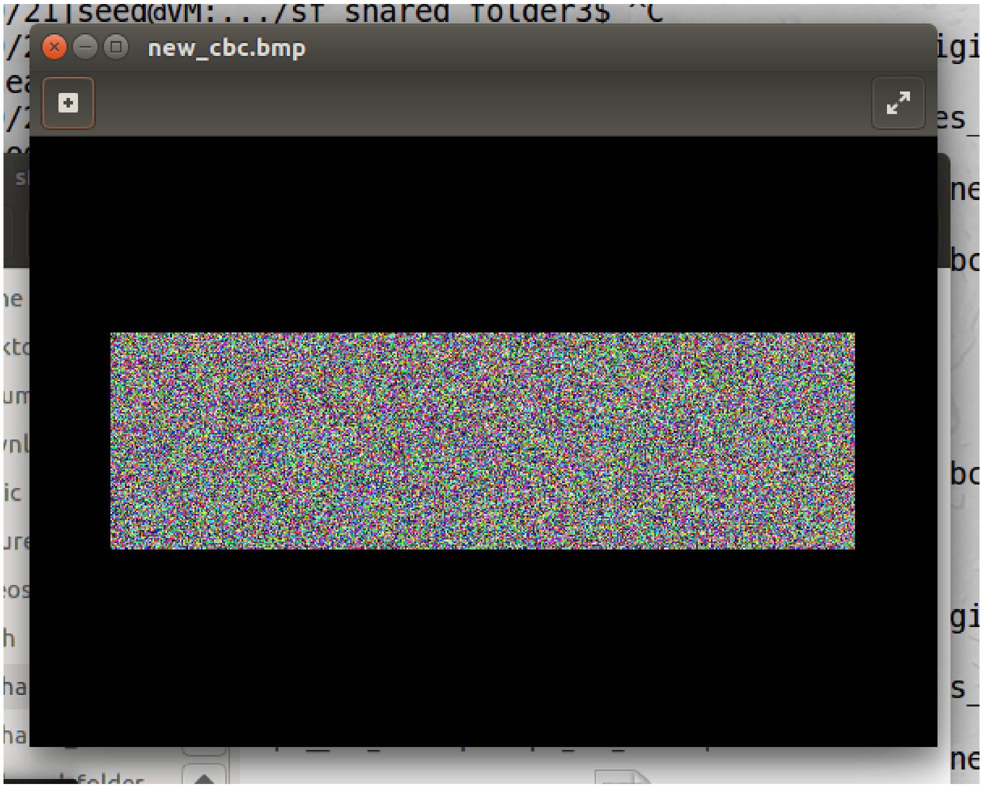
- ECB
  - ```
    [10/20/21]seed@VM:.../sf_shared_folder3$ openssl enc -des-ecb -e -
    in pic_original.bmp -out pic_des_ecb.bmp
    enter des-ecb encryption password:
    Verifying - enter des-ecb encryption password:
    [10/20/21]seed@VM:.../sf_shared_folder3$ head -c 54 pic_original.b
    mp > header
    [10/20/21]seed@VM:.../sf_shared_folder3$ tail -c pic_des_ecb.bmp >
     tail
    tail: invalid number of bytes: 'pic_des_ecb.bmp'
    [10/20/21]seed@VM:.../sf_shared_folder3$ ^C
    [10/20/21]seed@VM:.../sf_shared_folder3$ head -c 54 pic_original.b
    mp > header
    [10/20/21]seed@VM:.../sf_shared_folder3$ tail -c +55 pic_des_ecb.b
    mp > body
    [10/20/21]seed@VM:.../sf_shared_folder3$ cat header body > new_ecb
    .bmp
    [10/20/21]seed@VM:.../sf_shared_folder3$
    ```
    ```



- CBC



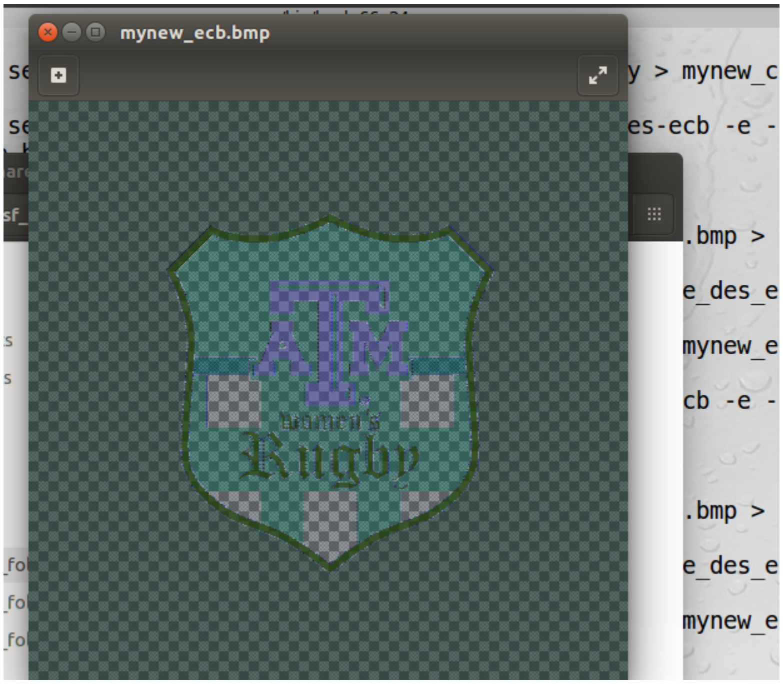
```
[10/20/21]seed@VM:.../sf_shared_folder3$ openssl enc -des-cbc -e -
in pic_original.bmp -out pic_des_cbc.bmp
enter des-cbc encryption password:
Verifying - enter des-cbc encryption password:
[10/20/21]seed@VM:.../sf_shared_folder3$ head -c 54 pic_original.b
mp > header
[10/20/21]seed@VM:.../sf_shared_folder3$ tail -c +55 pic_des_cbc.b
mp > body
[10/20/21]seed@VM:.../sf_shared_folder3$ cat header body > new_cbc
.bmp
```

- 

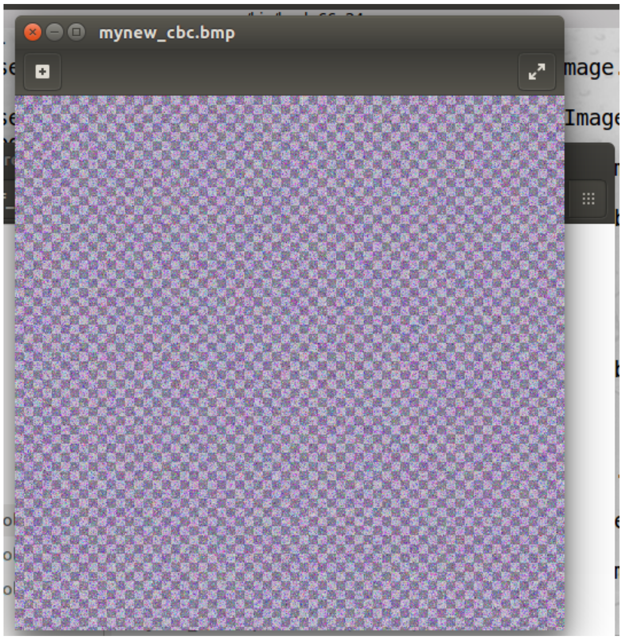
- Hg

My Image:

- ECB
  - ```
    [10/21/21]seed@VM:.../sf_shared_folder3$ openssl enc -des-ecb -e -
    in myImage.bmp -out myImage_des_ecb.bmp
    enter des-ecb encryption password:
    Verifying - enter des-ecb encryption password:
    [10/21/21]seed@VM:.../sf_shared_folder3$ head -c 54 myImage.bmp >
    header
    [10/21/21]seed@VM:.../sf_shared_folder3$ tail -c +55 myImage_des_e
    cb.bmp > body
    [10/21/21]seed@VM:.../sf_shared_folder3$ cat header body > mynew_e
    cb.bmp
    [10/21/21]seed@VM:.../sf_shared_folder3$
    ```
    ```

    - 

- CBC
    - ```
      [10/21/21]seed@VM:.../sf_shared_folder3$ openssl enc -des-cbc -e -
      in myImage.bmp -out myImage_des_cbc.bmp
      enter des-cbc encryption password:
      Verifying - enter des-cbc encryption password:
      [10/21/21]seed@VM:.../sf_shared_folder3$ head -c 54 myImage.bmp >
      header
      [10/21/21]seed@VM:.../sf_shared_folder3$ tail -c +55 myImage_des_c
      bc.bmp > body
      [10/21/21]seed@VM:.../sf_shared_folder3$ cat header body > mynew_c
      bc.bmp
      [10/21/21]seed@VM:.../sf_shared_folder3$
      ```
      ```

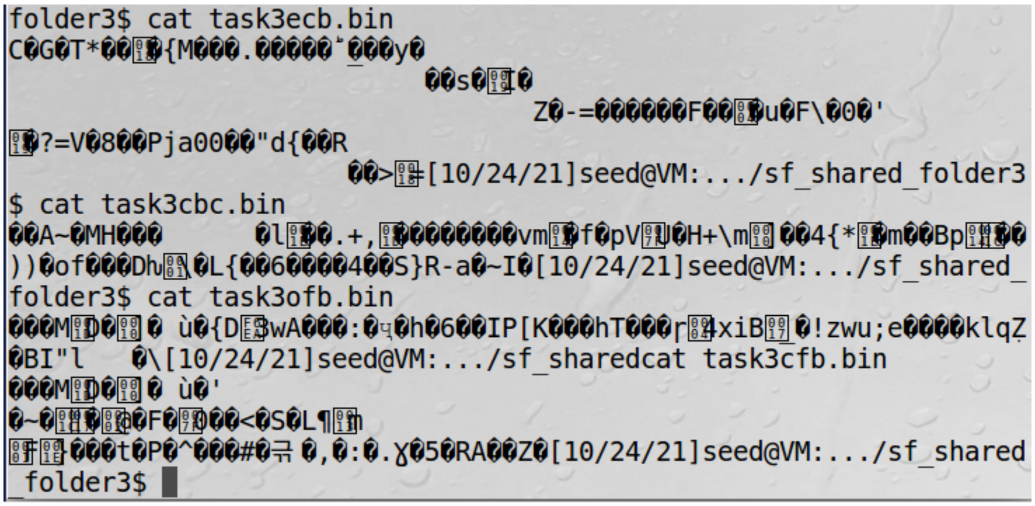
- 

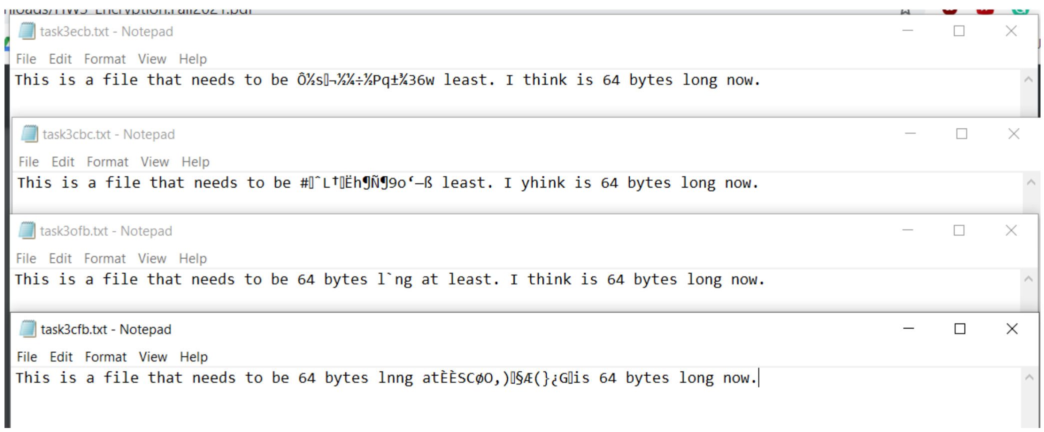
From looking at the above examples it is evident that encrypting with ecb allows you to still make out the original contents of the picture because if you look at the picture given and also my chosen picture, the shapes are still visible. If you look at the encrypted output when encrypting with cbc you can see that the pictures are completely scrambled. This happens because for ecb mode generates similar ciphertext for repeating plaintext but cbc generates different ciphertext for repeating plaintext. So if you are wanting to encrypt an image, then choosing cbc mode is the better option.

Task 3:

1. Before starting the Task
   a. ECB: I think this will recover the most information because based on task 2 you could still make out what the original picture was.
   b. CBC: This will recover the least information based on what I saw from task 2. It completely scrambled the picture.
   c. CFB: I think this mode will recover the 2nd least amount of information because block is dependent on the previous blocks input.
   d. OFB: I think this mode will recover the 2nd most amount of information since it has parallel processing for decryption.

- Proof - For each task I changed the 30th byte using bless hex editor

- 

- 

2. Explain Why I was right or wrong?
    a. <u>ECB</u>: Since the 30th byte was changed, if affected the block of plaintext where the 30th byte was located. This happened because each block is independently encrypted, so the rest of the message stayed intact.
    b. <u>CBC</u>: The 2nd block was corrupted and the 46th byte was also corrupted. So I was sort of correct with my guess because I had initially said that CBC would corrupt the file the most. Either way, it encrypts better than ECB and OFB.
    c. <u>CFB</u>: The 30th byte and the 3rd block were corrupted. So I was sort of correct in that I felt this mode was going to corrupt more of the message than ECB and OFB.
    d. <u>OFB</u>: Only one character in the message was changed. The rest of the message was recovered. So this means only the 30th byte was corrupted. So I was wrong from my initial guess because this mode recovered the most amount of the message.
3. Implications:
    a. ECB: Since the rest of the message stay intact, I don't think this is the best mode of encryption because anyone would be able to recover most of the message except for one block.

b. CBC: This is a good and secure mode of encryption because error propagated through other blocks, while the 30th byte was corrupted.
c. CFB: What it seems is that the corruption in one block, lead to the corruption in another block. This means that this mode is better for encryption over OFB and ECB.
d. OFB: This is not a good encryption method because only one byte was corrupted, so if the message was hacked by an attacker, they would be able to uncover most of the message,

Task 4:
The purpose of this task was to find the key given the plaintext, ciphertext, and a list of words which the key could be. I was also told that the IV was all 0's. This task was accomplished by using the EVP functions as described in the reference given. By asking EVP to use the encryption method aes-128-cbc it allowed me access that method without running openssl tag in the command line. Other than that, I converted the cipher text to ascii characters, and if any key was less than 16 characters I appended the spaces to the end of it to make it 128 bits. After creating the outbuff form the EVP commands, I compared it to the ciphertext, and if all characters matched then that was the key, and I printed it out. So as you can see below, the key found was the word median. The code below shows the most important parts, such as the while loop, which loops through each word in the txt file and appends spaces to any which less than 16 characters. Then it calls all of the importan EVP commands, and then compares the cipher buffer to the outbuf completed by the EVP commands. If those match up we get the key!

** The hex_to_ascii and hex_to_int function were used from the second reference in the document.

```
@VM:.../sf_shared_folder3$ make all
gcc -I/usr/local/ssl/include/ -L/usr/local/ssl/lib/ -o enc task4.c
 -lcrypto
[10/24/21]seed@VM:.../sf_shared_folder3$ ./enc

The encryption key was found to be: median
[10/24/21]seed@VM:.../sf_shared_folder3$
```

Task4.c:

```
/


int find_key(unsigned char cipher[1024], unsigned char outbuf[1024]){
    for(int j=0; j<32; ++j){
        if(cipher[j]!=outbuf[j]){
            return 0;
        }
    }
```

```
    return 1;
}



    while(fgets(key, sizeof(key), words) != NULL){
        int len=0;

        // if key < 16 append spaces to the end 0x20 = ' '
        if(strlen(key) < 16){
            len = strlen(key)-1;
            while(len<16){
                key[len] = 0x20;  //appending the spaces to the end of the
key
                ++len;
            }
        }

        //initualizes encryption to -aes-128-cbc as well as key and iv
        EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, NULL, NULL);

        //makes sure iv and key are correct length
        OPENSSL_assert(EVP_CIPHER_CTX_key_length(&ctx) == 16);
        OPENSSL_assert(EVP_CIPHER_CTX_iv_length(&ctx) == 16);

        EVP_EncryptInit_ex(&ctx, 0, 0, key, iv);

        // updates the output buffer
        EVP_EncryptUpdate(&ctx, outbuf, &outlen, plaintext,
strlen(plaintext));

        //finalizes the output buffer
        EVP_EncryptFinal_ex(&ctx, outbuf + outlen, &buff_out_len);

        outlen += buff_out_len;
        // clean it up
        EVP_CIPHER_CTX_cleanup(&ctx);

        //The check_key function compares the ciphertexts, and it each
matches, then it will
```

```
        // return 1 else it will return 0;
        if(find_key(cipher, outbuf) == 1){
            printf("\nThe encryption key was found to be: %s \n", key);
            break;
        }

    }
```