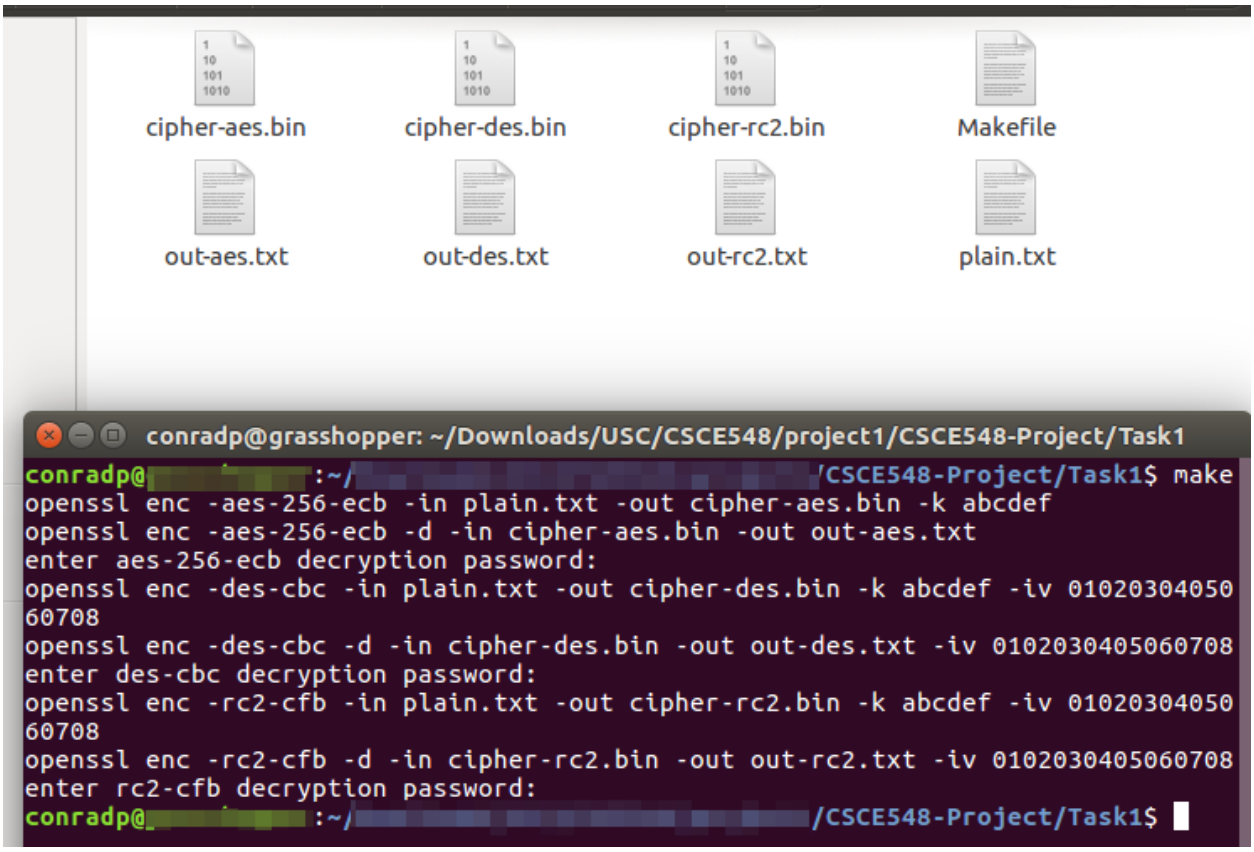
CSCE 548 Project 1
2/10/2018
Group 1: Philip Conrad, Nathaniel Stone, Theodore Stone, Ming Wong

Task1

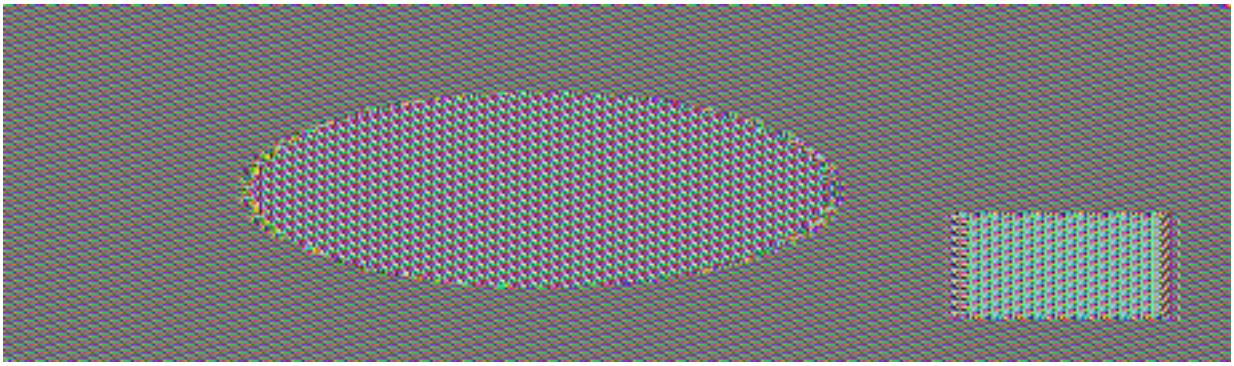


From the above screenshot, the first two commands utilize the AES-256 with ECB mode to encrypt the plain.txt and decrypt the output. Similarly, the next two commands utilize the DES with CBC mode while the last two commands utilize the RC2 with CFB mode. All the decryption results are in the out-*.txt files. We observe that all our decryption results are the same as the original plaintext.

Task 2

In this task, we encrypt one picture using AES-128 algorithm in ecb and cbc mode.
After the encryption, we replace the header of the encrypted picture with the first 54 bytes of the original picture.

Using ECB mode:



Using CBC mode:

After examining the encrypted images visually, we see that when the electronic codebook mode is used, we can clearly discern the patterns of the original image. This is because in this mode, the message is divided into blocks, each of which is processed independently. For this reason, identical plaintext blocks lead to identical ciphertext blocks, leading to information leakage.

When cipher block chaining is used, each previous cipher block is chained with the current plaintext block, meaning that identical plaintext blocks will NOT lead to identical ciphertext blocks, and that patterns in the plaintext will not be preserved in the ciphertext.

Task 3

First, we encrypt the plaintext file with ecb and cbc mode.
openssl aes-128-ecb -e -salt -pass pass:aaa111 -in plaintext.txt -out zout_ecb.txt
openssl aes-128-cbc -e -salt -pass pass:aaa111 -in plaintext.txt -out zout_cbc.txt

Then we modify the 30th byte in both zout_ecb.txt and zout_cbc.txt.

Finally, we decrypt the modified files and generate the plaintext_ecb.txt and plaintext_cbc.txt.
openssl aes-128-ecb -d -salt -pass pass:aaa111 -in zzout_ecb.txt -out plaintext_ecb.txt
openssl aes-128-cbc -d -salt -pass pass:aaa111 -in zzout_cbc.txt -out plaintext_cbc.txt

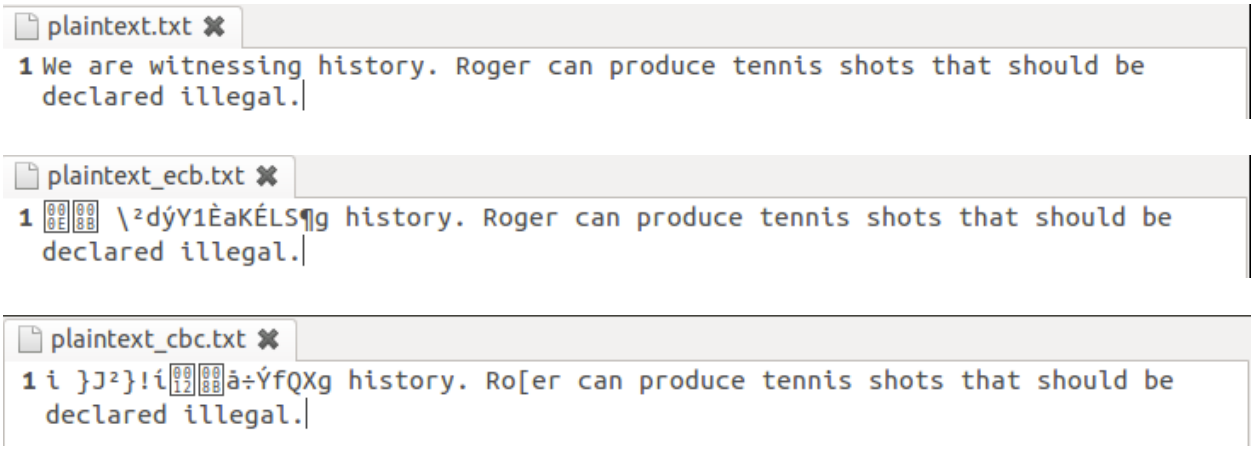
Prediction:

AES algorithm divides the message into 128 bits (16 bytes) block size. This means the corrupted byte 30 is located within the ciphertext block from byte 17 to byte 32.

Under ECB decryption, identical ciphertext block leads to identical plaintext block. Since the ciphertext block from byte 17 to 32 is affected, this means only the corresponding plaintext block is corrupted. Hence, we predict the plaintext blocks from byte 1 to byte 16 and byte 33 to byte 91 can be recovered.

Under CBC decryption, each cipher block is chained with next plaintext block.
Since the ciphertext block from byte 17 to 32 is affected, this means corresponding plaintext block (byte 17-32) and the next plaintext block (byte 33-48) is corrupted. Hence, we predict the plaintext message blocks from bytes 1-16 and bytes 49 -91 can be recovered.

Actual Results:



Our prediction is somewhat correct, but we forgot to take in account of the extra header information in the encrypted files. The original plaintext message is 91 bytes long, while the resulting encrypted message is 112 bytes long.

For ECB, the plaintext block from byte 1 to 16 is affected. This confirms our expectation that corrupted ciphertext block will only affect the corresponding plaintext block.

For CBC, the plaintext block from byte 1 to 16 and the next plaintext block from byte 17 to byte 32 are affected. This confirms our expectation that the corrupted ciphertext block will affect the corresponding plaintext block and one more plaintext block after it due to the chaining effect.

Implication:

From our ECB mode test results, we observe each block is processed independently so one corrupted ciphertext block will only alter its corresponding plaintext block during decryption and vice versa for encryption. This implies that ECB mode encryption will leak too much information for outsiders.

From our CBC mode test results, we observe that ciphertext block is chained to the next plaintext block. This implies that identical plaintext blocks can lead to different ciphertext blocks, and will leak less information than the ECB mode encryption.

Task 4

For this task, we are given an IV vector of all zeros, a plaintext and a ciphertext generated, by the AES-128-CBC. We also know that the key is a word shorter than 16 characters. Our goal for this task is to perform a dictionary attack to find the key.

First we have written a python program (padword.py) that adds padding to ensure every word in the word list to be exactly 128 bits. As shown in the following screenshots, in our task4.c program, we invoke the crypto library to encrypt the each word from the list and compare the encrypted result to the pre-known ciphertext string. If both strings match, we have found the key.

```
// Initialize the cipher contex ctx
EVP_CIPHER_CTX ctx;
EVP_CIPHER_CTX_init(&ctx);

for (i = 0; i < numWords && !found; ++i)
{
    strncpy(word, buffer+i*WORD_LEN, WORD_LEN);

    // Encrypt the plaintext (aes-128-cbc)
    EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, word, iv);

    if (!EVP_EncryptUpdate(&ctx, outbuf, &outlen, ptext, PTEXT_LEN))
    {
        fprintf(stderr, "Error in encrypt update!\n");
        return;
    }

    // Encrypt any data in the final partial block
    if (!EVP_EncryptFinal_ex(&ctx, outbuf + outlen, &tmplen))
    {
        fprintf(stderr, "Error in encrypt final!\n");
        return;
    }

    outlen += tmplen;

     // Compare to pre-known ciphertext
     if (outlen == CTEXT_LEN)
     {
         int j;
         found = 1;

         for (j = 0; j < CTEXT_LEN && found; ++j)
             found = (outbuf[j] == ctext[j]);

         if (found)
             fprintf(stdout, "Key found: %s\n", word);
     }
}
```

From the output of our program, we find out that the key is the 128-bit word “median” .

```
Medea           ->
44ab4aefafcfc5d5a0cf5aaaecd3be 368a1299e776d6b98f6 478b98050151f ?=
8d20e5 56a8d24d0462ce74e49 4c1b513e1 d1df4a2ef2ad454 fae1ca0aaf9
Medford         ->
33bb64c14be3abe1bc76de3f4ebedcde232d64a5837a6d1ed890c459c8efeb92 ?=
8d20e5 56a8d24d0462ce74e49 4c1b513e1 d1df4a2ef2ad454 fae1ca0aaf9
media           ->
f4bdb140224e39a9a6b188155713cd3d6a44fba2 af75b9f27ba167b4a4d 4 6 ?=
8d20e5 56a8d24d0462ce74e49 4c1b513e1 d1df4a2ef2ad454 fae1ca0aaf9
medial          ->
f051ba4b9985d82e8d5619df1c2344c66d40d147 2586f2d2d87bc7913543a9d ?=
8d20e5 56a8d24d0462ce74e49 4c1b513e1 d1df4a2ef2ad454 fae1ca0aaf9
median          ->
8d20e5 56a8d24d0462ce74e49 4c1b513e1 d1df4a2ef2ad454 fae1ca0aaf9 ?=
8d20e5 56a8d24d0462ce74e49 4c1b513e1 d1df4a2ef2ad454 fae1ca0aaf9
Key found: median
```

Lab Questions

What is OpenSSL? What is it used for?
OpenSSL is a software library used for applications that want to secure communications over computer networks. It incorporates an open source implementation of the SSl and TLS protocols, as well as basic cryptographic functions.

Why are Initialization Vectors necessary? Can they be transmitted in plaintext over the Internet?
Initialization Vector is an arbitrary number that can be added to the secret key for data encryption. Because the number generated is random, the IV vector is necessary to prevent the appearance of corresponding duplicate character sequences in the cipher text. This will make it difficult for the attackers to decrypt ciphertexts even if they are generated by the same plaintext and key. The initialization vector can be transmitted in plaintext.

Why are "modes" important?
Modes are important because it describes how block ciphers are applied to message longer than block. Different modes of operations can give different encryptions results with various security properties, so one must choose the mode carefully.

Contributions:

All members participate in the project and verify each other works.
Philip works primarily on Task 1.
Nathaniel works primarily on Task 2.
Ming works primarily on Task 3 and Lab Questions.
Theodore works primarily on Task 4.