Network Security Assignment 2

**Task 1**

Command used :

$ openssl enc ciphertype -e -in plain.txt -out cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Openssl “enc” allows data to be encrypted or decrypted using various block or stream cipher techniques. In this assignment the command used above encrypts(-e) input data “plain.txt” and generates a cipher text “cipher.bin”. The encryption is done by using the actual key, specified after “-K” parameter. When only the key is specified using the -K option, the “–iv” must explicitly be defined. For this task 3 ciphertype have been used namely -aes-128-cbc, -aes-128-cfb, -bf-cbc.

Example

$ openssl enc -aes-128-cbc -e -in plain.txt -out cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Ciphertype

-aes-128-cbc : 128 bit AES in CBC mode

-aes-128-cfb : 128 bit AES in CFB mode

-bf-cbc : Blowfish in CBC mode

**Task 2**

Commands used :

1. openssl enc -aes-128-ecb -e -in original.bmp -out outecb.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708
2. openssl enc -aes-128-cbc -e -in original.bmp -out outcbc.bmp -K 00112233445566778889aabbccddeeff -iv 0102030405060708

In this task we are encrypting a bmp file(“original.bmp “) using ECB and CBC cipher methods. The encrypted files are “outecb.bmp “ and “outcbc.bmp” respectively.  The first 54 bytes of the encrypted picture is replaced by the first 54 bytes of original.bmp as it contains the header information about the picture. To do so the following command is used:

$ dd if=pic\_original.bmp of=outcbc.bmp bs=1 count=54 conv=notrunc

Observation:

**EBC**

Well, EBC, or electronic codeblock, is a block cipher that operates on individual blocks at a time. ECB does not use an initialization vector to kickstart the encryption. So, each block is encrypted with the same algorithm. If any underlying block is the same as another, then the encrypted output is exactly the same. Thus, all "#000000" hexadecimal colors in our image, for example, will have the same encrypted output, per block.

**CBC**

Compare this to CBC, or cipher-block chaining. An initialization vector must be used before the encryption can begin. The parameter after “-iv” in our case is our initialization vector. It is hashed to provide a 128-bit output, then AES encrypts the hash, plus the first block to provide a 256-bit output, 128-bits for the next vector, and 128-bits encrypted output. That vector is then used to encrypt the next 128-bits. This chaining algorithm continues to the end of the file. This ensures that every "#000000" hexadecimal color will have a different output, thus causing the file to appear as random.

Output :

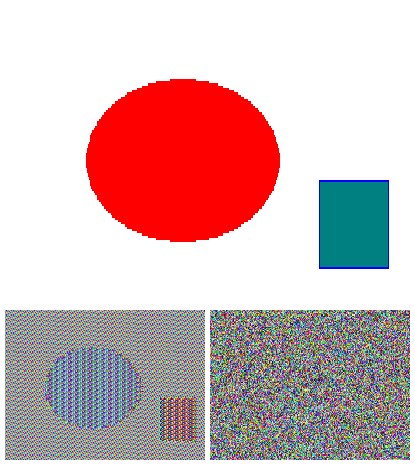


Figure 1: Top image-original.bmp(input). Bottomleft-outecb.bmp(encrypted file using ECB mode), bottomright-outcbc.bmp(encrypted file using CBC mode).

**Task 3**

Command used :

$ openssl enc ciphertype -e -in task3.txt -out cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

For this task a input text file “task3.txt” is created which is more than 64 bytes long. The input file is encrypted using 4 AES ciphertype namely:

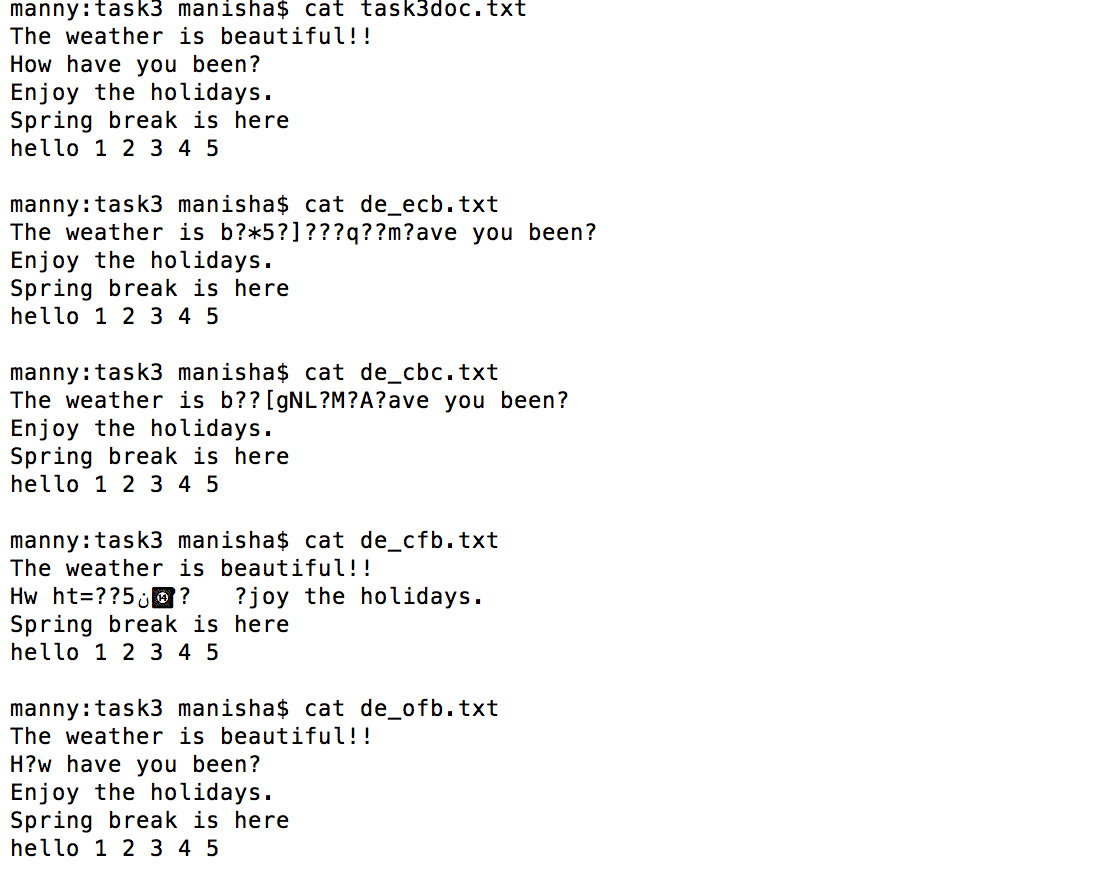
* -aes-128-cbc : 128 bit AES in CBC mode
* -aes-128-cfb : 128 bit AES in CFB mode
* -aes-128-ecb : 128 bit AES in EFB mode
* -aes-128-ofb : 128 bit AES in OFB mode

A single bit of the 30th byte in the encrypted file is corrupted using a hex editor.

This corrupted file is decrypted using the correct key and IV.   The output “test.txt” is the decrypted file generated by decrypting the corrupted files.

$ openssl enc ciphertype -d -in cipher.bin -out test.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Result



From the output decrypted files it can be deduced that:

* In ECB mode, only one block is affected as each block is decrypted independently. The corruption of the 30th affects the entire block as the block size is 16 bytes(128 bits) since we do the decryption one block at a time.
* In CBC mode, there was affect in two blocks. As second block depends on the first block because of chaining method of encryption.
* In CFB by definition of self-synchronising cipher, if part of the ciphertext is lost then receiver will lose only some part of the original message and should be able to continue correct decryption after processing some amount of input data. Therefore, the error propagation criterion is poorer.
* In OFB mode, the feedback is only in the key-generation system. If the 30th byte is corrupted, then in plain text only that byte gets corrupted. Thus, only OFB mode shows the most promising result and almost all the texts are recovered.

**Task 4**

For this task a plaintext and a ciphertext is known, and “aes-128-cbc” is used to generate the ciphertext from the plaintext. The characters in initialization vector are are all zeros (not the ASCII character `0'). The key used to encrypt this plaintext is an English word shorter than 16 characters, the word that can be found in the file “word\_dict.txt”. Since the word has less than 16 characters (i.e. 128 bits), space characters (hexadecimal value 0x20) are appended to the end of the word to form a key of 128 bits. In my code “encdec.c”, plaintext and ciphertext are stored in separate buffers. Key is read line by line from “word\_dict.txt”(dictionary file). The encryption is done using OpenSSL APIs and the ciphertext generated using each key from “word\_dict.txt”(dictionary file) is stored in “ciphertext.txt”. Once a match has been found the key is printed out. Given,

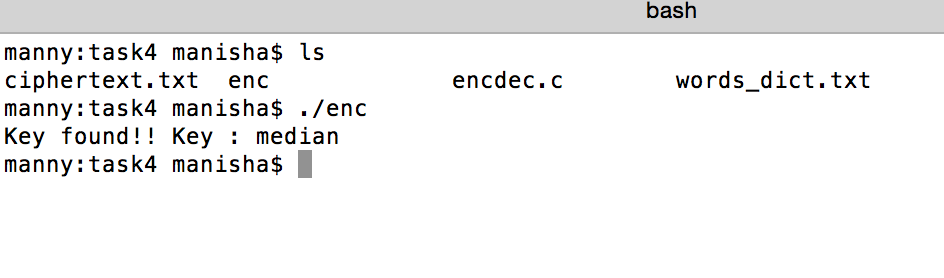
Plaintext (total 21 characters): This is a top secret.

Ciphertext (in hex format):8d20e5056a8d24d0462ce74e4

904c1b513e10d1df4a2ef2ad4540fae1ca0aaf9

Output:

$ gcc -I/usr/local/ssl/include/ -L/usr/local/ssl/lib/ -o enc yourcode.c -lcrypto -ldl

The output shows that the key that used to encrypt the given plain text is **median**.