TASK - 01

Commands that are conducted to solve Task 01 :

```
$ touch task01.txt
$ cat >> task01.txt << 'EOF'
> This is TASK 01 regarding “AES encryption and decryption” from
LAB MANUAL-03. Thank you!
> EOF
```

# Encryption - Decryption using -aes-128-cbc

```
$ openssl enc -aes-128-cbc -e -in task01.txt -out aes-cbc.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708 | more
aes-cbc.bin
V◆.ц
    =b*◆i◆◆K◆◆h◆◆◆~Z◆◆T\W\WJ◆◆"
hex string is too short, padding with zero bytes to length

$ openssl enc -aes-128-cbc -d -in aes-cbc.bin -out task01.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708 | cat
task01.txt
This is TASK 01 regarding aes encryption and decryption from LAB
MANUAL-03. Thank you!
warning: iv not used by this cipher
```

# Encryption - Decryption using -aes-128-cfb

```
$ openssl enc -aes-128-cfb -e -in task01.txt -out aes-cfb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-cfb -d -in aes-cfb.bin -out task01.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708 | cat
task01.txt
```

# Encryption - Decryption using -aes-128-ecb

```
$ openssl enc -aes-128-ecb -e -in task01.txt -out aes-ecb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-ecb -d -in aes-ecb.bin -out task01.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708 | cat
task01.txt
```

TASK 02

# Encryption - Decryption using -aes-128-ecb

```
$ openssl enc -aes-128-ecb -e -in pic_original.bmp -out
encrypted-ecb.bmp -K 00112233445566778889aabbccddeeff -iv
0102030405060708
$ openssl enc -aes-128-ecb -d -in encrypted-ecb.bmp -out
decrypted-ecb.bmp -K 00112233445566778889aabbccddeeff -iv
0102030405060708
```

Where decrypted-ecb.bmp is exactly the same as pic_original.bmp as well as header information.
But, encrypted-ecb.bmp isn't a legitimate formation of bmp. It shows [ Could not load image. BMP image has bogus header data ] errors. So we have to replace the header of the encrypted picture with that of the original picture using ghex. In encrypted-ecb.bmp, I can guess two shapes [ oval, rectangle ] with glitches that are slightly identical to the original one. This encryption compromises the original picture's information.

# Encryption - Decryption using -aes-128-cbc

```
$ openssl enc -aes-128-cbc -e -in pic_original.bmp -out
encrypted-cbc.bmp -K 00112233445566778889aabbccddeeff -iv
0102030405060708
```

Unlike encrypted-ecb.bmp, here in encrypted-cbc.bmp, there is no way to guess a shape or anything else about the original image. That makes a huge difference from ECB one.

```
$ openssl enc -aes-128-cbc -d -in encrypted-cbc.bmp -out
decrypted-cbc.bmp -K 00112233445566778889aabbccddeeff -iv
0102030405060708
```

After changing the header of encrypted-cbc.bmp, decryption method decrypt decrypted-cbc.bmp that doesn't have legitimate formation of bmp. You have to replace the header with the original one to open decrypted bmp in any photo viewer.

# I have also tried using -aes-128-ecb with pbkdf2

```
$ openssl enc -aes-128-ecb -e -pbkdf2 -in pic_original.bmp -out
encrypted-ecb-pbkdf2.bmp
enter aes-128-ecb encryption password: [hello]
```

```
Verifying - enter aes-128-ecb encryption password: [hello]
```
Result is the same as ecb before with -K and -iv. I used PBKDF2 (Password-Based Key Derivation Function 1 and 2) key derivation functions to observe any differences of using ECB from -K and -iv, but no luck. This generates encrypted-ecb-pbkdf2.bmp which is almost same as encrypted-ecb.bmp

## TASK 03

### # 1 2

In ECB mode, only one block is affected when any problem in a ciphertext happens while each block is decrypted independently. The corrupted bit of the 30th byte in ciphertext might spread to all n bits in plaintext of 8 bytes block since we do the decryption one block at a time.

In CBC mode, the corrupted bit not only affects a particular block but also may affect next blocks of plaintext because the corrupted block will produce more corrupted blocks by XORing them.

In CFB mode, the corrupted bit may affect n number of blocks although the error propagation criterion is poorer.

In OFB mode, If the single digit of the 30th byte corrupted, then in plain text that only that byte or character is corrupted.

```
$ cat task03.gedit
This is TASK 03 regarding "Encryption mode- corrupted cipher
text" from LAB MANUAL-03. Thank you!
```

# Encryption - Decryption using -aes-128-ecb

```
$ openssl enc -aes-128-ecb -e -in task03.gedit -out ecb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

```
$ openssl enc -aes-128-ecb -d -in ecb.bin -out ecb-output.gedit
-K 00112233445566778889aabbccddeeff -iv 0102030405060708 | cat
ecb-output.gedit
This is TASK 03 €íÍ?mŒõšý@vEªßption mode- corrupted cipher text"
from LAB MANUAL-03. Thank you!
```

Particular block is affected where change is made [ single bit of 30th byte ]

# Encryption - Decryption using -aes-128-cbc

```
$ openssl enc -aes-128-cbc -e -in task03.gedit -out cbc.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-cbc -d -in cbc.bin -out cbc-output.gedit
-K 00112233445566778889aabbccddeeff -iv 0102030405060708 | cat
cbc-output.gedit
This is TASK 03 \V_o' RÙF ™!$#ption mode- corrupted cipher text"
from LAB MANUAL-03. Thank you!
```

Particular block is affected where the change is made just like ecb.

# Encryption - Decryption using -aes-128-cfb

```
$ openssl enc -aes-128-cfb -e -in task03.gedit -out cfb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-cfb -d -in cfb.bin -out cfb-output.gedit
-K 00112233445566778889aabbccddeeff -iv 0102030405060708 | cat
cfb-output.gedit
This is TASK 03 regarding "Encyˇ$◆{◆◆◆◆◆◆◆2Tpupted cipher text"
from LAB MANUAL-03. Thank you!
```

Corrupted single bit spread corruption to less number of bytes of block unlike ECB or CBC.

# Encryption - Decryption using -aes-128-ofb

```
$ openssl enc -aes-128-ofb -e -in task03.gedit -out ofb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-ofb -d -in ofb.bin -out ofb-output.gedit
-K 00112233445566778889aabbccddeeff -iv 0102030405060708 | cat
ofb-output.gedit
```

```
This is TASK 03 regarding "EncÂyption mode- corrupted cipher
text" from LAB MANUAL-03. Thank you!
```
Single character is corrupted just like we said before!

# 3

*ECB mode* is considered unsuitable for implementation for several reasons. During the process of encryption, the identical blocks are encrypted into identical cipher blocks, which allows for easy recognition of a repeated message. The attacker can also change the order of the encrypted blocks, without giving it away to the receiver. Although this mode of operation is intended for encryption of the data within the size of a single block and is considered as an erroneous one and abandoned, and therefore it should not be used.

In *CBC mode*, by means of the initialization vector – IV, when encrypting identical blocks of plaintext, produces different outputs for every block. If there is an error in a block, it will propagate through all the following blocks. In order to achieve security, the IV must be a random non-predictable value.

*CFB mode* allows stream encryption. The encryption operation is the same as the decryption. An error in a block affects the other blocks.

*OFB mode* also enables stream encryption, and the operation can tolerate block losses, and it can be performed in parallel, both for encryption and decryption, provided the sequence of pad vectors has already been computed. An error in a block does not affect the other blocks. The security of this mode of operation is achieved if the encryption key is changed for every n blocks of encryption, where n is the number of bits in a block.

TASK 04

```
$ openssl dgst -md5 task04.txt
MD5(task04.txt)= d567932cc39e324bf899bacf856d5661
```
The MD5 hash function produces a 128-bit hash value.

```
$ openssl dgst -sha1 task04.txt
SHA1(task04.txt)= d58bad793d8c6244302d4b665dade659785205b9
```
SHA stands for Secure Hash Algorithm. The first version of this algorithm is SHA-1. Whereas MD5 produces a 128-bit hash, SHA1 generates 160-bit hash (20 bytes). In hexadecimal format, it is an integer 40 digits long.

```
$ openssl dgst -sha256 task04.txt
SHA256(task04.txt)=
87872c49a360f896633f9ea61352679c24d50244d157dc8349169c38c1b8018c
```
The second version of SHA, called SHA-2, has many variants. Probably the one most commonly used is SHA-256. The SHA-256 algorithm returns a hash value of 256-bits, or 64 hexadecimal digits. It is considerably more secure than either MD5 or SHA-1, but not quite perfect. A SHA-256 hash is slower to calculate than either MD5 or SHA-1 hashes.

## TASK 05

### # Using -md5

```
$ openssl dgst -md5 -hmac "dlrowOlleh69" task05.txt
HMAC-MD5(task05.txt)= 27ce44f3ffdcee98b977e20d3216f13e

$ openssl dgst -md5 -hmac "rajkst uigyojkef sqpxml cdzvwqpnb"
task05.txt
HMAC-MD5(task05.txt)= f244023793f7b03f4cc4dac1b0a37871

$ openssl dgst -md5 -hmac "helloWorld96" task05.txt
HMAC-MD5(task05.txt)= 71acb5fbdcfa067577b8d5bed1cf7e2d

$ openssl dgst -md5 -hmac "92653589793238" task05.txt
HMAC-MD5(task05.txt)= 45040e363bdfc0da50e224ebdc71a671
```

### # Using -sha1

```
$ openssl dgst -sha1 -hmac "helloWorld96" task05.txt
HMAC-SHA1(task05.txt)= 630207822be84de327fa36903fd666633d37faa2
```

```
$ openssl dgst -sha1 -hmac "rajkst uigyojkef sqpxml cdzvwqpnb"
task05.txt
HMAC-SHA1(task05.txt)= df9df689cb172833cb060334a76593840a11ba52

$ openssl dgst -sha1 -hmac "dlrowOlleh69" task05.txt
HMAC-SHA1(task05.txt)= 76f038401556561c95f0183946481e016356164e

$ openssl dgst -sha1 -hmac "92653589793238" task05.txt
HMAC-SHA1(task05.txt)= 9d7bb92f49fa0a3ab33af24d86a2e1898a5c0f2a
```

# Using -sha256

```
$ openssl dgst -sha256 -hmac "helloWorld96" task05.txt
HMAC-SHA256(task05.txt)=
7b70e04183009d6e9be438d457180599e8a674f2e2a7478b3f7b3e902da6ce07

$ openssl dgst -sha256 -hmac "dlrowOlleh69" task05.txt
HMAC-SHA256(task05.txt)=
82d1b657fe88826d99741eae47e2def0b9ec52d7cf5ef99386afb20bd2dc0e0f

$ openssl dgst -sha256 -hmac "rajkst uigyojkef sqpxml cdzvwqpnb"
task05.txt
HMAC-SHA256(task05.txt)=
3390210e4e87f205dea8d9893878baaa93cfece119874eec394c05b65e36ccbb

$ openssl dgst -sha256 -hmac "92653589793238" task05.txt
HMAC-SHA256(task05.txt)=
fd2e36531a256471782cd8f21daeebca44cd6a3bd45559bcfdcb405da95a991f
```

I've used "rajkst uigyojkef sqpxml cdzvwqpnb" as a key which is 32 bytes long, and also has less entropy. HMAC doesn't break in any terrible way if the key has somewhat less entropy.
The key for HMAC can be of any length. Any key above 128 bits is probably OK. 32 bytes (256 bits) is long enough, and any longer than that probably isn't useful. Password and key aren't the exact same thing. While creating a key, we have to make sure that it's random which produces less entropy.