Task 1: AES encryption using different modes

Check if installed
ghex

Create textfile and open it

nano mytext.txt
robin@Robin:~$ ghex mytext.txt &

Installing OpenSSL
sudo apt-get install openssl -y

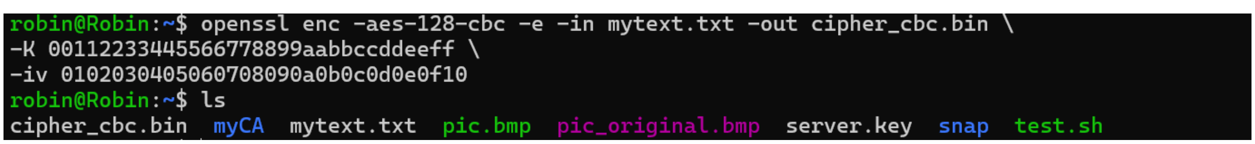
Encrypt the file (3 times, 3 modes)

Key
```
-K 00112233445566778899aabbccddeeff \
```
Initial Vector
```
-iv 0102030405060708090a0b0c0d0e0f10
```

Encryption Commands

1️⃣ AES-128-CBC

```
openssl enc -aes-128-cbc -e -in mytext.txt -out cipher_cbc.bin \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
```

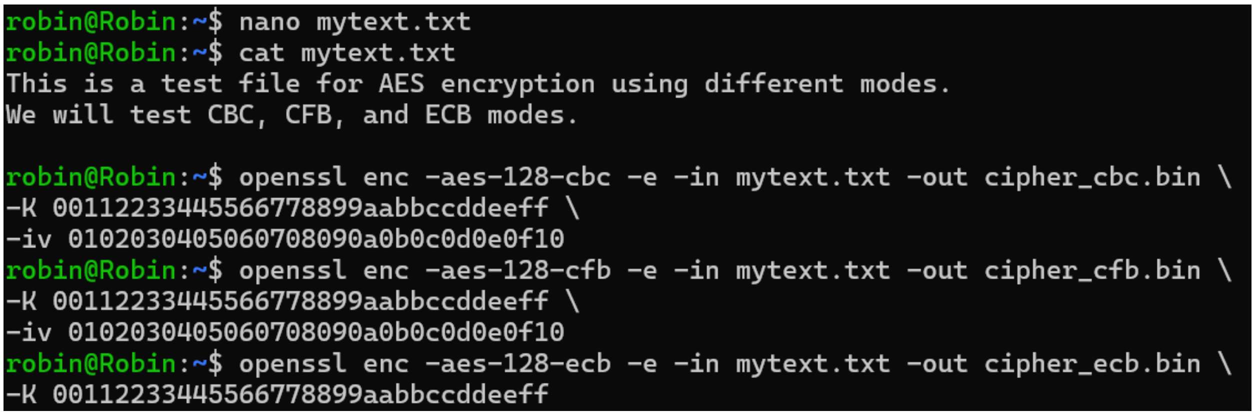

```
robin@Robin:~$ openssl enc -aes-128-cbc -e -in mytext.txt -out cipher_cbc.bin \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
robin@Robin:~$ ls
cipher_cbc.bin  myCA  mytext.txt  pic.bmp  pic_original.bmp  server.key  snap  test.sh
```

2️⃣ AES-128-CFB

```
openssl enc -aes-128-cfb -e -in mytext.txt -out cipher_cfb.bin \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
```

3️⃣ AES-128-ECB (no IV needed)

```
openssl enc -aes-128-ecb -e -in mytext.txt -out cipher_ecb.bin \
-K 00112233445566778899aabbccddeeff
```



```
robin@Robin:~$ nano mytext.txt
robin@Robin:~$ cat mytext.txt
This is a test file for AES encryption using different modes.
We will test CBC, CFB, and ECB modes.

robin@Robin:~$ openssl enc -aes-128-cbc -e -in mytext.txt -out cipher_cbc.bin \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
robin@Robin:~$ openssl enc -aes-128-cfb -e -in mytext.txt -out cipher_cfb.bin \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
robin@Robin:~$ openssl enc -aes-128-ecb -e -in mytext.txt -out cipher_ecb.bin \
-K 00112233445566778899aabbccddeeff
```

To decrypt:

```
openssl enc -aes-128-cbc -d -in cipher_cbc.bin -out decrypted_cbc.txt
\
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708
```

CFB

```
openssl enc -aes-128-cfb -d -in cipher_cfb.bin -out decrypt_cfb.txt \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
```

ECB

```
openssl enc -aes-128-ecb -d -in cipher_ecb.bin -out decrypt_ecb.txt \
-K 00112233445566778899aabbccddeeff
```

```
robin@Robin:~$ openssl enc -aes-128-cbc -d -in cipher_cbc.bin -out decrypt_cbc.txt \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
robin@Robin:~$ openssl enc -aes-128-cfb -d -in cipher_cfb.bin -out decrypt_cfb.txt \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
robin@Robin:~$ openssl enc -aes-128-cfb -d -in cipher_cfb.bin -out decrypt_cfb.txt \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
robin@Robin:~$ cat decrypt_cbc.txt
cat decrypt_cfb.txt
cat decrypt_ecb.txt
This is a test file for AES encryption using different modes.
We will test CBC, CFB, and ECB modes.

This is a test file for AES encryption using different modes.
We will test CBC, CFB, and ECB modes.
```

Output

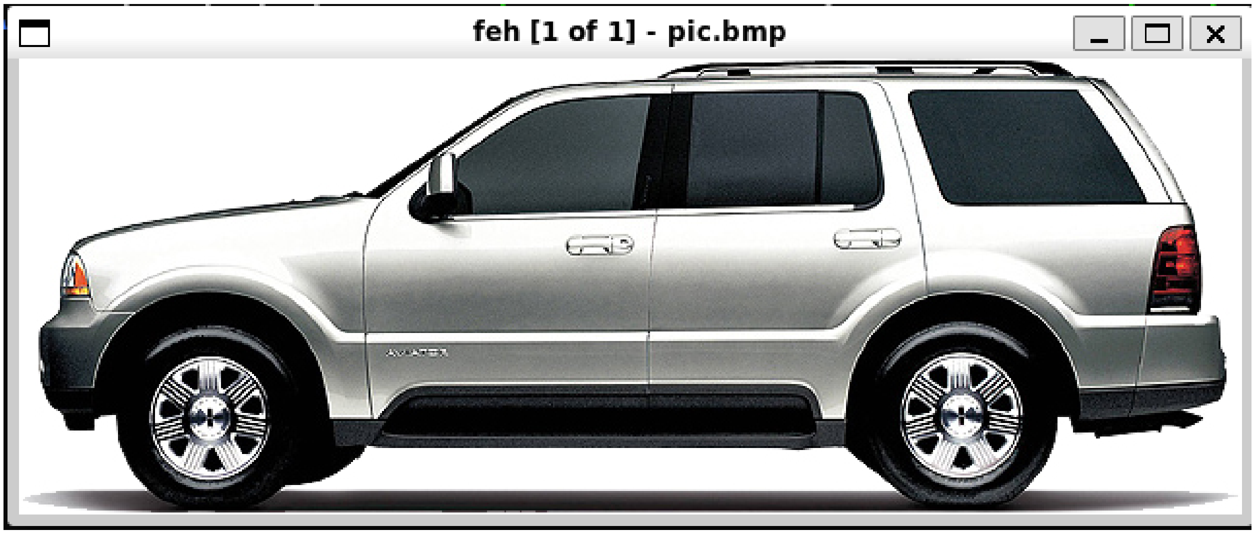
```
robin@Robin:~$ cat decrypt_cbc.txt
cat decrypt_cfb.txt
cat decrypt_ecb.txt
This is a test file for AES encryption using different modes.
We will test CBC, CFB, and ECB modes.

This is a test file for AES encryption using different modes.
We will test CBC, CFB, and ECB modes.

This is a test file for AES encryption using different modes.
We will test CBC, CFB, and ECB modes.
```

Task 2: Encryption Mode – ECB vs CBC

Goal: Encrypt a `.bmp` image using ECB and CBC and analyze the result.



feh pic.bmp

Extract header

```
head -c 54 pic.bmp > header.bin
```

Extract pixel data

```
tail -c +55 pic.bmp > body.bin
```

Encrypt the Pixel Data

4.1 AES-128-ECB

```
openssl enc -aes-128-ecb -in body.bin -out cipherimg_ecb.bin -K
00112233445566778899aabbccddeeff
```

4.2 AES-128-CBC

```
openssl enc -aes-128-cbc -in body.bin -out cipherimg_cbc.bin \
  -K 00112233445566778899aabbccddeeff \
  -iv 0102030405060708090a0b0c0d0e0f10
```

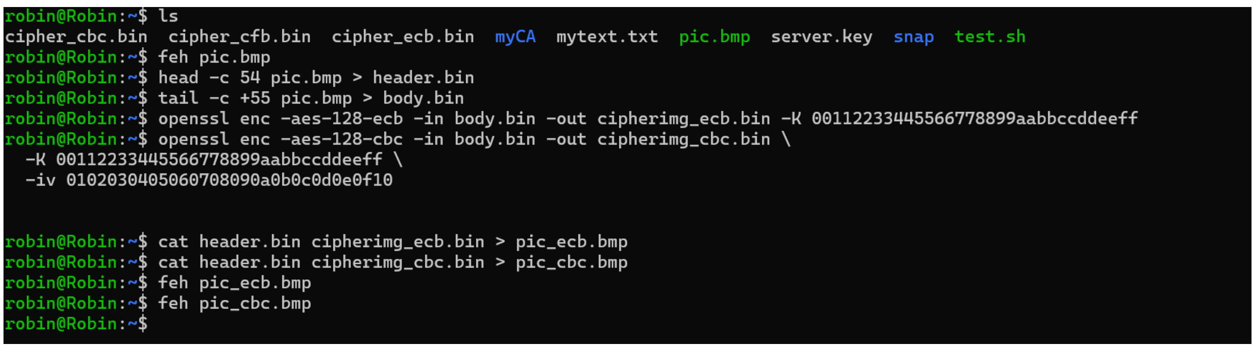
Combine Header with Encrypted Data

5.1 ECB Encrypted BMP

```
cat header.bin cipherimg_ecb.bin > pic_ecb.bmp
```

5.2 CBC Encrypted BMP

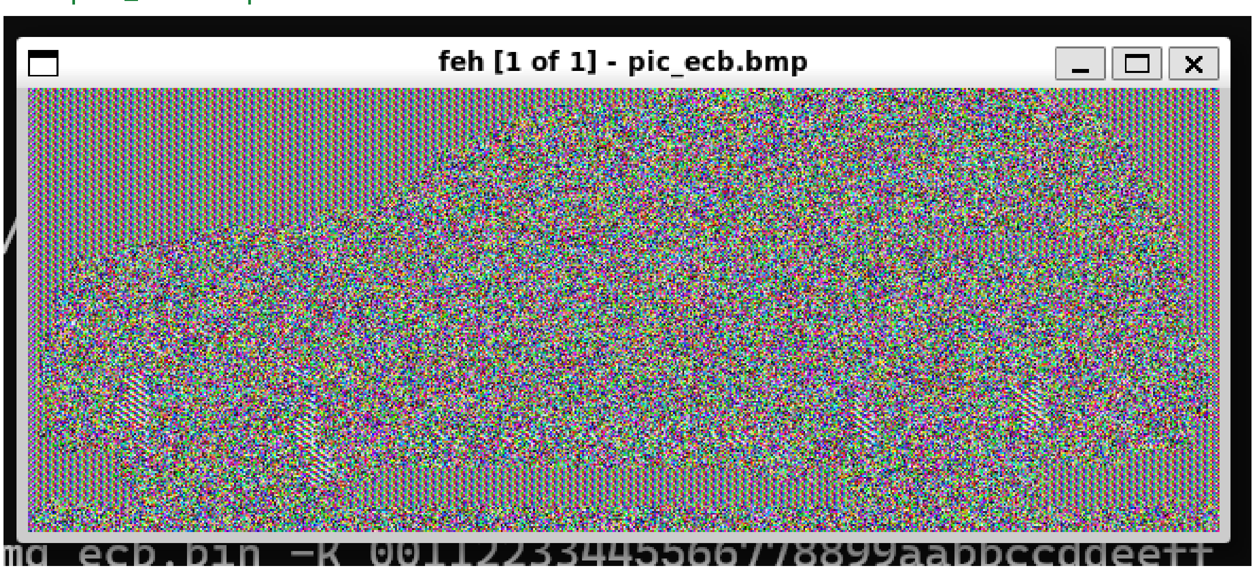
```
cat header.bin cipherimg_cbc.bin > pic_cbc.bmp
```

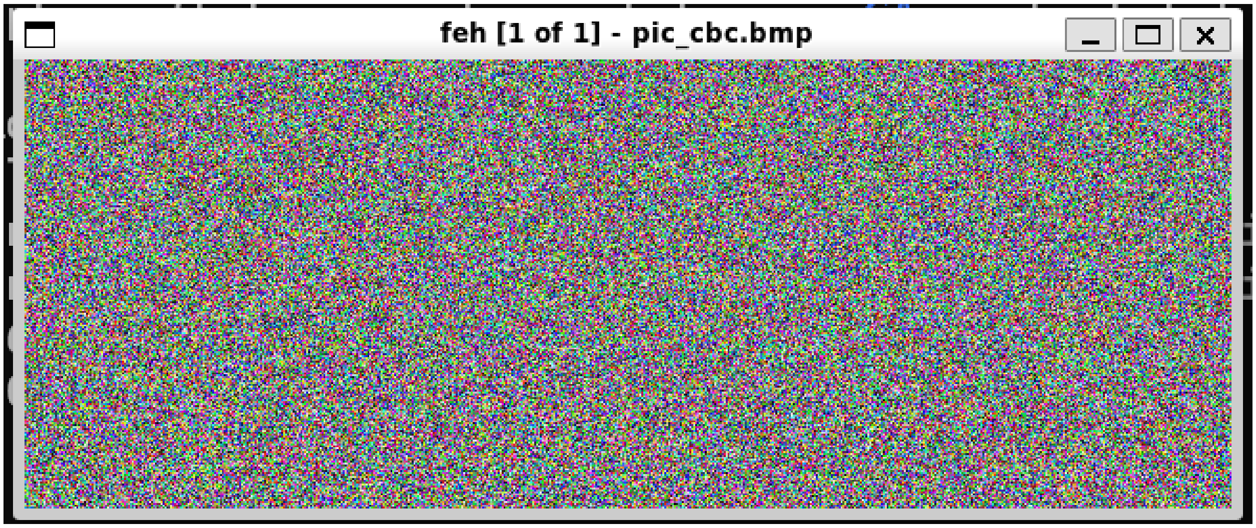


View the Encrypted Images

Using feh in Ubuntu WSL

```
feh pic_ecb.bmp
feh pic_cbc.bmp
```





Objective

The goal of this task is to encrypt a BMP image (`pic.bmp`) using two AES encryption modes: **ECB (Electronic Code Book)** and **CBC (Cipher Block Chaining)**. We then observe the visual differences in the encrypted images to understand how each mode handles data patterns.

- **ECB image (`pic_ecb.bmp`):**
  - Some patterns of the original image are still visible.
  - Identical blocks in the plaintext produce identical ciphertext blocks.
- **CBC image (`pic_cbc.bmp`):**
  - Appears completely random; no visible patterns remain.
  - Each plaintext block is XORed with the previous ciphertext block → patterns are hidden.

Task 3: AES encryption using different modes

Check length:

```
wc -c text.txt
```

Encrypt text.txt using AES-128

We use the same KEY & IV from the lab:

- **Key:** `00112233445566778899aabbccddeeff`
- **IV:** `0102030405060708090a0b0c0d0e0f10`

```
openssl enc -aes-128-ecb -e -in text.txt -out ecb.enc -K 00112233445566778899aabbccddeeff
```

Corrupt the 30th byte

Open each encrypted file in hex editor (example: CBC):

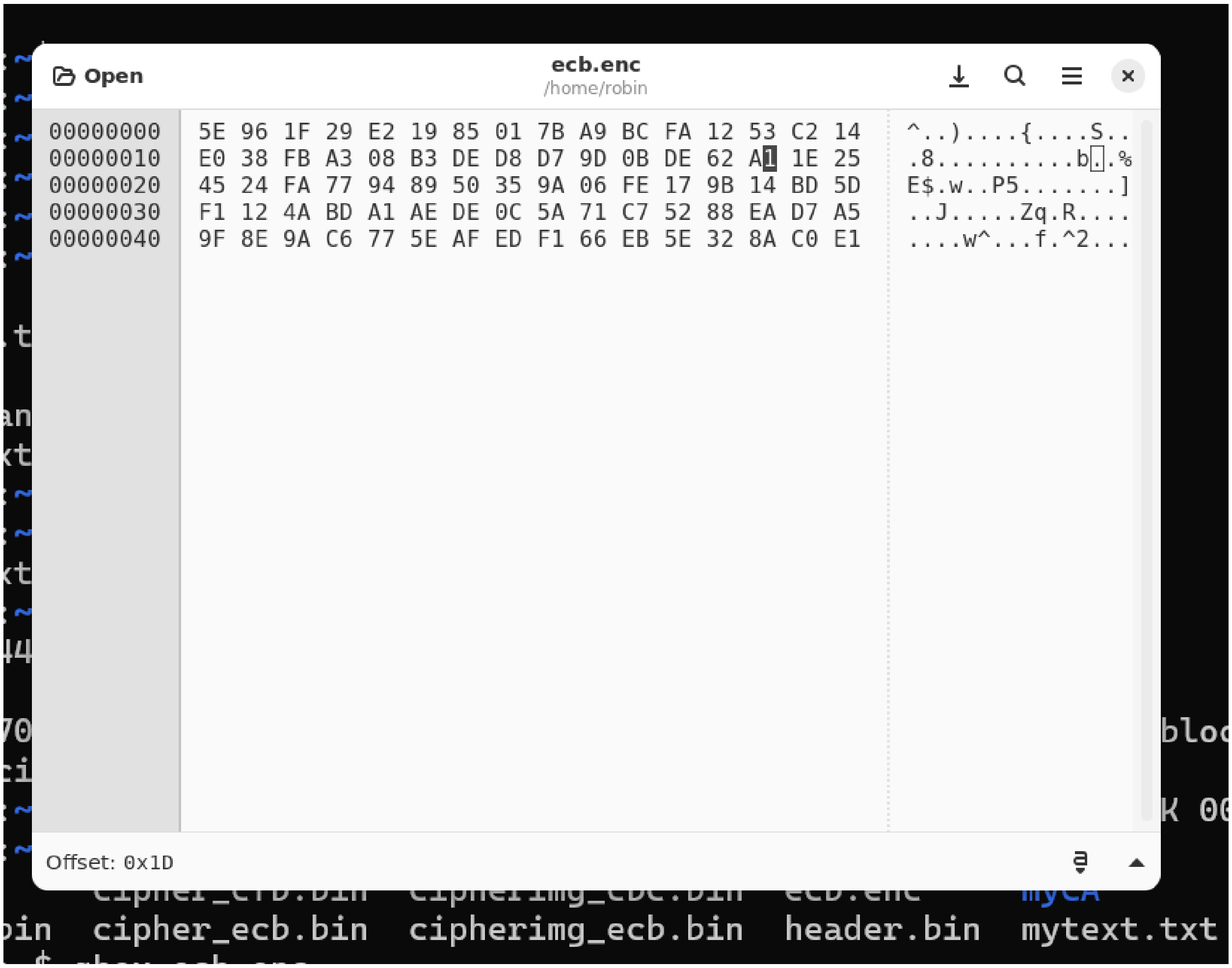
```
ghex cbc.enc
```

Go to offset:

- **0x1D** → 30th byte
  (0-indexed: 0 = 1st byte)

Change 1 bit, e.g.:

```
0B → A1
```



Save as:

```
ecb_corrupted.enc
```

Decrypt corrupted files

ECB Decrypt

```
openssl enc -aes-128-ecb -d -in cbc_corrupted.enc -out
ecb_decrypted.txt \
-K 00112233445566778899aabbccddeeff
```

```
robin@Robin:~$ openssl enc -aes-128-ecb -d -in cbc_corrupted.enc -out ecb_decrypted.txt \
-K 00112233445566778899aabbccddeeff
```

EXPECTED OUTPUT — after corruption

Mode	Result after corrupting 30th byte	Reason
ECB	Only the corrupted block becomes garbled	ECB has no chaining
CBC	Corrupted block + next block affected	CBC uses previous ciphertext block as IV
CFB	Only **one byte** corrupted	CFB is a stream-like mode
OFB	**No corruption at all**	OFB keystream is independent of ciphertext

ECB Output

Task 4: Padding

Prepare a plaintext file

Use your existing file:

`Text.txt`

`"This is a test message for padding experiment."`
`This is **not** a multiple of 16 bytes → good for testing padding.`

Encrypt using all 4 modes

`We use:`

- `Key: 00112233445566778899aabbccddeeff`
- `IV: 0102030405060708090a0b0c0d0e0f10`

1. AES-128-ECB (Block mode → Uses padding)

```
openssl enc -aes-128-ecb -e -in text.txt -out ecb.enc \
-K 00112233445566778899aabbccddeeff
```

ECB always needs padding if plaintext length not multiple of 16.

✅ 2. AES-128-CBC (Block mode → Uses padding)

```
openssl enc -aes-128-cbc -e -in text.txt -out cbc.enc \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
```

CBC also needs padding, because it uses 16-byte blocks.

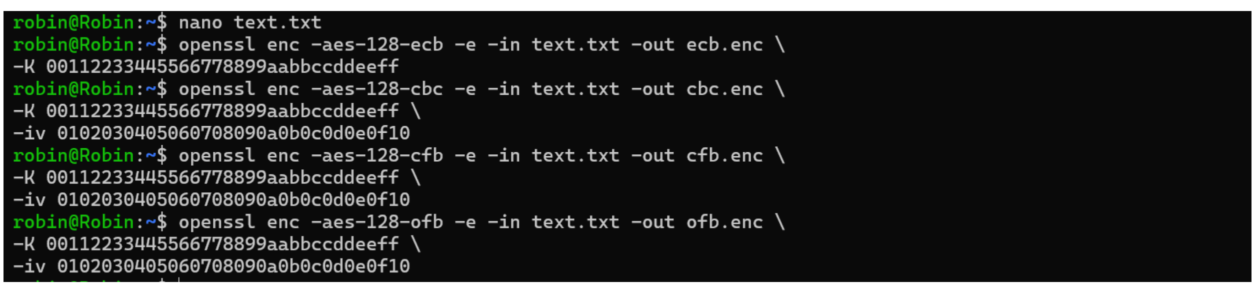
❌ 3. AES-128-CFB (Stream-like mode → No padding needed)

```
openssl enc -aes-128-cfb -e -in text.txt -out cfb.enc \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
```

CFB does NOT use padding, because it encrypts data byte-by-byte (stream-mode).

❌ 4. AES-128-OFB (Stream mode → No padding needed)

```
openssl enc -aes-128-ofb -e -in text.txt -out ofb.enc \
-K 00112233445566778899aabbccddeeff \
-iv 0102030405060708090a0b0c0d0e0f10
```



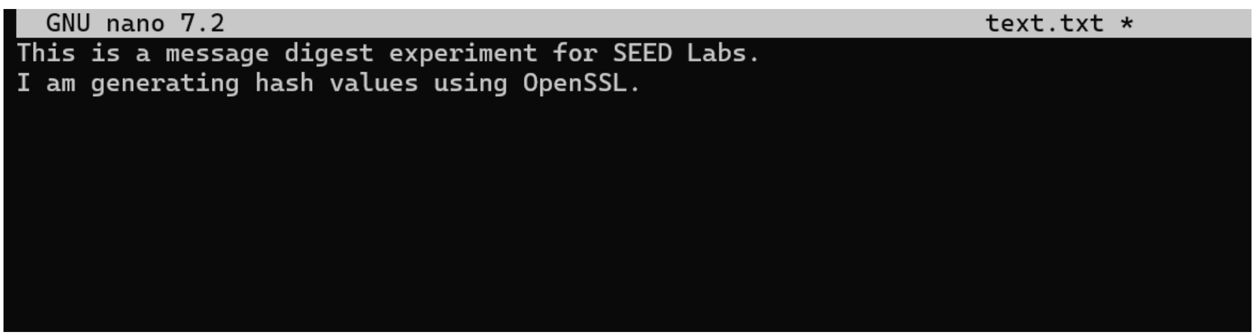
Mode	Padding ?	Why
ECB	✅ Yes	Block cipher → input must be multiple of 16 bytes → OpenSSL adds PKCS#7 padding
CBC	✅ Yes	Also block mode → requires full 16-byte blocks
CFB	❌ No	Stream-like → encrypts byte-by-byte → no block alignment needed

OFB	❌ No	True stream mode → keystream XOR with plaintext → no padding

Task 5: Generating Message Digest

We will:

1. Create a text file
2. Generate hash values using **MD5**, **SHA1**, **SHA256**
3. generated hash outputs



Generate message digests using OpenSSL

General format:

```
openssl dgst -ALGORITHM filename
```

✅ 1. MD5 (128-bit hash)

Command:

```
openssl dgst -md5 text.txt
```

Example output (your value will differ):

```
MD5(text.txt)= 3d4c0a8b8aabf78c28e6a0f80fef4fb7
```

✅ 2. SHA1 (160-bit hash)

```
openssl dgst -sha1 text.txt
```

Example output:

```
SHA1(text.txt)= 8c0a43c275ec01fa0cb2491e3ac3a4fc084d0e67
```

✅ 3. SHA256 (256-bit hash)

```
openssl dgst -sha256 text.txt
```

Example output:

```
SHA256(text.txt)=
62f8b740abcdf9e0e7586d2fadc96f88a57b19e59f3c7d5f05878ddb34734e3d
```

Algorithm	Hash Length	Security Level	Notes
MD5	128-bit	Broken (collisions easy)	Fast, but insecure for modern use
SHA1	160-bit	Weak	Collisions also found (Google SHAttered)
SHA256	256-bit	Strong	Recommended for modern cryptography

Task – 6: Keyed Hash (HMAC)

Create a text file

Generate HMAC using

- HMAC-MD5
- HMAC-SHA1
- HMAC-SHA256

Use multiple keys of different lengths

Explain whether HMAC requires a fixed-size key

Generate HMAC-MD5

Using key = **secret123**

```
openssl dgst -md5 -hmac "secret123" text.txt
```

Example Output:

```
HMAC-MD5(text.txt)= 5f1b8d76d3c953ba9e50a72c1d50a4f1
```

Generate HMAC-SHA256

Using key = **mykey**

```
openssl dgst -sha256 -hmac "mykey" text.txt
```

Example Output:

**HMAC-SHA256(text.txt)=
9fa0b6a20c2281fbe1b0b65f7e24d61c163f214674621c47f15270f9c8cbcf43**

```
robin@Robin:~$ openssl dgst -sha256 -hmac "mykey" text.txt
HMAC-SHA2-256(text.txt)= 9546a1986991ecbe68ac9ef39ba9a6b556d7be90c031cca65653915a9bc0693d
```

Generate HMAC-SHA1

Using key = 123456

```
openssl dgst -sha1 -hmac "123456" text.txt
```

Example Output:

```
HMAC-SHA1(text.txt)= 8b47c41afaf4cf46d79b475d41c587fbba820a09
```

```
robin@Robin:~$ openssl dgst -sha1 -hmac "123456" text.txt
HMAC-SHA1(text.txt)= fba30e87281f35c60d8a3bf4999078e1c6989cef
```

🔹 5. Try different-length keys

Short key:

```
openssl dgst -sha256 -hmac "a" text.txt
```

Long key:

```
openssl dgst -sha256 -hmac "averyverylongsecretkey123456789!" text.txt
```

HMAC does NOT require a fixed-size key.

Why?

- **HMAC internally hashes or pads the key to the block size of the hash function.**
- **Hash function block sizes:**
  - **MD5 → 64 bytes**
  - **SHA1 → 64 bytes**
  - **SHA256 → 64 bytes**

If key > block size → it is first hashed.
If key < block size → it is padded with zeros.

✔ So any key length is acceptable.
✔ But for security, key length ≈ hash output size (e.g., 256-bit for SHA256) is recommended.

Task – 7: Keyed Hash and One-Way Hash Property (3 Marks + Bonus)

Generate original hashes (H1)

MD5

```
openssl dgst -md5 text.txt
```

Example Output:

```
MD5(text.txt)= 56e1bbd7342c97d0a4b9f613c14b0c7a
```

SHA256

```
openssl dgst -sha256 text.txt
```

Example Output:

```
SHA256(text.txt)=
a8ff9d7990ef0d5086840cb5f11c867a2fbcb0da95da54e23a8791af6875743c
```

Save these as **H1(MD5)** and **H1(SHA256)**.

🔹 3. Flip ONE BIT using ghex (hex editor)

Steps:

Run:

```
ghex text.txt
```

1. Change **one bit** (e.g., change ASCII `74` → `75` in one byte).

Save the file as:

```
text_modified.txt
```

2.

🔹 4. Generate modified hashes (H2)

MD5

```
openssl dgst -md5 text_modified.txt
```

Example Output:

```
MD5(text_modified.txt)= c8d9df43bf786bc89d5e13dae089783d
```

SHA256

```
openssl dgst -sha256 text_modified.txt
```

Example Output:

```
SHA256(text_modified.txt)=
5d8b2e417027d1bb642665fa0b7a15bc1989f39fd93b755b4324949a76fba4e6
```