Lab report of lab 3

Prerequiristics:

1. Openssl

OpenSSL provides the SHA256 hash function as part of its cryptographic library. SHA256 is a member of the SHA-2 (Secure Hash Algorithm 2) family and produces a fixed-length 256-bit (32-byte) cryptographic digest for an input message of arbitrary length. It is widely used for ensuring data integrity, digital signatures, and cryptographic security.

2. Hex fiend

Hex Fiend is a fast, clever, and open-source hexadecimal editor designed specifically for macOS. It allows users to view and edit binary files in hexadecimal or ASCII formats efficiently, even for very large files (tested up to over 100 GB in size).

Task 1

Objective:
The task was to encrypt and decrypt a test text file using 3 AES encryption modes—CBC, ECB, and CFB—employing a fixed key and initialization vector (IV) to ensure reproducibility. The goal was to observe mode-specific characteristics and verify correct encryption/decryption.

Procedure:

1. Created a test plaintext file: `text.txt`
2. Used fixed Key and IV for the experiments:
- Key: 00112233445566778889aabbccddeeff
- IV: 0102030405060708
3. Encrypted and decrypted with each AES-128 mode using OpenSSL commands:
- AES-128-CBC mode (uses IV, block chaining):

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

openssl enc -aes-128-cbc -d -in cipher_cbc.bin -out decrypted_cbc.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

- AES-128-ECB mode (no IV, independent block encryption):

```
text

openssl enc -aes-128-ecb -e -in plain.txt -out cipher_ecb.bin -K
00112233445566778889aabbccddeeff

openssl enc -aes-128-ecb -d -in cipher_ecb.bin -out decrypted_ecb.txt -K
00112233445566778889aabbccddeeff
```

- AES-128-CFB mode (uses IV, stream cipher variant):

```
text

openssl enc -aes-128-cfb -e -in plain.txt -out cipher_cfb.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-cfb -d -in cipher_cfb.bin -out decrypted_cfb.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Task 2

Report: Image Encryption Using AES-128 ECB vs. CBC Modes

Objective:
This task demonstrates the difference in encryption effects on an image file when encrypted using AES-128 ECB and CBC modes. The goal is to observe the impact of the mode on data patterns within encrypted images.

Procedure:

1. Obtain or create a sample BMP image named `pic_original.bmp`. This can be a downloaded BMP or an exported image from Preview.
2. Encrypt the image using AES-128 ECB mode (no IV):

```
text

openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb.bmp -K 00112233445566778889aabbccddeeff
```

3. Encrypt the image using AES-128 CBC mode (with IV):

```
text

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

4. View the two encrypted images:

```
text

open pic_ecb.bmp
open pic_cbc.bmp
```

Results and Analysis:

- The ECB-encrypted image (`pic_ecb.bmp`) still reveals the general shapes and structural patterns of the original image. This occurs because ECB encrypts each 128-bit block independently; identical plaintext blocks produce identical ciphertext blocks, preserving visual patterns. This mode leaks structural information and is thus insecure for image or repetitive data encryption.
- The CBC-encrypted image (`pic_cbc.bmp`) appears randomized with no visible structure. CBC mode chains blocks by XORing the plaintext with the previous ciphertext block and uses an initialization vector (IV), ensuring each ciphertext block depends on all previous blocks. This effectively hides patterns and provides stronger security.

Conclusion:
The test clearly shows the insecurity of ECB mode for image data due to pattern leakage, while CBC mode better conceals plaintext structure and is recommended for encrypting image files or data with repetitive patterns.

Task 3

Report: Effect of Ciphertext Corruption on AES-CBC Mode Decryption

Objective:
The experiment aims to observe the impact of a single-bit or byte corruption in the ciphertext on the decrypted plaintext when using AES-128 in CBC mode.

Procedure:

1. Created a plaintext file with sample text exceeding 64 bytes:

```
text
```

```
echo "This is a sample text used to test corruption in AES encryption mode demonstration." > longtext.txt
```

2. Encrypted the file using AES-128-CBC with a fixed key and IV:

```
text
```

```
openssl enc -aes-128-cbc -e -in longtext.txt -out longtext_enc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

3. Corrupted one byte at offset 30 (0x1E) in the encrypted ciphertext file using a hex editor (e.g., Hex Fiend), changing a byte like 3A to 3B, saved as `longtext_corrupted.bin`.
4. Decrypted both original and corrupted ciphertext files:

```
text
```

```
openssl enc -aes-128-cbc -d -in longtext_enc.bin -out dec_ok.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

```
openssl enc -aes-128-cbc -d -in longtext_corrupted.bin -out dec_corrupt.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

Observations:

- The original ciphertext decrypts correctly to the original plaintext.

- The corrupted ciphertext causes decryption errors:
    - The plaintext block corresponding to the corrupted ciphertext block becomes unreadable or garbled.
    - The immediately following plaintext block is partially corrupted due to the chaining nature of CBC mode.
    - All subsequent blocks after the next block decrypt normally ("self-healing" property).

Explanation:
In CBC mode, each plaintext block after decryption is XORed with the previous ciphertext block. Therefore, a changed ciphertext block affects its own plaintext block's decryption and also corrupts the next block through the XOR operation. However, corruption does not propagate beyond two blocks.

Comparison to ECB mode (if tested):

- ECB mode corruption affects only the block containing the corrupted byte since blocks are encrypted independently.

Conclusion:
AES-CBC mode error propagation affects two plaintext blocks when a single ciphertext byte is corrupted, in contrast to one block for ECB. This shows CBC's partial error propagation and self-healing properties, important for understanding data integrity and fault tolerance in encrypted communications.

Task 4

Report: Padding Requirements Across AES Encryption Modes

Objective:
To determine which AES-128 encryption modes require padding of plaintext and explain why, using a sample plaintext file encrypted across four modes: ECB, CBC, CFB, and OFB.

Procedure:

1. Created plaintext file:

```bash
echo "This is a test message for padding experiment." > plain.txt
```

2. Encrypted the file without salt for reproducibility with fixed key and IV (where applicable) using OpenSSL commands:

- ECB mode (requires padding):

```bash
```

```
openssl enc -aes-128-ecb -in plain.txt -out ecb.enc -K
00112233445566778899aabbccddeeff -nosalt
```

- CBC mode (requires padding):

```
bash

openssl enc -aes-128-cbc -in plain.txt -out cbc.enc -K
00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10
-nosalt
```

- CFB mode (no padding needed):

```
bash

openssl enc -aes-128-cfb -in plain.txt -out cfb.enc -K
00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10
-nosalt
```

- OFB mode (no padding needed):

```
bash

openssl enc -aes-128-ofb -in plain.txt -out ofb.enc -K
00112233445566778899aabbccddeeff -iv 01020304050607080 90a0b0c0d0e0f10
-nosalt
```

Observations and Explanation:

Mode	Padding Needed?	Reason
ECB	Yes	ECB encrypts fixed-size blocks independently. Plaintext must be a multiple of 16 bytes, so if it is not, padding is added to complete the final block.
CBC	Yes	CBC also operates on fixed-size blocks and requires padding to fill the last partial block. Padding is essential for proper block chaining.

CFB	No	CFB works as a stream cipher, encrypting smaller units of data sequentially. It does not require input to be block-aligned, so no padding is necessary.
OFB	No	OFB is also a stream cipher mode that generates keystream blocks XORed with plaintext bytes. No block alignment needed, so no padding required.

Conclusion:
Padding ensures input length fits block cipher requirements. Modes like ECB and CBC require padding due to block-wise operations, while stream cipher modes CFB and OFB process data byte-by-byte, eliminating padding needs. Understanding padding is critical to avoid decryption errors and security vulnerabilities.

Task 5

Report: Generating and Comparing Message Digests with MD5, SHA1, and SHA256

Objective:
To generate hash values (message digests) of a sample text using three different one-way cryptographic hashing algorithms—MD5, SHA1, and SHA256—and compare their output lengths, security strength, and typical use cases.

Procedure:

1. Created a sample text file containing:

```
text
```

```
echo "This is a SEED lab hashing experiment." > message.txt
```

2. Generated hashes using OpenSSL commands for each algorithm:

- MD5:

```
text
```

```
openssl dgst -md5 message.txt
```

- SHA1:

```
text
```

```
openssl dgst -sha1 message.txt
```

- SHA256:

```
text

openssl dgst -sha256 message.txt
```

Observations and Explanation:

Algorithm	**Output Length (bits)**	**Collision Resistance**	**Typical Use Cases**
MD5	128 bits	Weak (broken)	Legacy systems, checksum validation (but not secure)
SHA1	160 bits	Weak (partially broken)	Deprecated, previously used in certificates and signatures
SHA256	256 bits	Strong	Current secure hashing standard, digital signatures, certificates

- MD5 produces a 128-bit hash and is very fast but vulnerable to collision attacks, making it unsuitable for security-critical applications.
- SHA1 outputs a longer 160-bit hash, offering somewhat improved security but is also considered insecure due to collision vulnerabilities discovered in recent years.
- SHA256 outputs a 256-bit hash, providing strong resistance against collisions and preimage attacks, making it the preferable choice today for secure hashing needs.

The hash values produced by these algorithms are unique and of increasing length corresponding to their cryptographic strength.

Task 6

Report: Generating HMAC with Variable-Length Keys Using OpenSSL

Objective:
To compute HMAC (Hash-based Message Authentication Code) values for a plaintext file using MD5,

SHA1, and SHA256 hash functions with different variable-length keys. This exercise demonstrates that HMAC keys need not be fixed length, as OpenSSL internally manages key adjustment.

Procedure:

1. Defined three keys of varying lengths:
- KEY1 = "abcdefg" (short key)
- KEY2 = "1234567890abcdef" (medium length)
- KEY3 = "longerkeyforhmac12345" (longer key)
2. Computed HMAC-MD5 for each key:

```
text
```

```
openssl dgst -md5 -hmac "$KEY1" plaintext.txt
openssl dgst -md5 -hmac "$KEY2" plaintext.txt
openssl dgst -md5 -hmac "$KEY3" plaintext.txt
```

Obtained outputs:

- KEY1: c047ecc0852b276947e0a54d814e936a
- KEY2: ded9df1a5726c8bc4ad1d4ae0df6d61c
- KEY3: 94ef3d21197c3176d4e7950167ac93a9

3. Computed HMAC-SHA1 for each key:

```
text
```

```
openssl dgst -sha1 -hmac "$KEY1" plaintext.txt
openssl dgst -sha1 -hmac "$KEY2" plaintext.txt
openssl dgst -sha1 -hmac "$KEY3" plaintext.txt
```

Outputs:

- KEY1: 0d84c8f63bd6e368c9405b03db9002ad00840706
- KEY2: a24af3336bf3dbb3eeb4f82be4d32660140c88ae
- KEY3: 9312fb3dc52973b98d072620e26f8266afb94978

4. Computed HMAC-SHA256 for each key:

```
text
```

```
openssl dgst -sha256 -hmac "$KEY1" plaintext.txt
openssl dgst -sha256 -hmac "$KEY2" plaintext.txt
openssl dgst -sha256 -hmac "$KEY3" plaintext.txt
```

Outputs:

- KEY1: 0a509d2391d5c844a89ba976dbb4cac86fd224e149e08a41a62b6dfead76ed94
- KEY2: 6a049b7cdb46a6caf2bdccbb75687bfc0ab8720b0f50743785a4b853b8a40660
- KEY3: 32d8a9f3a56d06a5a33a6fe0ba880d68d70f6f3d8b142c7c0361c22bc6c61e62

Key Insights:

- HMAC accepts keys of any length.

- For keys longer than the hash block size, OpenSSL hashes the key before use.
- For shorter keys, the key is padded to the block size internally.
- This handling ensures consistent HMAC operation regardless of original key length or size.

Conclusion:
OpenSSL supports variable-length keys for HMAC generation seamlessly, abstracting complexity from the user. The different keys produce distinct HMAC values for the same file content across MD5, SHA1, and SHA256, demonstrating key sensitivity in HMAC security.

Task 7

Objective:
To observe how a small change (a single bit) in the input file drastically affects the output hash when using the SHA256 cryptographic hash function, illustrating the important property known as the avalanche effect.

Procedure:

1. Created two files:
- Original: plaintext.txt
- Slightly modified: copy_plaintext.txt (differs by one bit from plaintext.txt)
2. Generated SHA256 hash values of both files using OpenSSL:

```
text
```

```
openssl dgst -sha256 plaintext.txt
openssl dgst -sha256 copy_plaintext.txt
```

Observations:

- The hash outputs H1 and H2 are completely different despite the minimal change in input.
- Bit-level comparison shows only 163 bits out of 256 bits (63.67%) matched between H1 and H2, highlighting a drastic output difference.
- This demonstrates the avalanche effect: even a single bit change in input produces a drastically different hash output.

Importance of Avalanche Effect:

- It ensures strong diffusion in cryptographic algorithms.
- Prevents attackers from making predictions about input based on output.
- Essential for hash function security to avoid collisions and maintain unpredictability.

Conclusion:
This experiment confirms SHA256 exhibits a strong avalanche effect, crucial for its role in secure hashing and cryptography. It effectively ensures that small changes in data produce uncorrelated, unpredictable hash outputs, enhancing overall security.