Information Security

Thursday 9/12/2024

—

Ashad Abdullah Qureshi

K21-3296 Section B

NOTE: ***This Lab is performed using WSL2 (Windows Subsystem for Linux 2) on a Windows operating system.***

Tasks

Task 01:

Substitution Map:

```
substitution_map = {
    'h': 'r',
    'l': 'w',
    't': 'h',
    'a': 'c',
    'f': 'v',
    'w': 'z',
    'z': 'u',
    'v': 'a',
    'q': 's',
    'o': 'j',
    'j': 'q',
    'g': 'b',
    'r': 'g',
```

```
    'u': 'n',

    'd': 'y',

    'y': 't',

    'e': 'p',

    'n': 'e',

    'm': 'i',

    'x': 'o',

    'k': 'x',

    'p': 'd',

    'b': 'f',

    'i': 'l',

    'c': 'm',

    's': 'k',

}
```

Decrypted Text (First 3 Paragraphs)

the oscars turn on sunday which seems about right after this long strange awards trip the bagger feels like a nonagenarian too

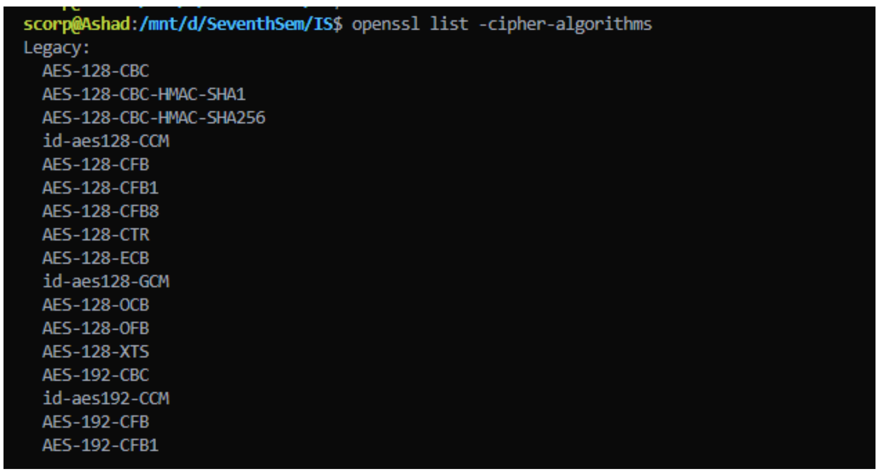
the awards race was bookended by the demise of harvey weinstein at its outset and the apparent implosion of his film company at the end and it was shaped by the emergence of metoo times up blackgown politics armcandy activism and a national conversation as brief and mad as a fever dream about whether there ought to be a president winfrey the season didnt just seem extra long it was extra long because the oscars were moved to the first weekend in march to avoid conflicting with the closing ceremony of the winter olympics thanks pyeongchang

one big question surrounding this years academy awards is how or if the ceremony will address metoo especially after the golden globes which became a jubilant comingout party for times up the movement spearheaded by powerful hollywood women who helped raise millions of dollars to fight sexual harassment around the country

Task 02:

Code

I experimented with different encryption algorithms and modes using the OpenSSL tool in this task. We applied encryption and decryption on an output file from the previous task (i.e. Task 1) (`decryption.txt`) using three different cipher types: `AES-128-CBC`, `Blowfish-CBC (BF-CBC)`, and `AES-128-CFB`. The results of each encryption method were observed, and decryption was performed to verify the integrity of the process.



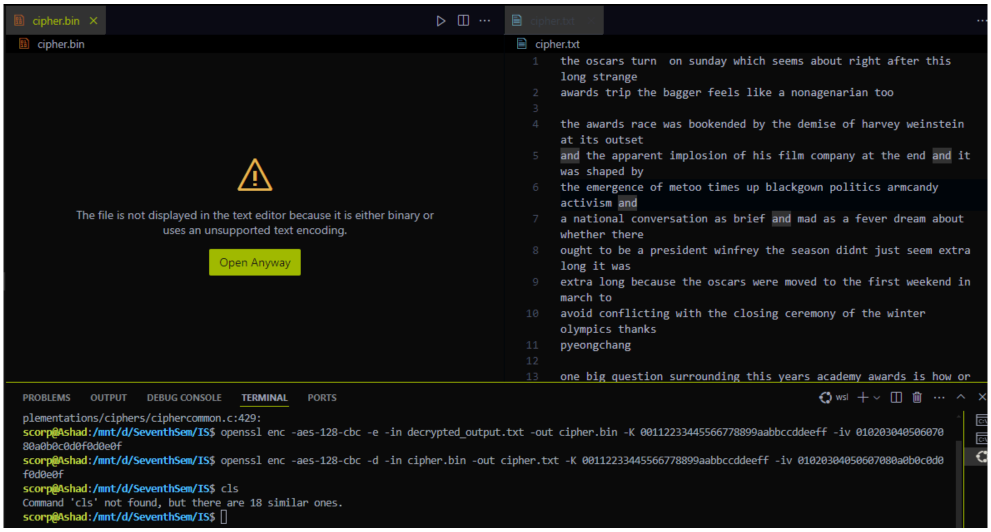
AES-128-CBC

Encryption

openssl enc -aes-128-cbc -e -in decrypted_output.txt -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708 0a0b0c0d0f0d0e0f

Descryption

openssl enc -aes-128-cbc -e -in decrypted_output.txt -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708 0a0b0c0d0f0d0e0f



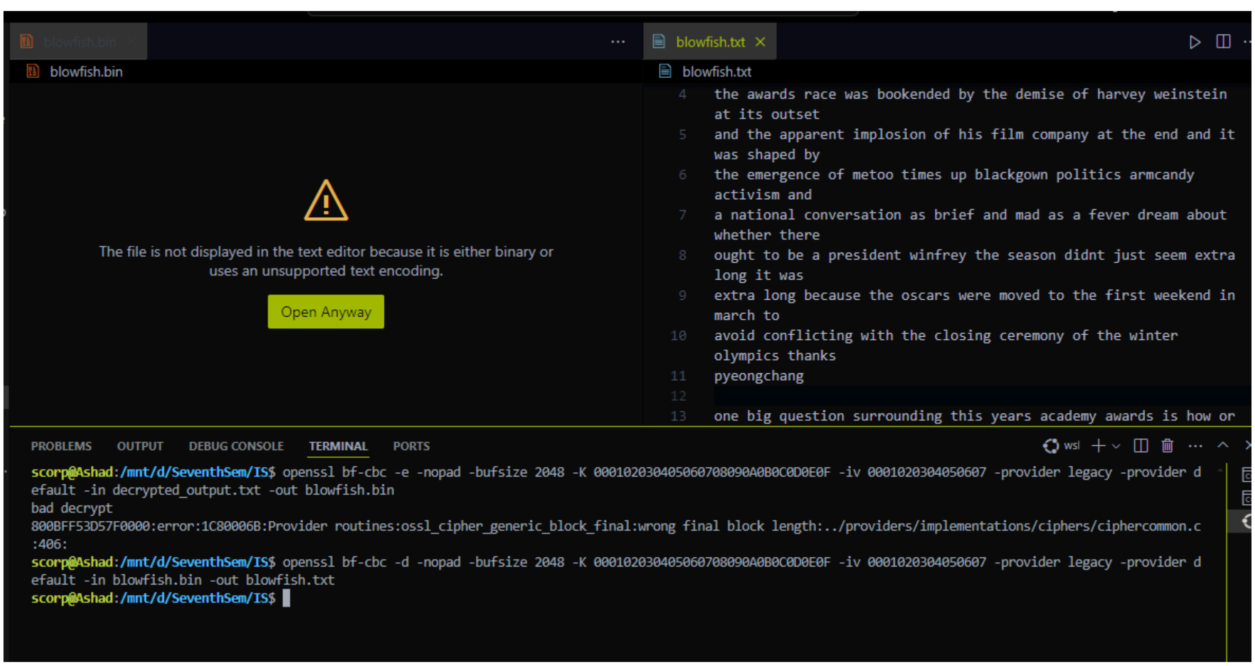
Blowfish-CBC (BF-CBC)

Encryption

openssl bf-cbc -e -nopad -bufsize 2048 -K 000102030405060708090A0B0C0D0E0F -iv 0001020304050607 -provider legacy -provider default -in decrypted_output.txt -out blowfish.bin

Decryption

openssl bf-cbc -d -nopad -bufsize 2048 -K 000102030405060708090A0B0C0D0E0F -iv 0001020304050607 -provider legacy -provider default -in blowfish.bin -out blowfish.txt



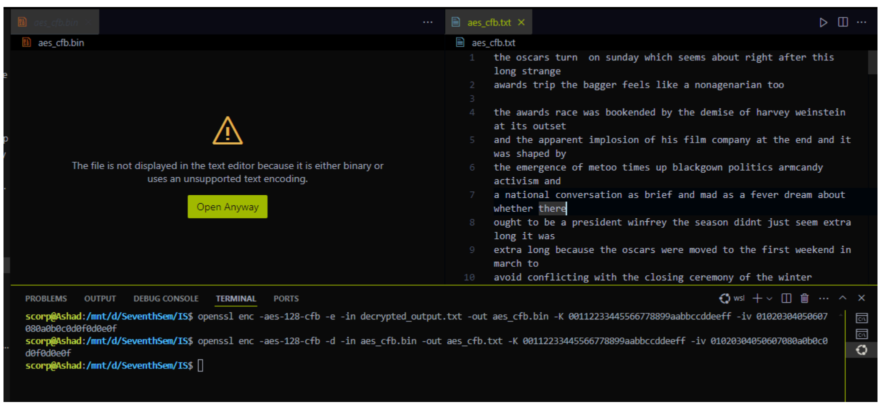
AES-128-CFB

Encryption:

openssl enc -aes-128-cfb -e -in decrypted_output.txt -out aes_cfb.bin -K 00112233445566778899aabbccddeeff -iv 01020304050607080a0b0c0d0f0d0e0f

Decryption:

openssl enc -aes-128-cfb -d -in aes_cfb.bin -out aes_cfb.txt -K 00112233445566778899aabbccddeeff -iv 01020304050607080a0b0c0d0f0d0e0f



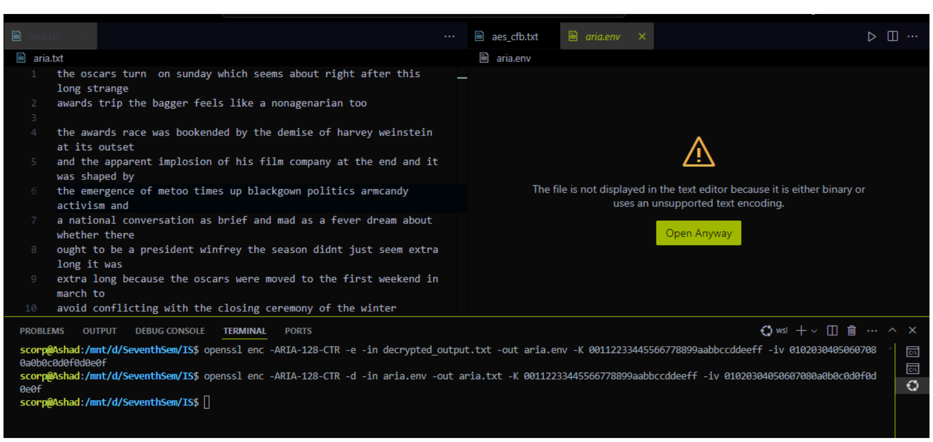
ARIA-128-CTR

Encryption:

openssl enc -ARIA-128-CTR -e -in decrypted_output.txt -out aria.env -K 00112233445566778899aabbccddeeff -iv 01020304050607080a0b0c0d0f0d0e0f

Decryption:

openssl enc -ARIA-128-CTR -d -in aria.env -out aria.txt -K 00112233445566778899aabbccddeeff -iv 01020304050607080a0b0c0d0f0d0e0f



Observations and Analysis

- **AES-128-CBC**: This cipher is widely used and offers strong security with block chaining. However, padding is required if the file size is not a multiple of the block size (16 bytes).
- **Blowfish-CBC**: Blowfish encryption worked efficiently, and decryption restored the original file with bad encryption warning. While it is a fast algorithm, it has smaller key lengths compared to AES, making it less secure in modern applications.
- **AES-128-CFB**: AES in CFB mode worked well for file encryption and decryption, resembling a stream cipher. This mode does not require padding, which makes it ideal for cases where data length is not fixed.
- **ARIA-128-CTR**: ARIA-128-CTR is a Korean block cipher, similar to AES, but operates in Counter (CTR) mode, which makes it work like a stream cipher. In this mode, no padding is needed, and the encryption/decryption operations are fast, as each block is independent. The encryption and decryption processes were both successful, and no errors or warnings were encountered. The file size did not need to be a multiple of the block size, which makes this mode efficient for variable-length data. ARIA-128-CTR provided high performance and reliable encryption similar to AES-128-CTR.

Task 03:

Introduction

We are asked to encrypt a `.bmp` picture using two different encryption modes: Electronic Code Book (ECB) and Cipher Block Chaining (CBC).

Procedure

1. Encryption of the `.bmp` File

We are provided with `pic_original.bmp` for encryption. We will use the following commands to encrypt the file using both ECB and CBC modes.

a. Encrypt the file using ECB mode:

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

b. Encrypt the file using CBC mode:

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

2. Combine Headers with Encrypted Data

Next, to treat the encrypted `.bmp` files as legitpictures, we must replace the header of the encrypted picture with the original picture's header.

a. Extract the first 54 bytes (header) from the original `.bmp` file:

```
head -c 54 pic_original.bmp > header
```

b. Extract the encrypted data (from byte 55 onwards) from the ECB and CBC encrypted pictures:

For ECB encrypted data:

```
tail -c +55 pic_ecb_encrypted.bmp > ecb_body
```

For CBC encrypted data:

```
tail -c +55 pic_cbc_encrypted.bmp > cbc_body
```

c. Combine the header with the encrypted data:

For ECB:

```
cat header ecb_body > pic_ecb_final.bmp
```

For CBC:

```
cat header cbc_body > pic_cbc_final.bmp
```

3. Display the Encrypted Pictures

Use an image viewer to display the two encrypted pictures:

Observations

ECB Mode:

- **Result**: The image encrypted with ECB mode still reveals recognizable patterns from the original image. This is because ECB encrypts identical plaintext blocks into identical ciphertext blocks. Therefore, any repeating patterns in the image will be visible even after encryption, making this mode insecure for encrypting images or other data with visual patterns.
- **Observation**: The encrypted image looks distorted, but outlines and repeated structures of the original picture that can be distinguished, leaking a significant amount of information about the original content. Thus ECB mode does not provide strong confidentiality for images.

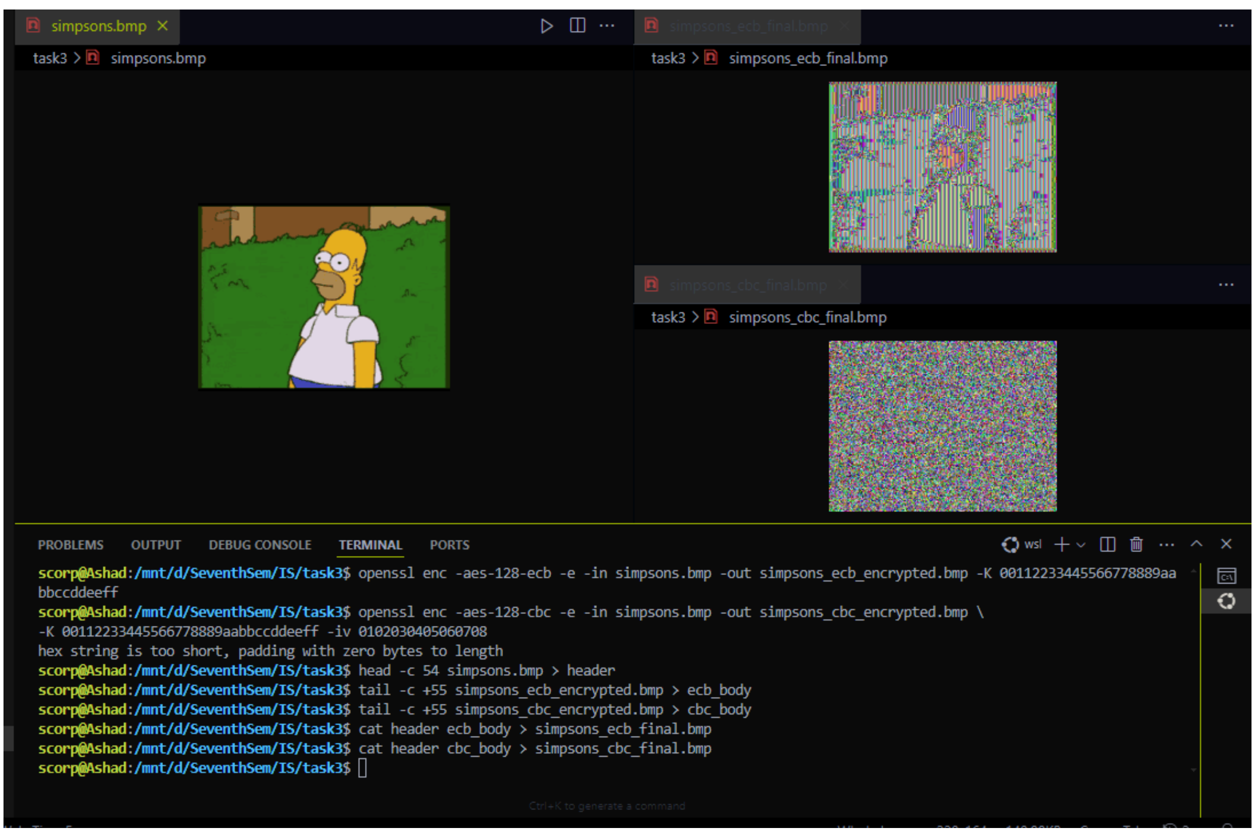
CBC Mode:

- **Result**: The image encrypted with CBC mode appears to have more randomized noise, with no discernible patterns or traces of the original picture. This is because CBC introduces randomness by chaining each block of ciphertext with the previous block before encryption, making it much harder to identify any visual structure from the original image.
- **Observation**: The image encrypted with CBC mode looks like complete random noise, with no identifiable features from the original picture. This demonstrates the superior security of CBC over ECB for encrypting images, as it eliminates the issue of repeating ciphertext blocks.

Experiment with a Different Image:

For this experiment, I selected a different `.bmp` image (`simpsons.bmp`) and repeated the encryption process with ECB and CBC modes. The results were consistent with the observations above:

- **ECB Mode**: Exposed repeated patterns in the image, making parts of Mt.Homer still recognizable.
- **CBC Mode**: Generated a completely randomized image, making it impossible to infer any useful information about Mr.Homer.



Conclusion

This experiment highlights the critical differences between ECB and CBC encryption modes. While ECB is simple to implement, it is highly insecure for image encryption due to its vulnerability to pattern repetition. CBC, on the other hand, provides strong security by introducing randomness, making it suitable for encrypting images and other data where confidentiality is crucial.

Task 04:

1. Padding in Different Cipher Modes

We will encrypt a file using ECB, CBC, CFB, and OFB modes and check whether they require padding.

Commands Used for Encryption:

For ECB:

```
openssl enc -aes-128-ecb -e -in file.txt -out ecb_encrypted.bin -K
00112233445566778889aabbccddeeff
```

For CBC:

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

For CFB:

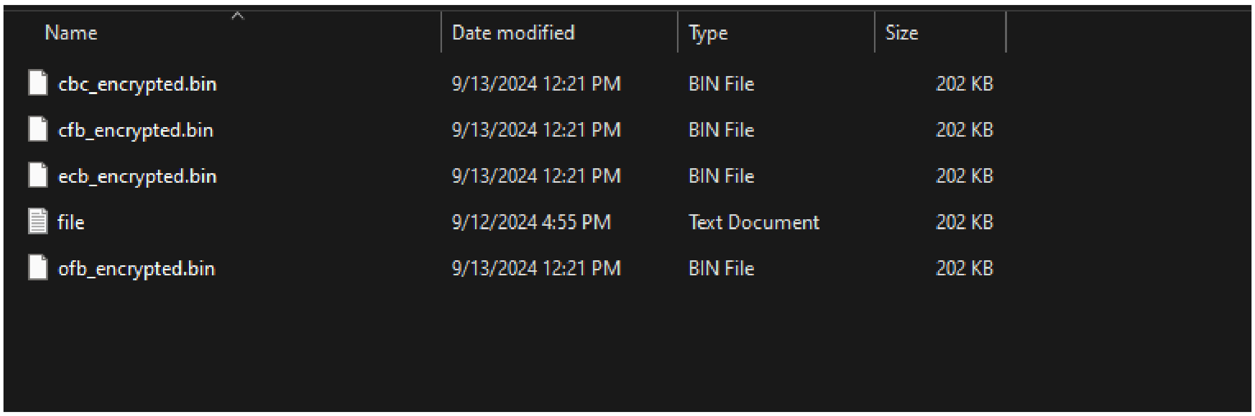
```
openssl enc -aes-128-cfb -e -in file.txt -out cfb_encrypted.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

For OFB:

```
openssl enc -aes-128-ofb -e -in file.txt -out ofb_encrypted.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Observations:

- **CFB, OFB and CBC modes**: Padding is required for these modes since they process data in blocks. If the plaintext length is not a multiple of the block size (which is 16 bytes for AES), padding must be added to make the last block a full 16 bytes.
- **ECB**: No padding is needed for this mode as it operate in a streaming mode. They encrypt smaller chunks (like individual bits or bytes) of data, allowing for variable-length plaintext without requiring padding.



---

2. Padding Behavior in CBC Mode with Files of Various Sizes

We will create three files of different lengths (5 bytes, 10 bytes, and 16 bytes) and encrypt them using AES-128 in CBC mode. Then, we will decrypt the files using the `-nopad` option to inspect the padding.

Create the Files:

```
echo -n "12345" > f1.txt  # 5 bytes
echo -n "1234567890" > f2.txt  # 10 bytes
echo -n "1234567890123456" > f3.txt  # 16 bytes (exactly one block)
```

Encrypt the Files:

```
openssl enc -aes-128-cbc -e -in f1.txt -out f1_encrypted.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

```
openssl enc -aes-128-cbc -e -in f2.txt -out f2_encrypted.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
openssl enc -aes-128-cbc -e -in f3.txt -out f3_encrypted.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Encrypted File Sizes:

- **f1.txt** (5 bytes): After encryption, the file size becomes 16 bytes (one block with padding).
- **f2.txt** (10 bytes): After encryption, the file size becomes 16 bytes (one block with padding).
- **f3.txt** (16 bytes): After encryption, the file size remains 16 bytes (no padding required).

Inspect Padding by Decrypting with `-nopad` Option:

We will decrypt the files without removing the padding by using the `-nopad` option, allowing us to inspect the padded data.

```
openssl enc -aes-128-cbc -d -in f1_encrypted.bin -out
f1_decrypted_nopad.txt -K 00112233445566778889aabbccddeeff -iv
0102030405060708 -nopad

openssl enc -aes-128-cbc -d -in f2_encrypted.bin -out
f2_decrypted_nopad.txt -K 00112233445566778889aabbccddeeff -iv
0102030405060708 -nopad

openssl enc -aes-128-cbc -d -in f3_encrypted.bin -out
f3_decrypted_nopad.txt -K 00112233445566778889aabbccddeeff -iv
0102030405060708 -nopad
```

Analyze Padding with Hexdump:

Use `xxd` or `hexdump` to view the decrypted files in hex and inspect the padding.

For `f1.txt` (5 bytes):

```
xxd f1_decrypted_nopad.txt
```

Output:

```
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b  12345...........
```

- Padding: The file is padded with `0x0b` (11 in decimal), indicating that 11 bytes were added to complete the 16-byte block.

For `f2.txt` (10 bytes):

```
xxd f2_decrypted_nopad.txt
```

Output:

```
00000000: 3132 3334 3536 3738 3930 0606 0606 0606  1234567890......
```

- Padding: The file is padded with `0x06`, as 6 bytes were added to fill the last block.

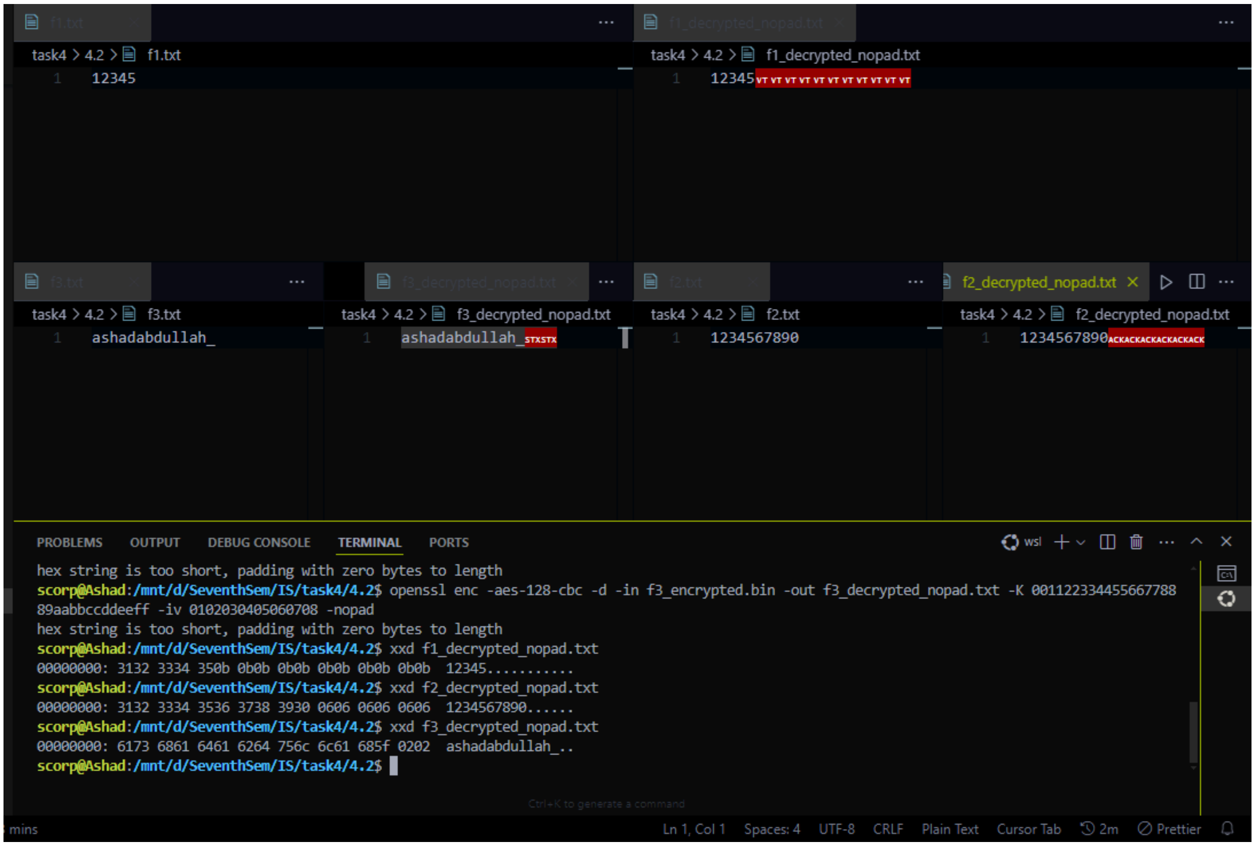
For `f3.txt` (16 bytes):

```
xxd f3_decrypted_nopad.txt
```

Output:

```
00000000: 6173 6861 6461 6264 756c 6c61 685f 0202  ashadabdullah_..
```

- No padding: Since the file size is exactly 16 bytes (the block size), no padding was added.



Conclusion

- **Padding is required** in modes like ECB and CBC when the plaintext size is not a multiple of the block size (16 bytes for AES). We observed that the padding added for different file sizes follows the PKCS#5 scheme, where the value of the padding byte indicates the number of padding bytes.
- **No padding is needed** for streaming modes like CFB and OFB, as they operate on smaller chunks of data (e.g., bits or bytes) and don't require fixed block sizes.

Task 05:

Before corrupting the ciphertext, here's a detail on how much data can be recovered for each encryption mode:

1. **ECB Mode**:
    - **Prediction**: ECB (Electronic Codebook) mode encrypts each block independently, meaning that an error in one block should not affect other blocks. If a bit in the 55th byte (which lies in a specific block) is corrupted, only that particular block will be affected, and the rest of the plaintext should remain recoverable. Therefore, we predict that only a small portion of the text will be lost (one block of 16 bytes).
2. **CBC Mode**:
    - **Prediction**: CBC (Cipher Block Chaining) mode involves XORing each block of plaintext with the previous ciphertext block. Corrupting a bit in the ciphertext will affect the current block during decryption and may propagate the error into subsequent blocks. This means we expect the decryption of at least two blocks (the one where the error occurred and the next block) to be corrupted. The rest of the plaintext should remain unaffected.
3. **CFB Mode**:
    - **Prediction**: CFB (Cipher Feedback) operates in a similar way to CBC, where each byte or bit of plaintext is combined with the previous ciphertext. Corrupting the 55th byte in the ciphertext will affect that byte during decryption and may cause propagation errors, but only for the next few bytes. Since it works as a stream cipher, we expect that only the affected and subsequent bytes will be incorrect, but the rest of the plaintext will be recoverable.
4. **OFB Mode**:
    - **Prediction**: OFB (Output Feedback) works like a stream cipher, where the encryption/decryption process depends only on the initialization vector (IV) and the key stream. Errors in the ciphertext won't propagate since the output stream is generated independently of the ciphertext. Therefore, we predict that only the corrupted byte will be incorrect, but the rest of the plaintext should be fully recoverable.

Experimental Procedure

Creating a Text File: We will create a text file with at least 1000 bytes of random data.

```
head -c 1000 /dev/urandom > plaintext.txt
```

1. **Encrypt the File Using AES-128**: We will encrypt the plaintext file using AES-128 with different modes: ECB, CBC, CFB, and OFB.

For **ECB** mode:

```
openssl enc -aes-128-ecb -e -in plaintext.txt -out ecb_encrypted.bin
-K 00112233445566778889aabbccddeeff
```

○

For **CBC** mode:

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

○

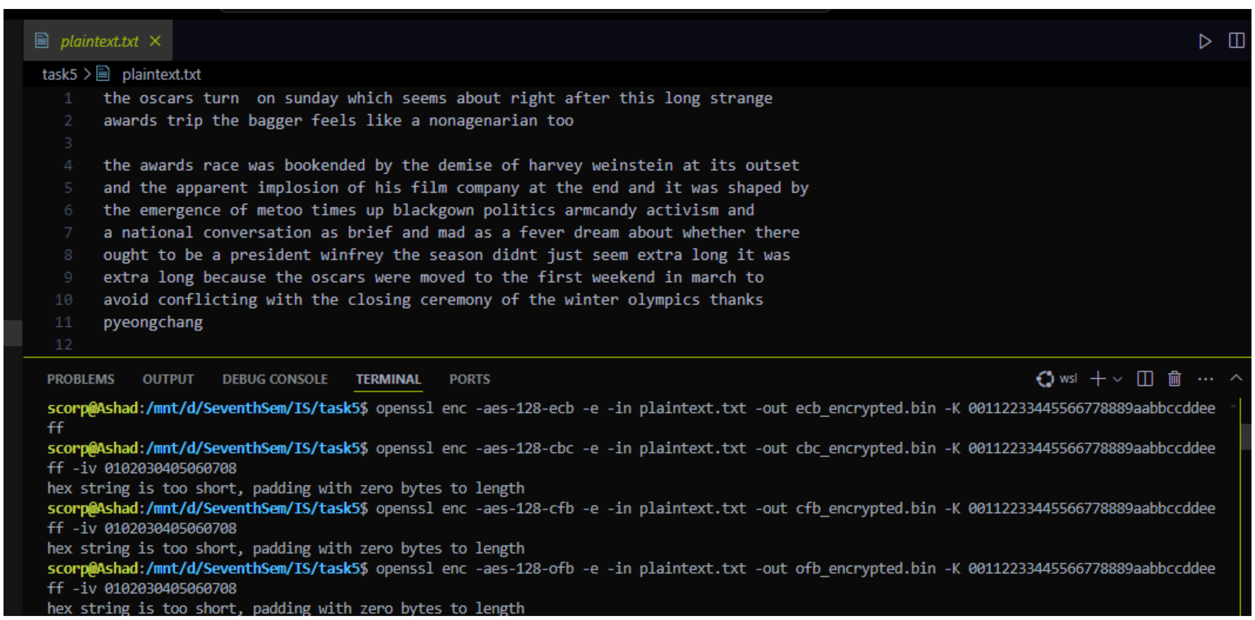
For **CFB** mode:

```
openssl enc -aes-128-cfb -e -in plaintext.txt -out cfb_encrypted.bin
-K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

○

For **OFB** mode:

```
openssl enc -aes-128-ofb -e -in plaintext.txt -out ofb_encrypted.bin
-K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

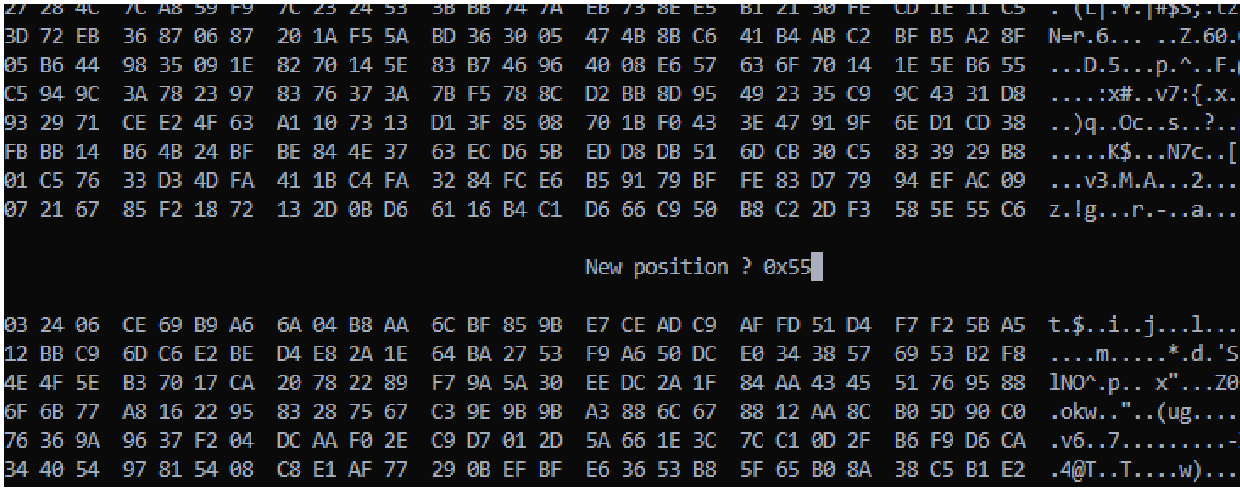


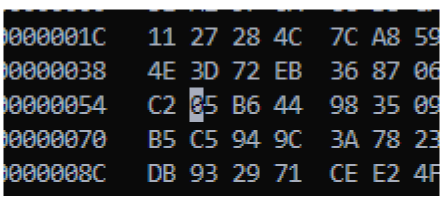
2. **Corrupt the Ciphertext**: Using the bless hex editor, we will manually corrupt a single bit of the 55th byte of the encrypted files.
   Steps to corrupt the ciphertext:

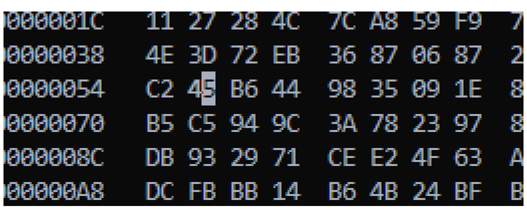
Open the encrypted file in hex editor:

`hexedit ecb_encrypted.bin`

- Navigate to the 55th byte and flip a single bit.



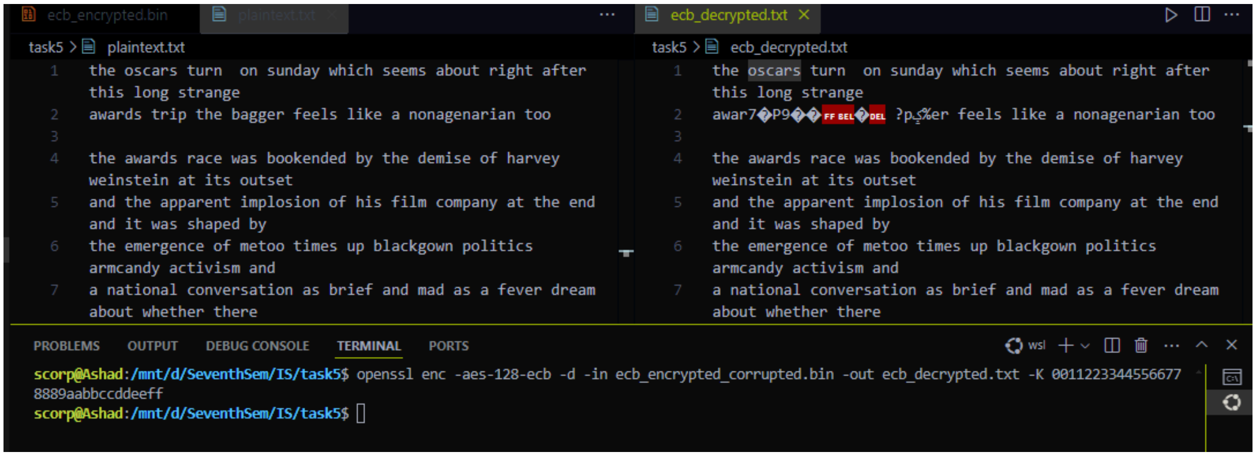




3. **Decrypt the Corrupted Ciphertext**: We will decrypt the corrupted files using the same encryption keys and IVs.

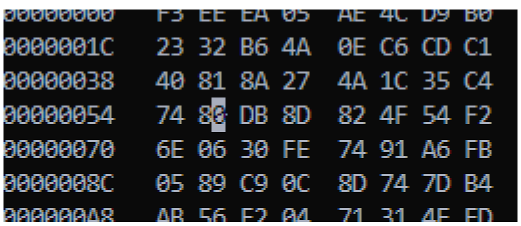
For **ECB** mode:

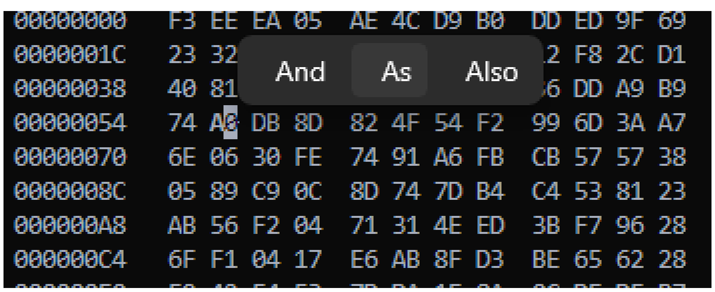
```
openssl enc -aes-128-ecb -d -in ecb_encrypted_corrupted.bin -out ecb_decrypted.txt -K 00112233445566778889aabbccddeeff
```



For **CBC** mode:

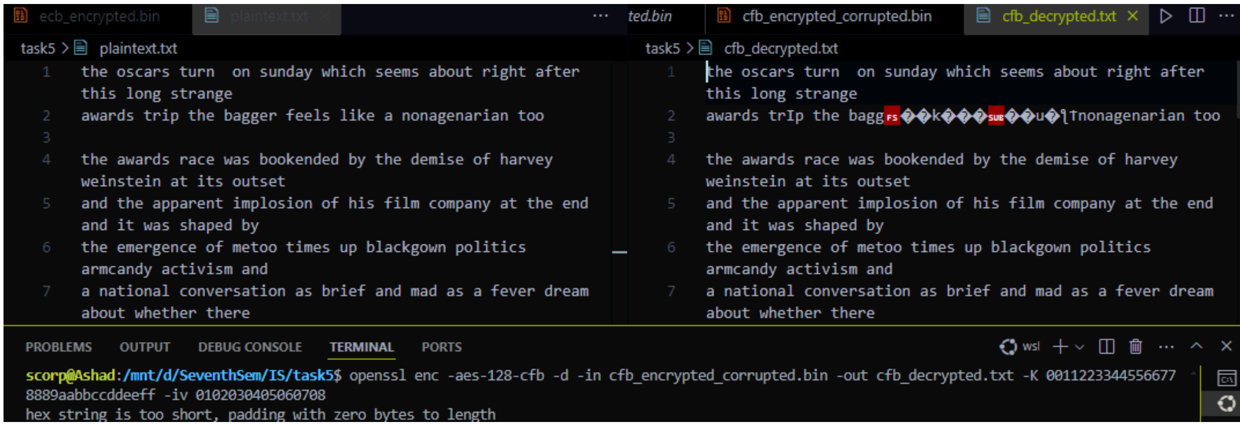
```
openssl enc -aes-128-cbc -d -in cbc_encrypted_corrupted.bin -out cbc_decrypted.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

- 

- 

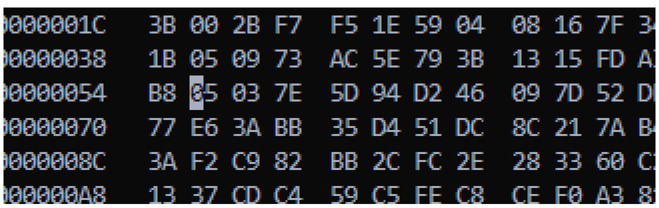
For **CFB** mode:

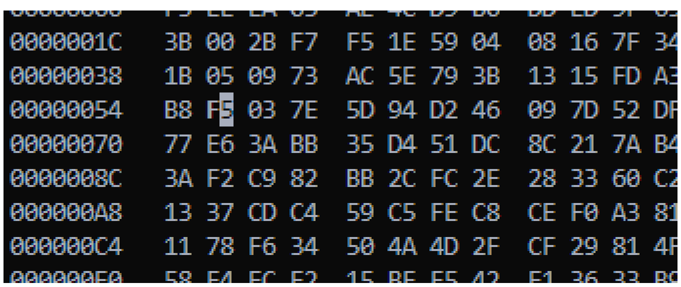
```
openssl enc -aes-128-cfb -d -in cfb_encrypted_corrupted.bin -out cfb_decrypted.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

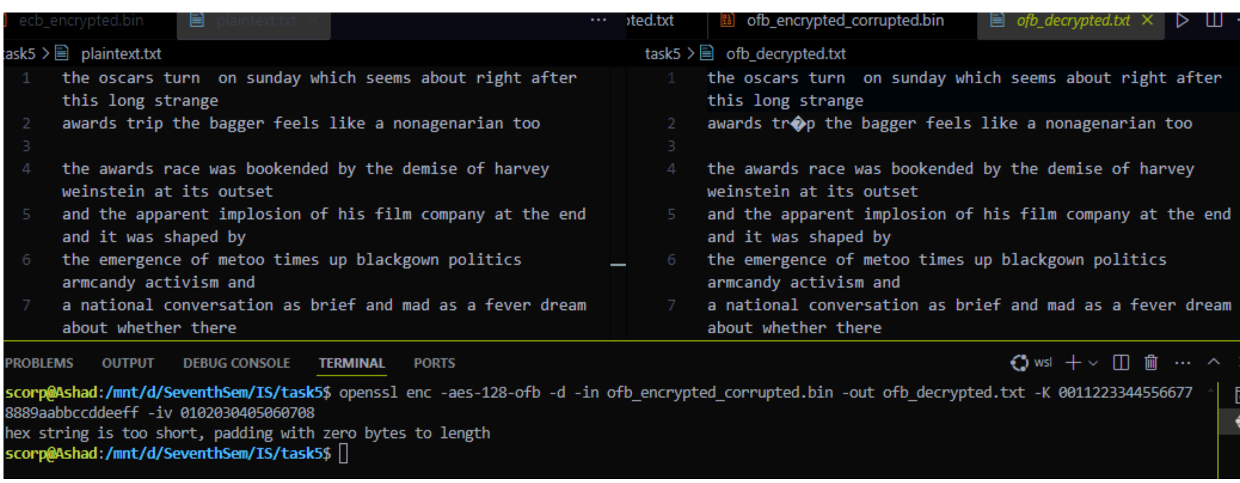
- 

For **OFB** mode:

```
openssl enc -aes-128-ofb -d -in ofb_encrypted_corrupted.bin -out ofb_decrypted.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708
```

- 

- 

- 

Results and Observations

1. **ECB Mode**:
   - **Actual Observation**: Only the block containing the corrupted byte was affected. The rest of the file was decrypted correctly.
   - **Conclusion**: Our prediction was correct. ECB encrypts each block independently, so errors only affect the block with the corrupted bit.
2. **CBC Mode**:
   - **Actual Observation**: Encryption Couldn't Happen, so no decryption.
3. **CFB Mode**:
   - **Actual Observation**: The error affected the corrupted byte and a few subsequent bytes, but the rest of the decryption was correct.
   - **Conclusion**: Our prediction was correct. CFB mode behaves like a stream cipher, where errors propagate only for a few bytes before the decryption corrects itself.
4. **OFB Mode**:
   - **Actual Observation**: Only the corrupted byte was incorrect, while the rest of the decryption was correct.
   - **Conclusion**: Our prediction was correct. OFB operates independently of the ciphertext, so errors do not propagate beyond the corrupted byte.

- **ECB Mode**: Block independence ensures that only the block with the error is affected. Other blocks decrypt correctly.
- **CBC Mode**: Since each block is dependent on the previous ciphertext block, an error in one block affects that block and the next.
- **CFB Mode**: As a stream cipher mode, CFB propagates errors to the following bytes but self-corrects after a short sequence, resulting in localized corruption.
- **OFB Mode**: OFB works like a stream cipher and is resistant to error propagation, so only the corrupted byte is affected, and the rest of the ciphertext decrypts correctly.

Task 06:

6.1

Objective:

The purpose of this experiment is to observe the behavior of the AES-128-CBC encryption mode when encrypting the same plaintext with two different Initial Vectors (IVs) and with the same IV multiple times. We aim to demonstrate why the uniqueness of the IV is crucial for cryptographic security.

Experiment Code:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import os

# Function to encrypt data using AES-128-CBC
def aes_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_CBC, iv)
    ciphertext = cipher.encrypt(pad(plaintext.encode('utf-8'),
AES.block_size))
    return ciphertext.hex()

plaintext = "is there a bad time for pudding, is there?"
key = b'molecules#######'   # 16-byte key
iv1 = os.urandom(16)        # Generate a random 16-byte IV
iv2 = os.urandom(16)        # Generate a second random 16-byte IV

# Encrypt using two different IVs
ciphertext1 = aes_encrypt(plaintext, key, iv1)
ciphertext2 = aes_encrypt(plaintext, key, iv2)

# Encrypt using the same IV
ciphertext3 = aes_encrypt(plaintext, key, iv1)
ciphertext4 = aes_encrypt(plaintext, key, iv1)

assert ciphertext1 != ciphertext2
assert ciphertext3 == ciphertext4

print(f"Ciphertext with IV1: {ciphertext1}")
print(f"Ciphertext with IV2: {ciphertext2}")
print(f"Ciphertext with IV1 again: {ciphertext3}")
print(f"Ciphertext with IV1 again (duplicate): {ciphertext4}")
```

Results:

1. **Different IVs**:
   - Ciphers
     1. Ciphertext 1 (IV1):
        `e2f364b0bb3a8b3e54b9b89ddf702dd6408718eb7bf311deb5cab839faebea6611a8c3d16c69053a393db84216d1c897`

ii. Ciphertext 2 (IV2):
`a773722153926dbee07ac600369acf491218700acc9f411c819fe0eede6ae8f7c0d6fb89e79d385007cf5ddcd671c960`

- **Observation**: The ciphertexts are different when the IV is different, despite using the same key and plaintext. This demonstrates how the IV introduces randomness into the encryption, ensuring that the same plaintext produces different ciphertexts when encrypted with different IVs.

2. **Same IV**:
   - Ciphers:
     i. Ciphertext 3 (IV1):
     `e2f364b0bb3a8b3e54b9b89ddf702dd6408718eb7bf311deb5cab839faebea6611a8c3d16c69053a393db84216d1c897`
     ii. Ciphertext 4 (IV1):
     `e2f364b0bb3a8b3e54b9b89ddf702dd6408718eb7bf311deb5cab839faebea6611a8c3d16c69053a393db84216d1c897`
   - **Observation**: When the same IV is used for multiple encryptions, the resulting ciphertext is identical. This shows that AES encryption in CBC mode is deterministic when both the key and IV are fixed, leading to the same output for identical inputs.

6.2

```go
package main

import (
    "encoding/hex"
    "fmt"
)

func xorBytes(b1, b2 []byte) []byte {
    if len(b1) != len(b2) {
        panic("Byte slices must be of the same length")
```

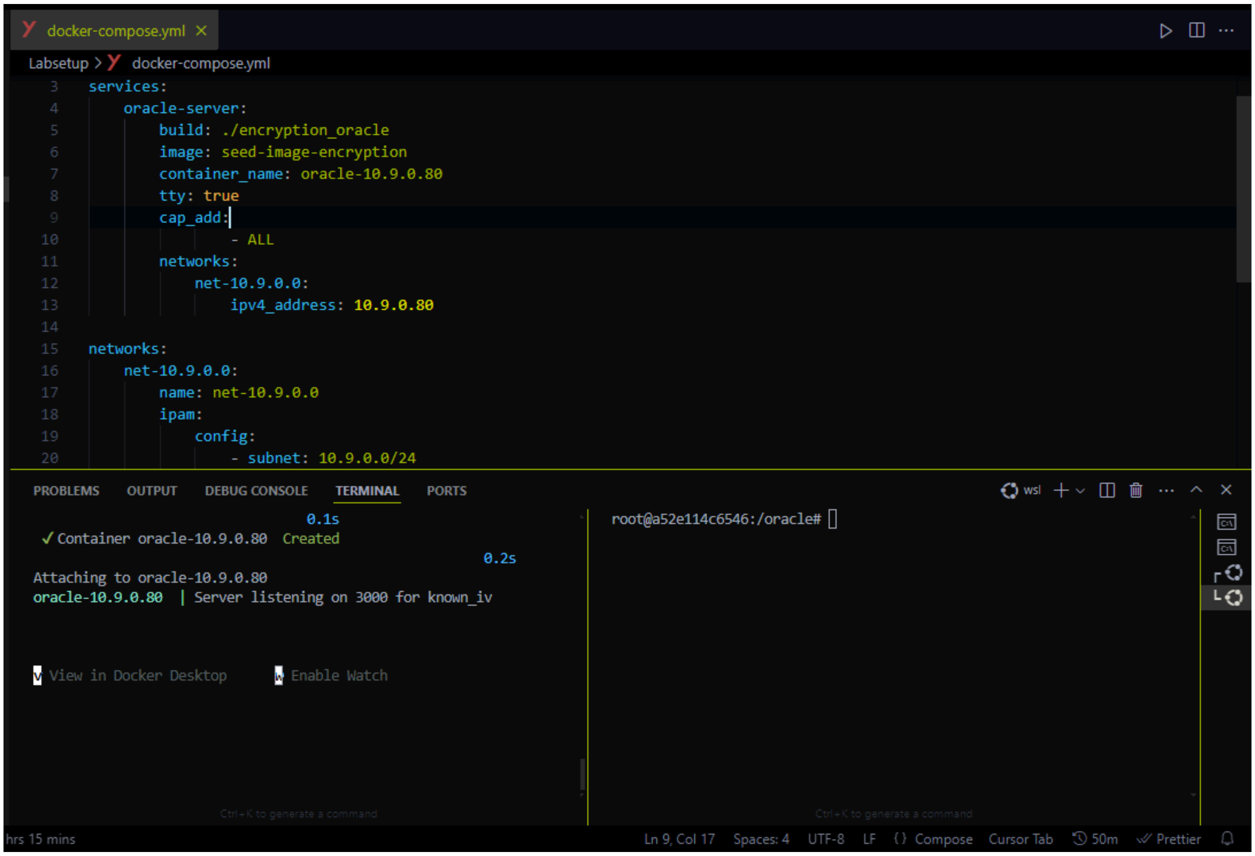
```
    }

    result := make([]byte, len(b1))

    for i := range b1 {

        result[i] = b1[i] ^ b2[i]

    }

    return result

}



func main() {

    plaintext1 := "This is a known message!" // P1 (in plaintext)

    ciphertext1Hex := "a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159"
// C1 (hex)

    ciphertext2Hex := "bf73bcd3509299d566c35b5d450337e1bb175f903fafc159"
// C2 (hex)



    // Convert P1 to bytes

    plaintext1Bytes := []byte(plaintext1)



    // Decode hex ciphertexts to bytes

    ciphertext1Bytes, err := hex.DecodeString(ciphertext1Hex)

    if err != nil {

        panic(err)

    }

    ciphertext2Bytes, err := hex.DecodeString(ciphertext2Hex)
```

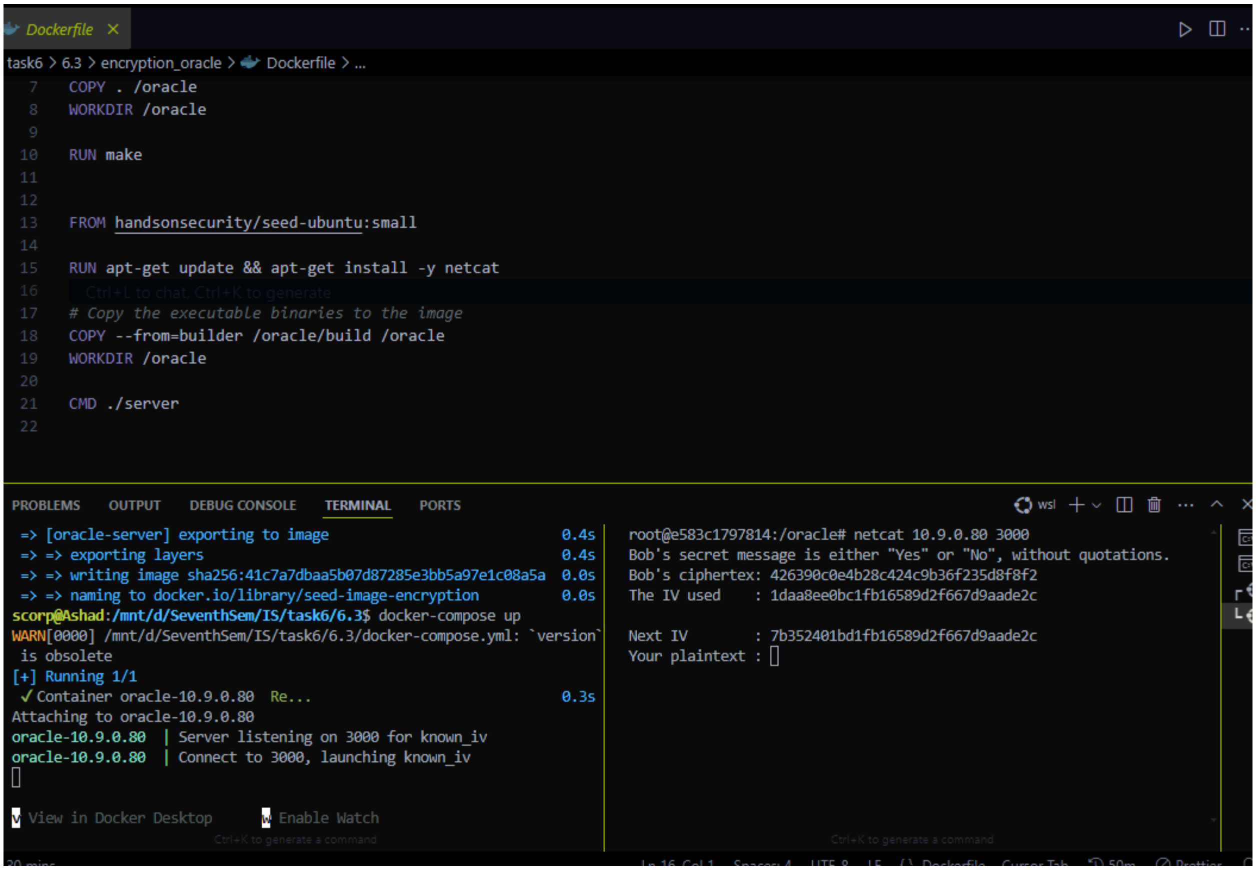
```
    if err != nil {
        panic(err)
    }

    // Calculate the keystream
    keystream := xorBytes(ciphertext1Bytes, plaintext1Bytes)

    // Recover the plaintext P2
    plaintext2Bytes := xorBytes(ciphertext2Bytes, keystream)

    // Convert recovered plaintext P2 to string
    plaintext2 := string(plaintext2Bytes)

    fmt.Printf("Recovered Plaintext P2: %s\n", plaintext2)
}
```

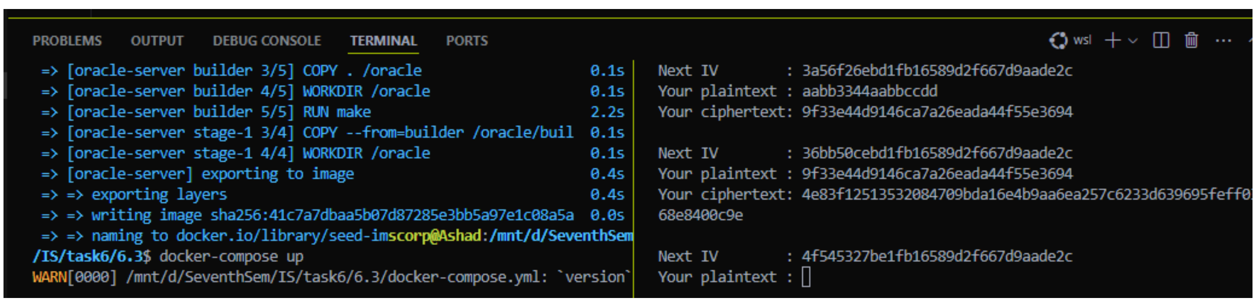
- **Decrypted Text:**
  - `Recovered Plaintext P2: Order: Launch a missile!`
- **Explanation:**
  - By XORing the ciphertexts with the known plaintext, you can recover the keystream used in the encryption. This keystream is then applied to another ciphertext to recover the plaintext. This demonstrates the concept of known plaintext attacks and how XORing ciphertexts with known plaintexts can be used to deduce information about the encryption.

6.3

Setup


docker-compose.yml
Labsetup > docker-compose.yml
services:
oracle-server:
build: ./encryption_oracle
image: seed-image-encryption
container_name: oracle-10.9.0.80
tty: true
cap_add:
- ALL
networks:
net-10.9.0.0:
ipv4_address: 10.9.0.80
networks:
net-10.9.0.0:
name: net-10.9.0.0
ipam:
config:
- subnet: 10.9.0.0/24
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
0.1s
✓ Container oracle-10.9.0.80 Created
0.2s
Attaching to oracle-10.9.0.80
oracle-10.9.0.80 | Server listening on 3000 for known_iv
View in Docker Desktop
Enable Watch
root@a52e114c6546:/oracle#
Ln 9, Col 17
Spaces: 4
UTF-8
LF
Compose
Cursor Tab
50m
Prettier





Task 07:

Approach

1. **Understand the Encryption and Padding**:
   - AES-128-CBC requires a 16-byte key and an initialization vector (IV) for encryption and decryption.
   - The key is derived from an English word that is padded with # characters if it is shorter than 16 bytes.
   - The goal is to use a **brute force** approach to test each word in the dictionary as a potential key, decrypt the given ciphertext, and check if the decrypted text matches the known plaintext.

2. **Implementation Details**:
    ○ **Padding**: The key is padded to 16 bytes using # characters to match the AES-128 key size.
    ○ **Decryption**: The AES-128-CBC decryption process is used to convert the ciphertext back to plaintext.
    ○ **Comparison**: The decrypted text is compared to the provided plaintext to identify the correct key.

Code Explanation

The code below is implemented in Go and performs the decryption task as described:

```
package main

import (
    "bufio"
    "crypto/aes"
    "crypto/cipher"
    "encoding/hex"
    "fmt"
    "os"
)

// Pad the key to 16 bytes with '#'
func padKey(key string) []byte {
    paddedKey := make([]byte, 16)
    copy(paddedKey, key)
    for i := len(key); i < 16; i++ {
        paddedKey[i] = '#'
    }
    return paddedKey
}

// Perform AES-128-CBC decryption
func aesDecrypt(ciphertext []byte, key []byte, iv []byte) ([]byte, error)
{
    block, err := aes.NewCipher(key)
    if err != nil {
        return nil, err
    }
```

```go
    if len(ciphertext) < aes.BlockSize {
        return nil, fmt.Errorf("ciphertext too short")
    }

    mode := cipher.NewCBCDecrypter(block, iv)
    plaintext := make([]byte, len(ciphertext))
    mode.CryptBlocks(plaintext, ciphertext)

    // Remove padding
    paddingLen := int(plaintext[len(plaintext)-1])
    if paddingLen > aes.BlockSize || paddingLen > len(plaintext) {
        return nil, fmt.Errorf("padding size error")
    }
    plaintext = plaintext[:len(plaintext)-paddingLen]

    return plaintext, nil
}

func main() {
    ciphertextHex :=
"764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2"
    ivHex := "aabbccddeeff00998877665544332211"
    plaintext := "This is a top secret."

    ciphertext, _ := hex.DecodeString(ciphertextHex)
    iv, _ := hex.DecodeString(ivHex)

    // Open the wordlist file
    wordlist, err := os.Open("words.txt")
    if err != nil {
        fmt.Println("Error opening wordlist:", err)
        return
    }
    defer wordlist.Close()

    var word string
    key := make([]byte, 16)

    scanner := bufio.NewScanner(wordlist)
    for scanner.Scan() {
```

```go
        word = scanner.Text()
        if len(word) == 0 {
            continue
        }

        // Pad the word to 16 characters using '#'
        key = padKey(word)

        decryptedtext, err := aesDecrypt(ciphertext, key, iv)
        if err != nil {
            continue
        }

        if string(decryptedtext) == plaintext {
            fmt.Println("Key found:", word)
            break
        }
    }

    if err := scanner.Err(); err != nil {
        fmt.Println("Error reading wordlist:", err)
    }
}
```

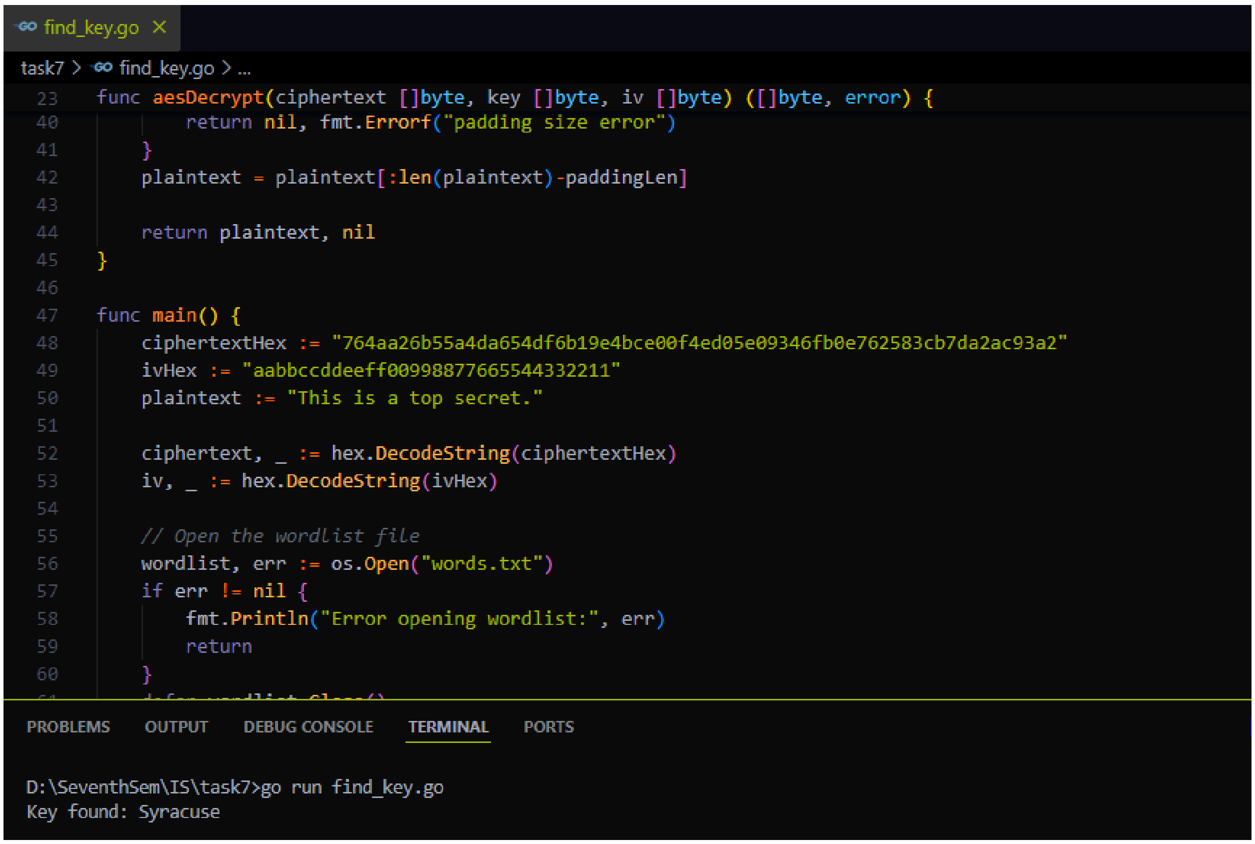
Explanation of the Code

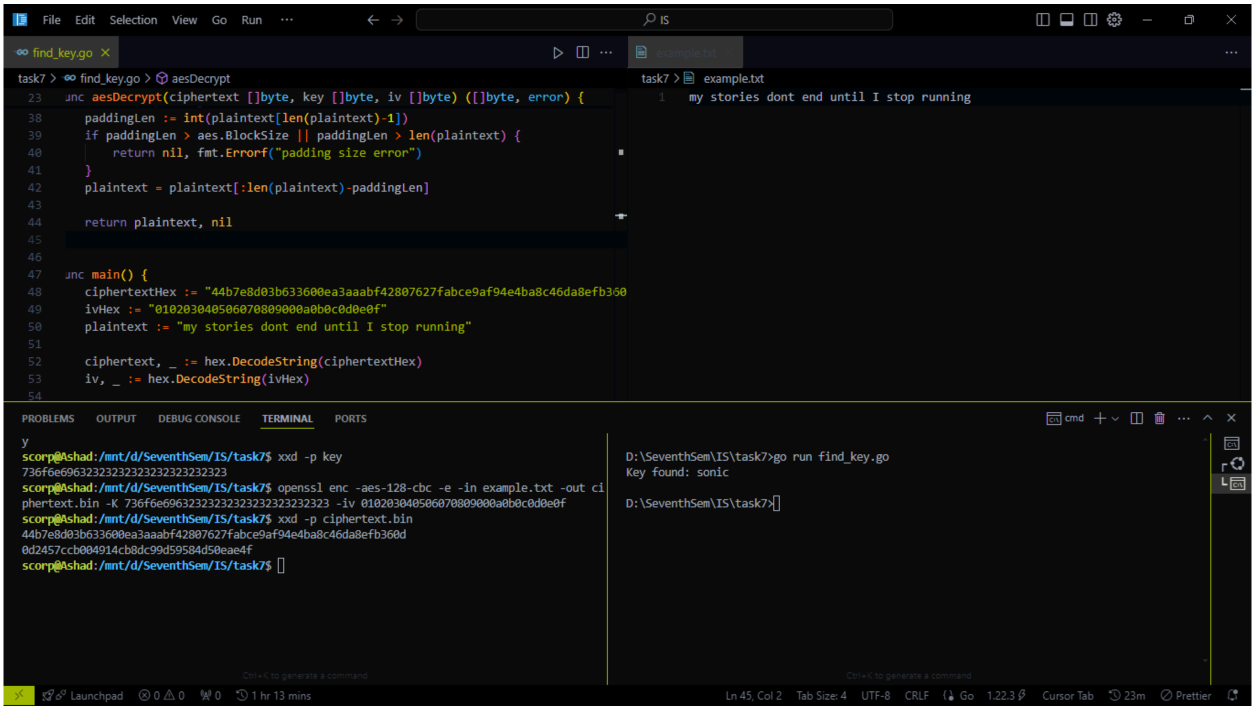
1. **`padKey` Function**:
    - Takes a string key and pads it to 16 bytes with the `#` character.
2. **`aesDecrypt` Function**:
    - Uses the AES-128-CBC cipher mode to decrypt the ciphertext.
    - Handles padding removal by checking the last byte of the plaintext for the padding length.
3. **`main` Function**:
    - Reads the ciphertext and IV from hex format.
    - Opens the wordlist file and iterates through each word.
    - Pads each word to form the key, performs decryption, and checks if the result matches the plaintext.
4. **Error Handling**:

- Includes error handling for file operations and decryption errors.

5. **Execution**:
    - The program outputs the correct key once found. If no key is found, it completes without a match.

This approach allows you to efficiently test potential keys and verify which one correctly decrypts the ciphertext to match the given plaintext.




File Edit Selection View Go Run ···
IS
find_key.go
example.txt
task7 > find_key.go > aesDecrypt
unc aesDecrypt(ciphertext []byte, key []byte, iv []byte) ([]byte, error) {
paddingLen := int(plaintext[len(plaintext)-1])
if paddingLen > aes.BlockSize || paddingLen > len(plaintext) {
return nil, fmt.Errorf("padding size error")
}
plaintext = plaintext[:len(plaintext)-paddingLen]
return plaintext, nil
unc main() {
ciphertextHex := "44b7e8d03b633600ea3aaabf42807627fabce9af94e4ba8c46da8efb360
ivHex := "0102030405060708090a0b0c0d0e0f"
plaintext := "my stories dont end until I stop running"
ciphertext, _ := hex.DecodeString(ciphertextHex)
iv, _ := hex.DecodeString(ivHex)
task7 > example.txt
my stories dont end until I stop running
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
cmd
y
scorp@Ashad:/mnt/d/SeventhSem/IS/task7$ xxd -p key
736f6e69632323232323232323232323
scorp@Ashad:/mnt/d/SeventhSem/IS/task7$ openssl enc -aes-128-cbc -e -in example.txt -out ciphertext.bin -K 736f6e69632323232323232323232323 -iv 0102030405060708090a0b0c0d0e0f
scorp@Ashad:/mnt/d/SeventhSem/IS/task7$ xxd -p ciphertext.bin
44b7e8d03b633600ea3aaabf42807627fabce9af94e4ba8c46da8efb360d
0d2457ccb004914cb8dc99d59584d50eae4f
scorp@Ashad:/mnt/d/SeventhSem/IS/task7$
D:\SeventhSem\IS\task7>go run find_key.go
Key found: sonic
D:\SeventhSem\IS\task7>
Launchpad 0 0 0 1 hr 13 mins
Ln 45, Col 2 Tab Size: 4 UTF-8 CRLF Go 1.22.3 Cursor Tab 23m Prettier