Symmetric Encryption Lab

Assignment 2

EDMUND LAWERH AMANOR Fundamentals of Info. Security, Fall 2024

This report is a documentation of my observation of the process while I simulated the symmetric encyption lab in the SEED virtual machine. I provided screenshots, from the various stages.

I commented out the steps I took and observations made directly in the terminal window, both before and after executing commands to make is easy to follow what was happening.

Task 1

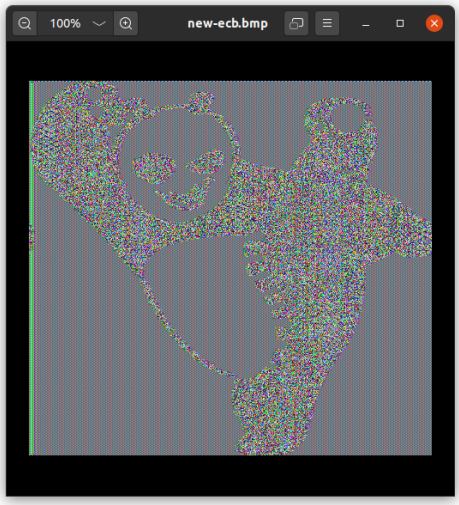
- The image below shows the basic encryption and decryption of simple plaintext file using a symmetric key.

```
[10/19/24]seed@VM:~/.../Lab_symmetric$ cd task1/
[10/19/24]seed@VM:~/.../task1$ openssl rand -hex 32 # for key
4a71c490305456404e4662825e2fb5fc94c104f9fa8623f6d390a79bc95e054f
[10/19/24]seed@VM:~/.../task1$ openssl rand -hex 16 # for iv
c4a0d815d9f8d3c114295a699c849bfb
[10/19/24]seed@VM:~/.../task1$ openssl enc -aes-256-cbc -e -in plain.txt -out cipher.bin \
> -K 4a71c490305456404e4662825e2fb5fc94c104f9fa8623f6d390a79bc95e054f \
> -iv c4a0d815d9f8d3c114295a699c849bfb
[10/19/24]seed@VM:~/.../task1$ ls
cipher.bin  plain.txt
[10/19/24]seed@VM:~/.../task1$ #now let's decrypt ciper.bin
[10/19/24]seed@VM:~/.../task1$
[10/19/24]seed@VM:~/.../task1$ # before that let's see what was encrypted
[10/19/24]seed@VM:~/.../task1$ cat plain.txt
My first hands-on experience with encryption and decryption.
[10/19/24]seed@VM:~/.../task1$
[10/19/24]seed@VM:~/.../task1$ openssl enc -aes-256-cbc -d -in cipher.bin -out decrypted.txt \
> -K 4a71c490305456404e4662825e2fb5fc94c104f9fa8623f6d390a79bc95e054f \
> -iv c4a0d815d9f8d3c114295a699c849bfb
[10/19/24]seed@VM:~/.../task1$ ls
cipher.bin  decrypted.txt  plain.txt
[10/19/24]seed@VM:~/.../task1$ # now let's see what we decrypted...
[10/19/24]seed@VM:~/.../task1$
[10/19/24]seed@VM:~/.../task1$ cat decrypted.txt
My first hands-on experience with encryption and decryption.
[10/19/24]seed@VM:~/.../task1$
[10/19/24]seed@VM:~/.../task1$
[10/19/24]seed@VM:~/.../task1$ # yay!! it matches what we encrypted
[10/19/24]seed@VM:~/.../task1$
[10/19/24]seed@VM:~/.../task1$ # task 1 completed
[10/19/24]seed@VM:~/.../task1$
```

Task 2

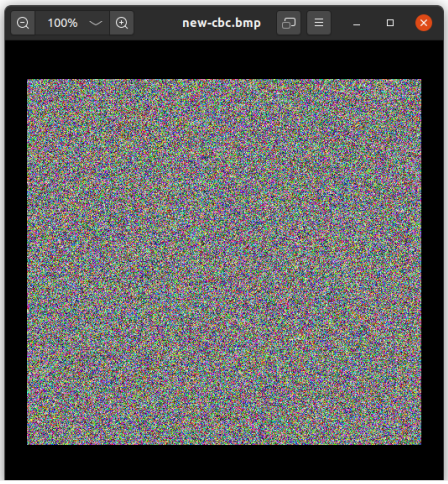
- Here, I show the encryption of an image using the ECB encyption mode and displayed the encrypted picture.

```
[10/20/24]seed@VM:~/.../task2$ # ECB Encryption
[10/20/24]seed@VM:~/.../task2$
[10/20/24]seed@VM:~/.../task2$ openssl enc -aes-192-ecb -e -in pic_original.bmp -out ecb_pic.enc -k task2paz -p
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
salt=5EA12787BCCDA235
key=EBEBC1D081A69ED601C540FDC123EF006F1145E1EB047556
[10/20/24]seed@VM:~/.../task2$
[10/20/24]seed@VM:~/.../task2$ head -c 54 pic_original.bmp > header
[10/20/24]seed@VM:~/.../task2$ tail -c +55 ecb_pic.enc > ecb_body
[10/20/24]seed@VM:~/.../task2$ cat header ecb_body > new-ecb.bmp
[10/20/24]seed@VM:~/.../task2$
[10/20/24]seed@VM:~/.../task2$ # let's view the ECB encrypted pic
[10/20/24]seed@VM:~/.../task2$
[10/20/24]seed@VM:~/.../task2$ eog new-ecb.bmp
```



- Next, the image shows how the same picture looks when encrypted with CBC mode.

```
[10/20/24]seed@VM:~/.../task2$ # CBC Encryption
[10/20/24]seed@VM:~/.../task2$
[10/20/24]seed@VM:~/.../task2$ openssl enc -aes-192-cbc -e -in pic_original.bmp -out cbc_pic.enc -k task2Bpas -p
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
salt=31CC4ED34E582942
key=9C01B05DD1BE887046D6C521978B011C5419DCE995919332
iv =CB410A126B7F6F5ED7243E463B316653
[10/20/24]seed@VM:~/.../task2$
[10/20/24]seed@VM:~/.../task2$ # since the already have the header, we just get the body
[10/20/24]seed@VM:~/.../task2$
[10/20/24]seed@VM:~/.../task2$ tail -c +55 cbc_pic.enc > cbc_body
[10/20/24]seed@VM:~/.../task2$ cat header cbc_body > new-cbc.bmp
[10/20/24]seed@VM:~/.../task2$
[10/20/24]seed@VM:~/.../task2$ # now let's view the cbc encrypted pic
[10/20/24]seed@VM:~/.../task2$
[10/20/24]seed@VM:~/.../task2$ eog new-cbc.bmp
```



Observation

From the two images displayed above, it's easy to tell what the original image looks like from the ECB encrypted file while we can't decipher anything from the CBC encrypted image. This observation simply aligns with the way the two encryption modes work.

- With ECB, each block of the plaintext is independently encrypted which means that if the same block appears multiple times, it will produce the same ciphertext block every time. This characteristic makes the repeating patterns and colors in the image evident in the ECB-encrypted image.
- CBC on ther other hand XORs each block with the previous ciphertext block before being encrypted. As a result, identical blocks will produce different ciphertext blocks, depending on the preceding ciphertext. This chaining effect in CBC mode obscures any patterns in the plaintext hence making our encrypted picture random unlike the ECB encrypted one which reveals some structure.

Task 3

For this task, I wrote the encryption and decryption scripts for CBC and OFB modes. Let's take a look at snippets of the code before executing them. **I made sure to provide relevant comments within the code snippets to avoid lengthy explanations here**.

cbc_aes.py | ofb_aes.py | compare.py | Task3.txt

compare.py > compare_with_bin_file

```python
def compare_with_bin_file(file_path, generated_ciphertext):
    """
    Compare the content of a binary file with the generated ciphertext.

    :param file_path: Path to the binary file to compare
    :param generated_ciphertext: The ciphertext generated from the encryption
    :return: True if the content matches, False otherwise
    """
    try:
        with open(file_path, 'rb') as f:
            expected_ciphertext = f.read()
            if expected_ciphertext == generated_ciphertext:
                print(f"\nOutput of encrypting the plaintext matches {file_path}!")
                return True
            else:
                print(f"Ciphertext does NOT match for {file_path}.")
                return False
    except FileNotFoundError:
        print(f"File {file_path} not found.")
        return False
```

```python
from Crypto.Random import get_random_bytes
import binascii

# import compare function
from compare import compare_with_bin_file

# Hardcoded keys and IV (hex strings converted to bytes)
key_128_hardcoded = binascii.unhexlify('ffeeddccbbaa99887766554433221100')  # AES-128 key
key_256_hardcoded = binascii.unhexlify('ffeeddccbbaa99887766554433221100ffeeddccbbaa99887766554433221100')  # AES-256 key
iv_hardcoded = binascii.unhexlify('0f0e0d0c0b0a09080f0e0d0c0b0a0908')  # IV for CBC mode

# Function for AES CBC encryption
def aes_cbc_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_CBC, iv)  # Create AES object in CBC mode
    padded_plaintext = pad(plaintext, AES.block_size)  # Pad plaintext
    ciphertext = cipher.encrypt(padded_plaintext)  # Encrypt padded plaintext
    return ciphertext

# Function for AES CBC decryption
def aes_cbc_decrypt(ciphertext, key, iv):
    cipher = AES.new(key, AES.MODE_CBC, iv)  # Create AES object in CBC mode
    decrypted_padded = cipher.decrypt(ciphertext)  # Decrypt ciphertext
    plaintext = unpad(decrypted_padded, AES.block_size)  # Unpad the plaintext
    return plaintext

# Set up argument parsing
parser = argparse.ArgumentParser(description='AES CBC Encryption with optional random key generation.')
parser.add_argument('--random', action='store_true', help='Use randomly generated key and IV instead of hardcoded ones')

```

```python
args = parser.parse_args()

# If --random is specified, generate random key and IV and read plaintext from 'my_cbc_plaintext.txt'
if args.random:
    key_256 = get_random_bytes(32)  # AES-256: 32 bytes
    iv = get_random_bytes(16)       # IV: 16 bytes for AES block size
    print("Using random AES-256 key and IV.\n")
    print(f"Generated AES-256 Key (hex): {binascii.hexlify(key_256).decode()}")
    print(f"Generated IV (hex): {binascii.hexlify(iv).decode()}")

    # Read the plaintext from 'my_cbc_plaintext.txt'
    with open('my_cbc_plaintext.txt', 'rb') as f:
        plaintext = f.read()
else:
    # Use hardcoded key and IV and read plaintext from 'AES_CBC_Plaintext1.txt'
    key_256 = key_256_hardcoded
    iv = iv_hardcoded
    print("\nUsing hardcoded AES-256 key and IV.")

    # Read the plaintext from 'AES_CBC_Plaintext1.txt'
    with open('AES_CBC_Plaintext1.txt', 'rb') as f:
        plaintext = f.read()

```

```python

# --- CBC Encryption ---
print("\n--- AES CBC Mode ---")
encrypted_text_cbc = aes_cbc_encrypt(plaintext, key_256, iv)  # Using AES-256

# Convert the encrypted text to hex for readability
hex_encrypted_text_cbc = binascii.hexlify(encrypted_text_cbc).decode()

# If using hardcoded keys, compare the generated ciphertext with the expected ciphertext from the lab folder
if not args.random:
    compare_with_bin_file("AES_CBC_Ciphertext256.bin", encrypted_text_cbc)

# Print the encrypted text in hex format
print(f"\nEncrypted (CBC, hex): {hex_encrypted_text_cbc}")

# --- CBC Decryption ---
decrypted_text_cbc = aes_cbc_decrypt(encrypted_text_cbc, key_256, iv)

# Print the decrypted text
print(f"\nDecrypted (CBC): {decrypted_text_cbc.decode()}")

```

```python
import argparse
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
import binascii

# import compare function
from compare import compare_with_bin_file

# Hardcoded keys and nonce (hex strings converted to bytes)
key_128_hardcoded = binascii.unhexlify('ffeeddccbbaa99887766554433221100')  # AES-128 key
key_256_hardcoded = binascii.unhexlify('ffeeddccbbaa99887766554433221100ffeeddccbbaa99887766554433221100')  # AES-256 key
nonce_hardcoded = binascii.unhexlify('0f0e0d0c0b0a09080f0e0d0c0b0a0908')  # Nonce for OFB mode

# Function for AES OFB encryption
def aes_ofb_encrypt(plaintext, key, nonce):
    cipher = AES.new(key, AES.MODE_OFB, nonce)  # Create AES object in OFB mode
    ciphertext = cipher.encrypt(plaintext)  # Encrypt plaintext (no padding needed)
    return ciphertext

# Function for AES OFB decryption
def aes_ofb_decrypt(ciphertext, key, nonce):
    cipher = AES.new(key, AES.MODE_OFB, nonce)  # Create AES object in OFB mode
    plaintext = cipher.decrypt(ciphertext)  # Decrypt ciphertext
    return plaintext

# Set up argument parsing
parser = argparse.ArgumentParser(description='AES OFB Encryption with optional random key and nonce generation.')
parser.add_argument('--random', action='store_true', help='Use randomly generated key and nonce instead of hardcoded ones')

```

```python

args = parser.parse_args()

# If --random is specified, generate random key and nonce and read plaintext from 'my_ofb_plaintext.txt'
if args.random:
    key_256 = get_random_bytes(32)  # AES-256: 32 bytes
    nonce = get_random_bytes(16)    # Nonce: 16 bytes for AES block size
    print("Using random AES-256 key and nonce.\n")
    print(f"Generated AES-256 Key (hex): {binascii.hexlify(key_256).decode()}")
    print(f"Generated Nonce (hex): {binascii.hexlify(nonce).decode()}")

    # Read the plaintext from 'my_ofb_plaintext.txt'
    with open('my_ofb_plaintext.txt', 'rb') as f:
        plaintext = f.read()
else:
    # Use hardcoded key and nonce and read plaintext from 'AES_OFB_plaintext1.txt'
    key_256 = key_256_hardcoded
    nonce = nonce_hardcoded
    print("\nUsing hardcoded AES-256 key and nonce.")

    # Read the plaintext from 'AES_OFB_plaintext1.txt'
    with open('AES_OFB_plaintext1.txt', 'rb') as f:
        plaintext = f.read()

```

```python

# --- OFB Encryption ---
print("\n--- AES OFB Mode ---")
encrypted_text_ofb = aes_ofb_encrypt(plaintext, key_256, nonce)  # Using AES-256

# Convert the encrypted text to hex for readability
hex_encrypted_text_ofb = binascii.hexlify(encrypted_text_ofb).decode()

# If using hardcoded keys, compare the generated ciphertext with the expected ciphertext
if not args.random:
    compare_with_bin_file("AES_OFB_Ciphertext256.bin", encrypted_text_ofb)

# Print the encrypted text in hex format
print(f"\nEncrypted (OFB, hex): {hex_encrypted_text_ofb}")

# --- OFB Decryption ---
decrypted_text_ofb = aes_ofb_decrypt(encrypted_text_ofb, key_256, nonce)

# Print the decrypted text
print(f"\nDecrypted (OFB): {decrypted_text_ofb.decode()}")

```

The snapshots below show the validation of the code using the example files provided.

```
[10/20/24]seed@VM:~/.../task3$ # Let's Validate using the files provided:
[10/20/24]seed@VM:~/.../task3$
[10/20/24]seed@VM:~/.../task3$ python cbc_aes.py

Using hardcoded AES-256 key and IV.

--- AES CBC Mode ---

Output of encrypting the plaintext matches AES_CBC_Ciphertext256.bin!

Encrypted (CBC, hex): 26083dc8d3c3f700fade6a41612953b2987f2c6d5af894ee6d1864edd19446bd8e114454b188294dd53
eaa48dc8ea941cba01fe64a7ff2e510834a3b24ee9bcb82628db176fdfc459c2cf89d7d619cd4

Decrypted (CBC): This is a sample text messsage. Please make sure to use correct padding.

[10/20/24]seed@VM:~/.../task3$
[10/20/24]seed@VM:~/.../task3$ python ofb_aes.py

Using hardcoded AES-256 key and nonce.

--- AES OFB Mode ---

Output of encrypting the plaintext matches AES_OFB_Ciphertext256.bin!

Encrypted (OFB, hex): 7643425182fb740d2b8439bf4bb008a819b7deda3d6dd1c3ee6b647194f0dd

Decrypted (OFB): This is just a test plaintext.

[10/20/24]seed@VM:~/.../task3$
```

- Now, I tested the code using my own choice of plaintext, key, and IV(or Nonce). All codes and files can be found in the /task3 directory of my submission.

```
[10/20/24]seed@VM:~/.../task3$ # now let's test using my own key, iv and plaintext
[10/20/24]seed@VM:~/.../task3$
[10/20/24]seed@VM:~/.../task3$ python cbc_aes.py --random
Using random AES-256 key and IV.

Generated AES-256 Key (hex): fbbf56247ae195970bb3e8d295505e59f7678e5d806c5ae80b7f0704bc8f0c38
Generated IV (hex): b5ebaa62c7a5c92729dd5d191b85f8b0

--- AES CBC Mode ---

Encrypted (CBC, hex): 40609abc290ebf03f285be13f05d72914df9e41226ee4ee1496fbd3edf9d08bd1166393799debcbb838
08b64903bddf8

Decrypted (CBC): Practicing what you've learned is fun.
[10/20/24]seed@VM:~/.../task3$
[10/20/24]seed@VM:~/.../task3$
[10/20/24]seed@VM:~/.../task3$ python ofb_aes.py --random
Using random AES-256 key and nonce.

Generated AES-256 Key (hex): f1e62f840c87e39c7008cb44f6dbcaae14ac10d61920ca58f3226d820c757ae1
Generated Nonce (hex): aa800e5624c4aef62b7cab391c6feef5

--- AES OFB Mode ---

Encrypted (OFB, hex): 649e853b550e1bb11a028c545d34a6ee12d8a002408ca527c24f126608e25ebdf835e3d6c11cf124

Decrypted (OFB): I've learnt alot from performing task 3!
[10/20/24]seed@VM:~/.../task3$
```

Task 4

- In task 4, observed how different cipher modes (CBC, OFB, ECB and CFB) handle encryption and padding. For all the cipher modes considered, I used AES-128.
- I began by creating 3 different-sized text files which will later be used for encryption. The screenshot below shows the file creation process.

```
[10/20/24]seed@VM:~/.../task4$ # Creating different-sized files
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ echo -n "12345" > f1.txt
[10/20/24]seed@VM:~/.../task4$ echo -n "1234567890" > f2.txt
[10/20/24]seed@VM:~/.../task4$ echo -n "1234567890abcdef" > f3.txt
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ ls -la
total 20
drwxrwxr-x 2 seed seed 4096 Oct 20 14:03 .
drwxrwxrwx 8 seed seed 4096 Oct 20 13:47 ..
-rw-rw-r-- 1 seed seed    5 Oct 20 14:04 f1.txt
-rw-rw-r-- 1 seed seed   10 Oct 20 14:04 f2.txt
-rw-rw-r-- 1 seed seed   16 Oct 20 14:04 f3.txt
[10/20/24]seed@VM:~/.../task4$
```

CBC Mode

- The image below shows the encryption of all 3 files in CBC mode.

```
[10/20/24]seed@VM:~/.../task4$ # CBC Mode
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cbc -e -in f1.txt -out cbc_c1.enc -k task4pasz -pbkdf2 -p
salt=9EB03E56E0F732BC
key=6411789D55796DED61C3C89AA33C165A
iv =52BFE772E9C4DF6FA4C5F4290E3C362E
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cbc -e -in f2.txt -out cbc_c2.enc -k task4pasz -pbkdf2 -p
salt=C9675691996A87FF
key=9D680B5C2B3D58FEC99A1F92240A79C0
iv =4EA09FACC4336DDCBC3381E586C3FE84
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cbc -e -in f3.txt -out cbc_c3.enc -k task4pasz -pbkdf2 -p
salt=494C8D9E2AD4F0B5
key=9C5BF78FD153FE000807ED42A9573DD3
iv =32C65427ED5D10CFD97C93FD0E48A317
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # now let's check the size of the encrypted files
[10/20/24]seed@VM:~/.../task4$ ls -la
total 32
drwxrwxr-x 2 seed seed 4096 Oct 20 14:49 .
drwxrwxrwx 8 seed seed 4096 Oct 20 13:47 ..
-rw-rw-r-- 1 seed seed   32 Oct 20 14:48 cbc_c1.enc
-rw-rw-r-- 1 seed seed   32 Oct 20 14:48 cbc_c2.enc
-rw-rw-r-- 1 seed seed   48 Oct 20 14:49 cbc_c3.enc
-rw-rw-r-- 1 seed seed    5 Oct 20 14:04 f1.txt
-rw-rw-r-- 1 seed seed   10 Oct 20 14:04 f2.txt
-rw-rw-r-- 1 seed seed   16 Oct 20 14:04 f3.txt
[10/20/24]seed@VM:~/.../task4$
```

- Now we decrypt the encrypted files and compare the hex values of the original plaintext and the plaintext from the decryption.

```
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # Let's decrypt all ciphertexts
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cbc -d -nopad -in cbc_c1.enc -out cbc_p1.txt -k task4pasz -pbkdf2 -p
salt=9EB03E56E0F732BC
key=6411789D55796DED61C3C89AA33C165A
iv =52BFE772E9C4DF6FA4C5F4290E3C362E
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cbc -d -nopad -in cbc_c2.enc -out cbc_p2.txt -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cbc -d -nopad -in cbc_c3.enc -out cbc_p3.txt -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # now let's compare hex value for the two plaintexts
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ hexdump -C f1.txt
00000000  31 32 33 34 35                                   |12345|
00000005
[10/20/24]seed@VM:~/.../task4$ hexdump -C cbc_p1.txt
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
00000010
[10/20/24]seed@VM:~/.../task4$ hexdump -C f2.txt
00000000  31 32 33 34 35 36 37 38  39 30                    |1234567890|
0000000a
[10/20/24]seed@VM:~/.../task4$ hexdump -C cbc_p2.txt
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06 06  |1234567890......|
00000010
[10/20/24]seed@VM:~/.../task4$ hexdump -C f3.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010
[10/20/24]seed@VM:~/.../task4$ hexdump -C cbc_p3.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
[10/20/24]seed@VM:~/.../task4$
```

Observation

From the hexdump commands above, it's clear the in CBC mode, padding is used.

- **f1.txt**: Since this file is 5-bytes and 11-bytes less than the block size, 11-bytes of padding is added. And each byte is the hex representation of 11 (0x0b).
- **f2.txt**: Here, 6-bytes of padding is added and each byte is the hex form of 6 (0x06).
- **f3.txt**: For this file, it's original size is already a multiple of the block size for AES-128. As a result, a new block of size 16-bytes is added as padding with each byte having the hex representation of 16 (0x10). If no padding were added to the block-sized plaintext, the decryption process would have no way of knowing whether the last block is genuine plaintext or whether it includes padding.

ECB Mode

- Similarly, we repeat the process for ECB mode.

```
[10/20/24]seed@VM:~/.../task4$ # ECB Mode
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # Encrypting Files
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ecb -e -in f1.txt -out ecb_c1.enc -k task4pasz -pbkdf2 -p
salt=BD6894BA1AD91D76
key=BA2732731AAB9B425E41158F21EC2FAD
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ecb -e -in f2.txt -out ecb_c2.enc -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ecb -e -in f3.txt -out ecb_c3.enc -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # Decrypting Files
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ecb -d -nopad -in ecb_c1.enc -out ecb_p1.txt -k task4pasz -pbkdf2 -p
salt=BD6894BA1AD91D76
key=BA2732731AAB9B425E41158F21EC2FAD
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ecb -d -nopad -in ecb_c2.enc -out ecb_p2.txt -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ecb -d -nopad -in ecb_c3.enc -out ecb_p3.txt -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$
```

- Just like CBC mode above, the padding works the same way for ECB mode hence the identical output below as thr one we had for CBC mode. Therefore the same explanation holds true.

```
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # Comparing Hex values
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ hexdump -C f1.txt
00000000  31 32 33 34 35                                       |12345|
00000005
[10/20/24]seed@VM:~/.../task4$ hexdump -C ecb_p1.txt
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
00000010
[10/20/24]seed@VM:~/.../task4$ hexdump -C f2.txt
00000000  31 32 33 34 35 36 37 38  39 30                    |1234567890|
0000000a
[10/20/24]seed@VM:~/.../task4$ hexdump -C ecb_p2.txt
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06 06  |1234567890......|
00000010
[10/20/24]seed@VM:~/.../task4$ hexdump -C f3.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010
[10/20/24]seed@VM:~/.../task4$ hexdump -C ecb_p3.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
[10/20/24]seed@VM:~/.../task4$
```

CFB Mode

- In this mode, we show all the steps (encryption, decryption and comapring hex values) in the snapshot below.
- In this mode no padding is added for any of the files. This is made evident by the same hex value shown for both the original plaintext and the plaintext we got from decrypting the ciphertext.

```
[10/20/24]seed@VM:~/.../task4$ # CFB Mode
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # Encrypting Files
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cfb -e -in f1.txt -out cfb_c1.enc -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cfb -e -in f2.txt -out cfb_c2.enc -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cfb -e -in f3.txt -out cfb_c3.enc -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # Decrypting Files
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cfb -d -nopad -in cfb_c1.enc -out cfb_p1.txt -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cfb -d -nopad -in cfb_c2.enc -out cfb_p2.txt -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-cfb -d -nopad -in cfb_c3.enc -out cfb_p3.txt -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # Comparing Hex Values
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ hexdump -C f1.txt
00000000  31 32 33 34 35                                       |12345|
00000005
[10/20/24]seed@VM:~/.../task4$ hexdump -C cfb_p1.txt
00000000  31 32 33 34 35                                       |12345|
00000005
[10/20/24]seed@VM:~/.../task4$ hexdump -C f2.txt
00000000  31 32 33 34 35 36 37 38  39 30                    |1234567890|
0000000a
[10/20/24]seed@VM:~/.../task4$ hexdump -C cfb_p2.txt
00000000  31 32 33 34 35 36 37 38  39 30                    |1234567890|
0000000a
[10/20/24]seed@VM:~/.../task4$ hexdump -C f3.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010
[10/20/24]seed@VM:~/.../task4$ hexdump -C cfb_p3.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010
```

OFB Mode

- Similarly to CFB mode, no padding is added in OFB mode hence the identical result shown below.

```
[10/20/24]seed@VM:~/.../task4$ # OFB Mode
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # Encrypting Files
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ofb -e -in f1.txt -out ofb_c1.enc -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ofb -e -in f2.txt -out ofb_c2.enc -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ofb -e -in f3.txt -out ofb_c3.enc -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # Decrypting Files
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ofb -d -nopad -in ofb_c1.enc -out ofb_p1.txt -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ofb -d -nopad -in ofb_c2.enc -out ofb_p2.txt -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$ openssl enc -aes-128-ofb -d -nopad -in ofb_c3.enc -out ofb_p3.txt -k task4pasz -pbkdf2
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ # Comparing Hex Values
[10/20/24]seed@VM:~/.../task4$
[10/20/24]seed@VM:~/.../task4$ hexdump -C f1.txt
00000000  31 32 33 34 35                                    |12345|
00000005
[10/20/24]seed@VM:~/.../task4$ hexdump -C ofb_p1.txt
00000000  31 32 33 34 35                                    |12345|
00000005
[10/20/24]seed@VM:~/.../task4$ hexdump -C f2.txt
00000000  31 32 33 34 35 36 37 38  39 30                    |1234567890|
0000000a
[10/20/24]seed@VM:~/.../task4$ hexdump -C ofb_p2.txt
00000000  31 32 33 34 35 36 37 38  39 30                    |1234567890|
0000000a
[10/20/24]seed@VM:~/.../task4$ hexdump -C f3.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010
[10/20/24]seed@VM:~/.../task4$ hexdump -C ofb_p3.txt
00000000  31 32 33 34 35 36 37 38  39 30 61 62 63 64 65 66  |1234567890abcdef|
00000010
```

Task 5

This tasks simulated how the decrypted plaintext is affected when the ciphertext is corrupted before decryption.

Question: How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively?

Thoughts before conducting tasks

- **ECB**: We can recover the entire file correctly except for the 16-byte block containing the corrupted ciphertext. Only the corrupted block will be damaged, while other blocks will remain intact.
- **CBC**: We will lose the data in the corrupted block and the block immediately following it (a total of 32 bytes). The remaining data will be correctly decrypted.
- **CFB**: We can recover all but the 42nd byte of data that gets corrupted.
- **OFB**: Just as CFB, we will lose only the 42nd byte of the decrypted data. The rest of the data will be recovered correctly.

ECB Mode

- To make the observation easy, we began with a 2000-byte plaintext filled with all zeroes. We after encrypting this plaintext, we corrupt the 42nd byte of the ciphertext.

```
[10/20/24]seed@VM:~/.../task5$ # ECB Mode
[10/20/24]seed@VM:~/.../task5$
[10/20/24]seed@VM:~/.../task5$ # Creating 2K-sized file filled with null bytes
[10/20/24]seed@VM:~/.../task5$
[10/20/24]seed@VM:~/.../task5$ truncate -s 2K file.bin
[10/20/24]seed@VM:~/.../task5$
[10/20/24]seed@VM:~/.../task5$ # Encrypting File
[10/20/24]seed@VM:~/.../task5$
[10/20/24]seed@VM:~/.../task5$ openssl enc -aes-128-ecb -e -in file.bin -out ecb_file.enc -k task5pas -pbkdf2
[10/20/24]seed@VM:~/.../task5$
[10/20/24]seed@VM:~/.../task5$ # Decrypting Corrupted File
[10/20/24]seed@VM:~/.../task5$
[10/20/24]seed@VM:~/.../task5$ openssl enc -aes-128-ecb -d -in ecb_corrupted.enc -out ecb_plain.bin -k task5pas -pbkdf2
[10/20/24]seed@VM:~/.../task5$
```

- Now we decrypt the corrupted ciphertext and compare it to the original plaintext.

```
[10/20/24]seed@VM:~/.../task5$ # Now let't compare plaintexts
[10/20/24]seed@VM:~/.../task5$
[10/20/24]seed@VM:~/.../task5$ hexdump -C file.bin
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000800
[10/20/24]seed@VM:~/.../task5$ hexdump -C ecb_plain.bin
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000010  0a d4 b4 8f 36 ce 5e 87  19 52 c6 66 37 cb 54 3a  |....6.^..R.f7.T:|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000800
[10/20/24]seed@VM:~/.../task5$
```

- From the above snapshot, we can see that, only 16-bytes of the corrupted ciphertext which we decrypted are non-null bytes. And this matches the expectation I had before for ECB mode. This is due to the fact that in ECB, each block is encrypted and decrypted independently.

CBC Mode

The image below shows the entire process I went through to see how error propagates in the CBC cipher mode. The results affirm my expection of not being able to recover 32-bytes even though we had corrupted just the 42nd bytes. This is because a single corrupted byte in the ciphertext affects the decryption of both the current block (the 16-byte block containing the corruption) and the next block (due to the chaining mechanism).

```
[10/21/24]seed@VM:~/.../task5$ # CBC Mode
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Encryption
[10/21/24]seed@VM:~/.../task5$ openssl enc -aes-128-cbc -e -in file.bin -out cbc_file.enc -k task5pass -pbkdf2
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Corruption
[10/21/24]seed@VM:~/.../task5$ cp cbc_file.enc cbc_corrupted.enc
[10/21/24]seed@VM:~/.../task5$ hexedit cbc_corrupted.enc
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Decryption
[10/21/24]seed@VM:~/.../task5$ openssl enc -aes-128-cbc -d -in cbc_corrupted.enc -out cbc_plain.txt -k task5pass -pbkdf2
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Comparing Plaintexts
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ hexdump -C file.bin
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000800
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ hexdump -C cbc_plain.txt
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000010  a9 d5 47 e3 2e 55 13 fa  ed d4 de 75 85 4c e1 53  |..G..U.....u.L.S|
00000020  00 00 00 00 00 00 00 00  00 01 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000800
[10/21/24]seed@VM:~/.../task5$
```

CFB Mode

In this cipher mode, the observation I made after going through the task is quite different from what I expected for executing the task.

- I expected that to lose only the 42nd byte but I'm losing 32-bytes instead.
- This actually confirmed the working of the CFB Mode for me, where the ciphertext from the previous block is provided as input to the shift register which then feeds into the encryption algorithm.

```
[10/21/24]seed@VM:~/.../task5$ # CFB Mode
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Encryption
[10/21/24]seed@VM:~/.../task5$ openssl enc -aes-128-cfb -e -in file.bin -out cfb_file.enc -k task5pass -pbkdf2
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Corruption
[10/21/24]seed@VM:~/.../task5$ cp cfb_file.enc cfb_corrupted.enc
[10/21/24]seed@VM:~/.../task5$ hexedit cfb_corrupted.enc
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Decryption
[10/21/24]seed@VM:~/.../task5$ openssl enc -aes-128-cfb -d -in cfb_corrupted.enc -out cfb_plain.enc -k task5pass -pbkdf2
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Comparing Plaintexts
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ hexdump file.bin
0000000 0000 0000 0000 0000 0000 0000 0000 0000
*
0000800
[10/21/24]seed@VM:~/.../task5$ hexdump cfb_plain.enc
0000000 0000 0000 0000 0000 0000 0000 0000 0000
0000010 0000 0000 0000 0000 0100 0000 0000 0000
0000020 cb9d 3135 502d b56a 5a5c 21a7 ba31 319e
0000030 0000 0000 0000 0000 0000 0000 0000 0000
*
0000800
[10/21/24]seed@VM:~/.../task5$
```

OFB Mode

In this mode, my initial thought of having all but the 42nd byte recoverable proves to be correct. The reason is that OFB cipher mode turns a block cipher into a stream cipher. Hence a single bit error in the ciphertext will affect only the corresponding byte of the plaintext. No further bytes are affected because the keystream is not dependent on the ciphertext.

```
[10/21/24]seed@VM:~/.../task5$ # OFB Mode
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Encryption
[10/21/24]seed@VM:~/.../task5$ openssl enc -aes-128-ofb -e -in file.bin -out ofb_file.enc -k task5pass -pbkdf2
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Corruption
[10/21/24]seed@VM:~/.../task5$ cp ofb_file.enc ofb_corrupted.enc
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ hexedit ofb_corrupted.enc
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Decryption
[10/21/24]seed@VM:~/.../task5$ openssl enc -aes-128-ofb -d -in ofb_corrupted.enc -out ofb_plain.bin -k task5pass -pbkdf2
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ # Comparing Plaintexts
[10/21/24]seed@VM:~/.../task5$
[10/21/24]seed@VM:~/.../task5$ hexdump file.bin
0000000 0000 0000 0000 0000 0000 0000 0000 0000
*
0000800
[10/21/24]seed@VM:~/.../task5$ hexdump ofb_plain.bin
0000000 0000 0000 0000 0000 0000 0000 0000 0000
0000010 0000 0000 0000 0000 0300 0000 0000 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
*
0000800
[10/21/24]seed@VM:~/.../task5$
```

Task 6

6.1 - IV Experiment

The image below shows the creation a plaintext, key and two IVs. And then the encryption of the plaintext using (1) different IVs and (2) the same IV (iv1).

```
/task6$ # 6.1 - IV Experiment
/task6$
/task6$ # Let's create a simple file
/task6$ echo "This is a regular text!" > plaintext.txt
/task6$
/task6$ # now let's generate a key and 2 IVs
/task6$ openssl rand -hex 32 > key.bin
/task6$ openssl rand -hex 16 > iv1.bin
/task6$ openssl rand -hex 16 > iv2.bin
/task6$
/task6$ # we now encrypt using the different IVs
/task6$ openssl enc -aes-256-cbc -e -in plaintext.txt -out c1_iv1.enc -K $(cat key.bin) -iv $(cat iv1.bin)
/task6$
/task6$ openssl enc -aes-256-cbc -e -in plaintext.txt -out c2_iv2.enc -K $(cat key.bin) -iv $(cat iv2.bin)
/task6$
/task6$ # now let's use iv1 again
/task6$ openssl enc -aes-256-cbc -e -in plaintext.txt -out c3_iv1.enc -K $(cat key.bin) -iv $(cat iv1.bin)
/task6$
```

- Now we show the comparison between the ciphertexts generated when different IVs are used versus when the same IV is used.

```
[10/21/24]seed@VM:~/.../task6$
[10/21/24]seed@VM:~/.../task6$ # now it's time to compare the ciphertexts
[10/21/24]seed@VM:~/.../task6$
[10/21/24]seed@VM:~/.../task6$ diff c1_iv1.enc c2_iv2.enc
1c1
< ◆$I◆◆◆
        _[◆◆Fi<◆F◆ri◆◆◆X&.;8
\ No newline at end of file
---
WTR◆%x,8◆~◆◆◆◆◆◆◆◆
\ No newline at end of file
[10/21/24]seed@VM:~/.../task6$ diff c1_iv1.enc c3_iv1.enc
[10/21/24]seed@VM:~/.../task6$
```

- From the above output, the diff command clearly indicates that the ciphertexts generated using different IVs are not the same even thought we used the same key and plaintext.
- On the other hand, the diff command shows no output for comparing the ciphertexts generated with the same IV. This shows that they are identical and no differences exists between them.

Why IV needs to be unique

The IV ensures randomness in the encryption process. If the IV is reused, it compromises the confidentiality of the data by exposing patterns. This is particularly dangerous in modes like CBC, where identical plaintext blocks will result in identical ciphertext blocks if the same IV is reused.

6.2 - Common Mistake: Use the Same IV

In this section, I modified the given code sample to in order to use the given information on P1, C1 and C2 to reveal P2. The snapshot below shows the modified code.

```python
# XOR two bytearrays
def xor(first, second):
    return bytearray(x ^ y for x, y in zip(first, second))

# Given the following:
P1 = "This is a known message!"  # Plaintext 1
C1 = bytes.fromhex("a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159")  # Ciphertext 1 (in hex)
C2 = bytes.fromhex("bf73bcd3509299d566c35b5d450337e1bb175f903fafc159")  # Ciphertext 2

# Convert P1 to bytes
P1_bytes = bytes(P1, 'utf-8')

# Calculate the keystream
keystream = xor(P1_bytes, C1)

# Use the keystream to recover P2 by XORing with C2
P2_bytes = xor(keystream, C2)

# Convert P2 bytes back to a string
P2 = P2_bytes.decode('utf-8')
print(f"Recovered P2: {P2}")
```

- After executing the script above, I successfully recovered the contents of P2 as: "Order: Launch a missile!". This is shown in the image below:

```
[10/21/24]seed@VM:~/.../task6$ # 6.2 - Use the Same IV
[10/21/24]seed@VM:~/.../task6$
[10/21/24]seed@VM:~/.../task6$ # let's run the modified code sample
[10/21/24]seed@VM:~/.../task6$
[10/21/24]seed@VM:~/.../task6$ python ofb_task6-2.py
Recovered P2: Order: Launch a missile!
[10/21/24]seed@VM:~/.../task6$
[10/21/24]seed@VM:~/.../task6$
```

What Happens in CFB Mode?

In CFB Mode, the attack works in a similar way, but the ciphertext plays a role in generating the next keystream. If the IV gets reused, you can still use P1 and C1 to recover part of P2. How much of P2 can be exposed depends on the feedback size (e.g., CFB-1, CFB-8, CFB-128). In most real-world cases, a large chunk of P2 can be revealed if the feedback size is big enough, like in CFB-128.

6.3 - Use a Predictable IV

In this task I similated the known plaintext attack to determine what the Bob's secret message is when predictable IVs are used. The snapshot below shows the code that I wrote to manipulate the known parameters (known-iv, next-iv, and guess-msg) as a way of reverse engineering the cbc cipher to get the same plaintext that Bob got if my initial guess is right.

```python
task6 > cbc_task6-3.py > prepare_ciphertext_guess
# XOR two bytearrays
def xor(first, second):
    return bytearray(x ^ y for x, y in zip(first, second))

def prepare_ciphertext_guess(padded_plaintext, known_iv, next_iv):

    # XOR the padded plaintext with the next IV (which the oracle will use)
    xor_with_next_iv = xor(padded_plaintext, next_iv)

    # XOR the result with the known IV (which was used by Bob's secret message)
    manipulated_ciphertext = xor(xor_with_next_iv, known_iv)

    print(f"Manipulated Ciphertext Guess (hex): {manipulated_ciphertext.hex()}")

# Bob's IV used in the first encryption (known)
known_iv = bytes.fromhex("ef7fd84fdbeab0aa3c6ac3ad3f4295f0")

next_iv = bytes.fromhex("3c0d4ea9dbeab0aa3c6ac3ad3f4295f0") # The next IV

# Pre-padded guesses for "Yes" and "No"
padded_yes = bytes.fromhex("5965730d0d0d0d0d0d0d0d0d0d0d0d0d")
padded_no  = bytes.fromhex("4e6f0e0e0e0e0e0e0e0e0e0e0e0e0e0e")

# Choose the guess to test
padded_plaintext = padded_yes

# run the function
prepare_ciphertext_guess(padded_plaintext, known_iv, next_iv)
```

- Next, I show that the docker container is up and running:

```
[10/21/24]seed@VM:~/.../task6$ dcup
Creating www-10.9.0.80 ... done
Attaching to www-10.9.0.80
www-10.9.0.80 | Server listening on 3000 for known_iv
www-10.9.0.80 | Connect to 3000, launching known_iv
```

- Now let's see the manipulated ciphertext that I got from running the code shown above. My initial guess is that Bob's secret message is "Yes".

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

[10/21/24]seed@VM:~/.../task6$ python cbc_task6-3.py
Manipulated Ciphertext Guess (hex): 8a17e5eb0d0d0d0d0d0d0d0d0d0d0d0d
[10/21/24]seed@VM:~/.../task6$
```

- So I copied the manipulated ciphertext above and provided it as my plaintext to the encryption oracle which then gave my ciphertext.
- Comparing my ciphertext to Bob's we see that mine is twice the length of Bob's. It's clear that the first block of my ciphertext matches the Bob's 16-byte ciphertext. And this confirms that my guess of "Yes" was right.

- This proves that using predicatable IVs is wrong and can lead to serious consequences!

```
[10/21/24]seed@VM:~/.../task6$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 72856c0b465e2317a0dbcafba1cd0d03
The IV used    : ef7fd84fdbeab0aa3c6ac3ad3f4295f0

Next IV        : 3c0d4ea9dbeab0aa3c6ac3ad3f4295f0
Your plaintext : 8a17e5eb0d0d0d0d0d0d0d0d0d0d0d
Your ciphertext: 72856c0b465e2317a0dbcafba1cd0d037fd8812338060111647dadd0f2d77684

Next IV        : ebfc9f0bdceab0aa3c6ac3ad3f4295f0
Your plaintext :
```