SEED Labs

Task 2: Encryption using Different Ciphers and Modes

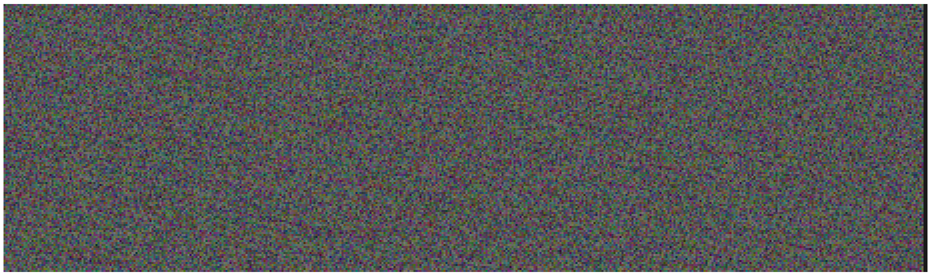
```
# Encryption using AES-128-CBC
openssl enc -aes-128-cbc -e -a -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_2/aes_128_cbc_cipher.txt

# Encryption using AES-256-ECB
openssl enc -aes-256-ecb -e -a -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_2/aes_256_ecb_cipher.txt

# Encryption using AES-256-CTR
openssl enc -aes-256-ctr -e -a -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_2/aes_256_ctr_cipher.txt

```

In the script above, we encrypt a plaintext we created with over 1000 bytes. We encrypt it using CBC, ECB and CTR respectively using AES encryption.

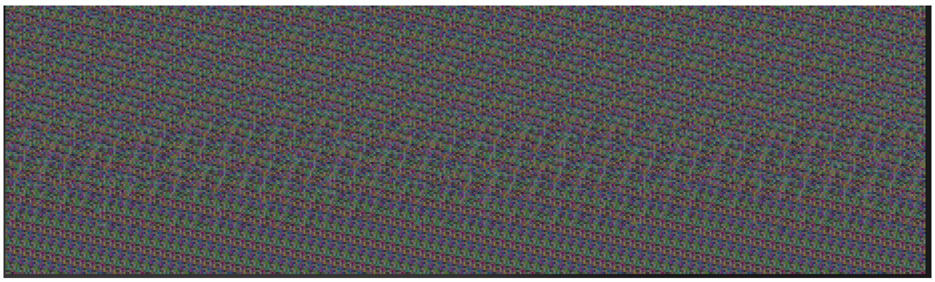
Task 3: Encryption Mode – ECB vs. CBC

```

# File Encryption using CBC
openssl enc -aes-128-cbc -e -a -pbkdf2 -in Labsetup/Files/pic_original.bmp -out outputs/task_3/pic_cbc.bmp

# File Encryption using ECB
openssl enc -aes-128-ecb -e -a -pbkdf2 -in Labsetup/Files/pic_original.bmp -out outputs/task_3/pic_ecb.bmp

# Fix the header for the CBC encrypted output
head -c 54 Labsetup/Files/pic_original.bmp > header
tail -c +55 outputs/task_3/pic_cbc.bmp > body
cat header body > outputs/task_3/pic_result_cbc.bmp

# Fix the header for the ECB encrypted output
head -c 54 Labsetup/Files/pic_original.bmp > header
tail -c +55 outputs/task_3/pic_ecb.bmp > body
cat header body > outputs/task_3/pic_result_ecb.bmp

rm header
rm body


```

In this task, we encrypt a .bmp image using both CBC and ECB encryption types with AES-128. Then, we change the headers of both encrypted images with the original header information; the first 54 bytes. This ensures the bitmap image can be successfully read by image viewer applications. It is impossible to derive any information of the original image by looking at the encrypted one. It is just a mix of many colors and is unintelligible in general.



The picture above is the result of the encrypted image using CBC.



The picture above isthe result of the encrypted image using ECB.

Both images cannot be understood as they are both just a mix of many colors.

Task 4: Padding

```
# Encryption using AES-256-CBC
openssl enc -aes-256-cbc -e -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_4/aes_256_cbc_cipher.txt

# Encryption using AES-256-ECB
openssl enc -aes-256-ecb -e -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_4/aes_256_ecb_cipher.txt

# Encryption using AES-256-CFB
openssl enc -aes-256-cfb -e -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_4/aes_256_cfb_cipher.txt

# Encryption using AES-256-OFB
openssl enc -aes-256-ofb -e -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_4/aes_256_ofb_cipher.txt

# Create 3 files which contain 5 bytes, 10 bytes, and 16 bytes, respectively
echo -n "12345" > outputs/task_4/f1.txt
echo -n "1234567890" > outputs/task_4/f2.txt
echo -n "1234567890123456" > outputs/task_4/f3.txt

# Encrypt the three files using 128-bit AES with CBC mode
openssl enc -aes-128-cbc -e -pbkdf2 -in outputs/task_4/f1.txt -out outputs/task_4/f1_enc.txt
openssl enc -aes-128-cbc -e -pbkdf2 -in outputs/task_4/f2.txt -out outputs/task_4/f2_enc.txt
openssl enc -aes-128-cbc -e -pbkdf2 -in outputs/task_4/f3.txt -out outputs/task_4/f3_enc.txt

# Decrypt the three files using 128-bit AES with CBC mode
openssl enc -aes-128-cbc -d -nopad -pbkdf2 -in outputs/task_4/f1_enc.txt -out outputs/task_4/f1_dec.txt
openssl enc -aes-128-cbc -d -nopad -pbkdf2 -in outputs/task_4/f2_enc.txt -out outputs/task_4/f2_dec.txt
openssl enc -aes-128-cbc -d -nopad -pbkdf2 -in outputs/task_4/f3_enc.txt -out outputs/task_4/f3_dec.txt

echo "File 1"
hexdump -C outputs/task_4/f1_dec.txt

echo "File 2"
hexdump -C outputs/task_4/f2_dec.txt

echo "File 3"
hexdump -C outputs/task_4/f3_dec.txt
```

4.1: Which encryption types use padding?

ECB and CBC modes require padding. ECB mode encrypts each block of plaintext independently, and the size of the ciphertext is more than the plaintext size by one padding block. CBC mode uses the previous ciphertext block as the input to the next block, and the size of the ciphertext is also more than the plaintext size by one padding block. Padding is used in CBC mode to ensure that the plaintext is a multiple of the block size.
CFB and OFB modes are meant for streaming and don't require padding. CFB does require padding unless you use a segment size of 1 bit (or 8 bits if your message is byte-oriented). In CFB mode, the previous ciphertext block is encrypted and the output is XORed with the current plaintext block to create the current ciphertext block. Plaintext can be any size, and no padding is required. OFB mode also does not require any padding, and there are no other limitations on the plaintext.

4.2: Encrypting 3 Files

After creating 3 files which contain 5 bytes, 10 bytes, and 16 bytes, respectively, we successfully encrypted the files using -aes-128-cbc encryption. Both the 5 and 10 byte files had the same size of 16 bytes while the 16 byte file was 32 bytes in size, double compared to the other files.

Using hexdump, we were able to see the paddings that were applied to each of the files as shown below:

```
File 1
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
00000010
File 2
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06 06  |1234567890......|
00000010
File 3
00000000  31 32 33 34 35 36 37 38  39 30 31 32 33 34 35 36  |1234567890123456|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020

SEED_Lab on main [!?] via v3.10.12 took 8s
❯
```

Task 5: Error Propagation – Corrupted Cipher Text

```
# Encrypt the file using the AES-128 cipher
openssl enc -aes-256-ecb -e -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_5/aes_256_ecb_cipher.txt
openssl enc -aes-256-cbc -e -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_5/aes_256_cbc_cipher.txt
openssl enc -aes-256-cfb -e -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_5/aes_256_cfb_cipher.txt
openssl enc -aes-256-ofb -e -pbkdf2 -in Labsetup/Files/plaintext.txt -out outputs/task_5/aes_256_ofb_cipher.txt

# Use Bless to corrupt a single bit of the 55th byte in each encrypted file
bless outputs/task_5/aes_256_ecb_cipher.txt
bless outputs/task_5/aes_256_cbc_cipher.txt
bless outputs/task_5/aes_256_cfb_cipher.txt
bless outputs/task_5/aes_256_ofb_cipher.txt

# Decrypt the corrupted ciphertext file using the correct key and IV
openssl enc -aes-256-ecb -d -pbkdf2 -in outputs/task_5/aes_256_ecb_cipher.txt -out outputs/task_5/aes_256_ecb_cipher_dec.txt
openssl enc -aes-256-cbc -d -pbkdf2 -in outputs/task_5/aes_256_cbc_cipher.txt -out outputs/task_5/aes_256_cbc_cipher_dec.txt
openssl enc -aes-256-cfb -d -pbkdf2 -in outputs/task_5/aes_256_cfb_cipher.txt -out outputs/task_5/aes_256_cfb_cipher_dec.txt
openssl enc -aes-256-ofb -d -pbkdf2 -in outputs/task_5/aes_256_ofb_cipher.txt -out outputs/task_5/aes_256_ofb_cipher_dec.txt
```

For this task, we used the same plaintext we created earlier to understand error propagation. Using bless hex editor, we changed a single bit of the 55th byte in each of the encrypted files to simulate the files being corrupted.

On decrypting these files and reading them, we found that using OFB had the least error propagation with only one word in our plaintext being slightly different from our original text.