Mini Project 2: Secret-Key Encryption
(due Feb 23, end of the day)

In this mini project, you will learn the secret-key encryption concepts and how to use secret-key encryption algorithms in practice. In particular, you will use the OpenSSL library to: (1) test the AES encryption algorithm and the use of padding and IV in different modes of operations; (2) observe error propagation; and (3) learn common mistakes in using the secret-key cryptography. This project uses the SEED Crypto Lab materials with modifications.

Setting: You will use a computer with Ubuntu 20.03, the OpenSSL library, and hex editor GHex or bless. The computers in Eaton 2003 have installed the required software. On the desktop, you can find a folder named MP2, which contains the files for this project.

Task 1: Encryption using different ciphers and modes of operations

Learn to use openssl enc command to encrypt and decrypt a file. For example:

```
openssl enc -aes-128-cbc -e -in plain.txt -out cipher1.bin -K 00112233445566778899aabbccddeeff
-iv 01020304050607080910111213141516
```

"-aes-128-cbc" means using the AES cipher in the CBC mode and a 128-bit key. You can specify the key and/or IV and provide the key and/or IV with the correct length. You can type man enc to learn the meaning of the command-line options and the supported cipher types.

1. Please try three different ciphers. In the report, **show the commands that you used and the results.**

2. Use the ECB and CBC modes to encrypt a picture file. Then, convert both encrypted files to the .bmp format and use a picture viewing program (e.g., eog viewer) to open the encrypted files. A sample picture, pic_original.bmp, is provided in the MP2 folder. You can use AES-128 or any other block ciphers in this step. **Please show your results in the report.**

Hint: The first 54 bytes of the .bmp file contain the header information about the picture. To convert a file to the .bmp format, you can get the header from a valid .bmp file (e.g., pic_original.bmp) and the data from the target file. Then, combine the header and the data into a new file, using the following commands.

```
$ head -c 54 file1 > header
$ tail -c +55 file2 > data
$ cat header data > new.bmp
```

Question 1. What do you see in Step 2? Please explain your observations.

Task 2: Padding

Many block ciphers use the PKCS#5 padding scheme, which we will learn in this task.

1. Use the ECB, CBC, CFB, and OFB modes to encrypt a file. You can use any cipher and any file in this task. **Please report which modes have padding and which do not.**

Question 2. Some modes do not need padding. Please explain why.

2. Create three small files, which contain 5 bytes, 10 bytes, and 16 bytes, respectively. Encrypt them using 128-bit AES with the CBC mode. **Please report the size of each encrypted file.**

3. Then, decrypt the files to see what was added as padding during the encryption. **Report your results.**

Hint 1: We can use the command `echo -n` to create small files. For example, `echo -n "12345" > f1.txt` creates a file f1.txt with length 5. If we don't use the option "-n", the length will be 6, because a newline character will be added by "echo".

Hint 2: Decryption will automatically remove the padding by default. We can use the option "-nopad" to disable padding during the decryption, for example, `openssl enc -aes-128-cbc -d -nopad`. Since padding data may not be printable, you can use a hex tool to display the content. For example, you can use the below commands:

```
$ hexdump -C p1.txt
00000000 31 32 33 34 35 36 37 38 39 49 4a 4b 4c 0a     |123456789IJKL.|
$ xxd p1.txt
00000000: 3132 3334 3536 3738 3949 4a4b 4c0a          123456789IJKL.
```

Task 3: Error Propagation – Corrupted Cipher Text

This task is to understand the error propagation property of different encryption modes. The goal is to examine how much information can be recovered by decrypting a corrupted ciphertext file for the following encryption modes: ECB, CBC, CFB, and OFB. For **each** encryption mode, complete the following steps:

1. Create a text file that is at least 1000 bytes long and encrypt the file using AES ECB mode with a 128-bit key.

Hint: You can use any text file of your choice. Or, you can use the `yes` command that continuously prints the supplied string on the console and store the output in a text file. To control the size of the file, you can use the `head` command. For example,

```
$ yes "this is a test file" | head -c 1KB > f2.txt
```

2. Suppose that a single bit of the 56th byte in the encrypted file got corrupted. You can achieve this corruption using the bless hex editor. **Please take a screenshot of the corrupted file in the hex editor.**

Hint: You can press the insert button to switch to the overwrite mode. Otherwise, the editor will only add new values.

3. Decrypt the corrupted ciphertext file using the correct key and IV. **Please report how much information you recovered by decrypting the corrupted file.**

4. Repeat steps 1-3 and try aes-128-cbc, aes-128-cfb, and aes-128-ofb.

Question 3. Based on your observations, what is the difference in error propagation among different encryption modes?

Task 4: IV and Common Mistakes

The initial vector (IV) should be random and unique. Otherwise, the data encrypted may not be secure. This task is to understand the problems if an IV is not unique (reused).

1. Encrypt the same plaintext under aes-128-cbc using (1) two different IVs and (2) the same IV. **Please describe your observations in both cases. Then, discuss why IV needs to be unique.**

Hint: You can use the diff command to compare two files.

2. Can we use the same IV if the plaintext does not repeat? To answer this question, let us consider a *known-plaintext attack*. If the attacker gets hold of a plaintext (P1) and the corresponding ciphertext (C1) encrypted under aes-128-ofb, can she decrypt another ciphertext (assume the same IV is used)?

> Plaintext P1: This is a known message!
>
> Ciphertext C1: a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159
>
> Plaintext P2: (unknown, try to decrypt)
>
> Ciphertext C2: bf73bcd3509299d566c35b5d450337e1bb175f903fafc159

Given P1, C1, and C2 as above, **please try to figure out the actual content of P2**.

Hint 1: we provide a sample program, called sample_code.py, to compute the XOR of two strings. You can run it in the terminal using the command: python3 filename.py. You can modify it to figure out P2.

Hint 2:You can check the hex value of a file using hexdump or xxd as in Task 2.

Question 4. Based on your result, is aes-128-ofb secure when the same IV is used? If aes-128-cfb is used, how much of P2 can be revealed?

Report requirements and submission

Please submit a report for this project to Canvas.

1. For each task, describe what you have done and what you have observed.

2. Include screenshots in the report to show your results.

3. Answer all the questions.