CS 3339
Lab 2: Symmetric Key Cryptography

In this lab, we will be investigating symmetric key encryption and how you can use it.
This will require a system with OpenSSL 1.1.1 (or later). The Kali VM you installed in Lab 1 should have this already installed.

You can check this by opening the terminal and entering the command:

```
# openssl version
```

The first thing we will do is look at the list of encryption algorithms provided by openssl.

 # openssl enc -ciphers

To see the options available to you when encrypting, use the command:

 # openssl enc -help

Part 1:
Now we will use OpenSSL for symmetric key encryption with a password. This mean one key is used for both encryption and decryption. Lets encrypt plaintext.txt, which can be found in the same .zip file as this document. To encrypt with OpenSSL we select one of the ciphers and specify the input and the output files. We will use AES in cipher block chaining mode with a 256 bit key. This will ask for a password, which you need to remember to decrypt. The password is converted to a key of the appropriate length (256 bits).

 # openssl aes-256-cbc -e -in plaintext.txt -pbkdf2 -out ciphertext

Use the following commands to compare the files:

 # ls -l plaintext.txt ciphertext
 # file plaintext.txt ciphertext
 # xxd plaintext.txt
 # xxd ciphertext

How are they different? Is there anything notable about the ciphertext file?

To decrypt, we use a similar command, with a **-d** instead of **-e:**

```
# openssl aes-256-cbc -d -in ciphertext -pbkdf2 -out decrypted.txt
```

Verify that decrypted.txt is the same as plaintext.txt

Part 2:
Now, instead of using a password that is converted to a key, we will be generating the key and initialization vector ourselves. We can do this using the pseudo-random number generator in OpenSSL. The following command will generate 1 hex digits, which is 8 bits.

```
# openssl rand -hex 1
```

To generate our key, we will generate 256 bits and assign the output to an environmental variable. To generate the initialization vector, we only need to generate 128 bits, because that is the block size of AES.

```
# key=$(openssl rand -hex 32)
# echo $key
# iv=$(openssl rand -hex 16)
# echo $iv
# openssl aes-256-cbc -e -K $key -iv $iv -in plaintext.txt -out ciphertext2
```

Again, to decrypt:

```
# openssl aes-256-cbc -d -K $key -iv $iv -in ciphertext2 -out decrypted2.txt
```

decrypted2.txt should look identical to **plaintext.txt**, or something went wrong.

Why might we want to use a key generated in this manner rather than a password as in part 1?

Part 3:
In this section, we will be experimenting with different modes of operation for block ciphers. We will be encrypting **Tux.bmp** that is included in this lab with both Electronic Codebook mode and Cipher Block Chaining mode and comparing the differences.

```
# openssl aes-256-ecb -in Tux.bmp -out cipher_ecb -pbkdf2
# openssl aes-256-cbc -in Tux.bmp -out cipher_cbc -pbkdf2
```

Even if we examine these with the **xxd** command, we can already clearly start to see how they are different. To further illustrate this, we need to look at the encrypted images. However, we cannot do that as the encryption process has mangled the headers that need to recognize and display these as **.bmp** images. So what we are going to do is copy the 54 bit header from Tux.bmp and replace the first 54 bits of the encrypted files.

```
$ dd if=Tux.bmp of=header bs=1 count=54
$ dd if=cipher_cbc of=cipherbody_cbc bs=1 skip=54
$ dd if=cipher_ecb of=cipherbody_ecb bs=1 skip=54
$ cat header cipherbody_cbc >cbc.bmp
$ cat header cipherbody_ecb >ecb.bmp
```

Finally, open both cbc.bmp and ecb.bmp with an image viewer.

How are they different? Why would they produce such different results?

Part 4:
This selection will require a hex editor such as bless, which can be installed with:

```
# apt install bless
```

Here we will be looking at how an error in a ciphertext effect decryption for different modes of encryption. First, let us encrypt **plaintext.txt** with AES 256 using all the different modes of encryption available in openssl.

```
# openssl aes-256-cbc -e -in plaintext.txt -pbkdf2 -out ciphertext_cbc
# openssl aes-256-ecb -e -in plaintext.txt -pbkdf2 -out ciphertext_ecb
… etc
```

Next, let us simulate a fault or corruption in each of these ciphertexts. Use a hex editor to change exactly 1 bit in the 30th byte of each ciphertext. Decrypt each corrupted file using the correct password.

```
# openssl aes-256-cbc -d -in ciphertext_cbc -pbkdf2  -out decrypted_cbc.txt
# openssl aes-256-ecb -d -in ciphertext_ecb -pbkdf2  -out decrypted_ecb.txt
… etc
```

How much information can you recover by decrypting the corrupted file for each of the different encryption modes? Please explain these differences to the best of your ability.

Deliverables:

A zip file, lastname_firstname_lab2.zip, including:

- **The answers to all questions in this document that are in red**
- **cbc.bmp**
- **ecb.bmp**
- **All of the decrypted texts from part 4**