Secret-Key Encryption Lab

Note: This document has been customized for CISC 447/866, Fall 2025.
You must use the `Labsetup.zip` file provided on onQ, and not the one that is available on the SEED Labs website. Using the incorrect file will result in a loss of grades.

1 Overview

The learning objective of this lab is to familiarize yourself with secret-key encryption concepts and some common attacks on encryption. You will gain first-hand experience with encryption algorithms, block cipher encryption modes of operation, padding, and initial vectors. You will learn about common mistakes that are made when using encryption algorithms and how to exploit such mistakes. You will also learn to write a program using a cryptography API.

Readings. Coverage of secret-key encryption can be found in Chapter 2 of *Computer Security and the Internet: Tools and Jewels from Malware to Bitcoin, Second Edition* by Paul C. van Oorschot, available at: `https://people.scs.carleton.ca/~paulv/toolsjewels/TJrev1/ch2-rev1.pdf`.

The block cipher mode of operation mentioned herein are described NIST SP 800-38A (Recommendation for Block Cipher Modes of Operation: Methods and Techniques), available at: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf`. For example, refer to Section 6.4 for the Output Feedback Mode (OFB) mode of operation.

Explanations can be found on these pages for why the CBC mode of operation is susceptible to chosen-plaintext attacks if a predictable IV is used (understanding this will be useful for Task 6.3):

- `https://defuse.ca/cbcmodeiv.htm`
- `https://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt` (see page 8)

2 Submission

Please submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. Please answer the questions in detail, and explain any observations you make that are interesting or surprising. Please also explain any code that you write. Code submitted without any accompanying explanations will result in grade deductions.

Lab Environment. This lab has been tested on our pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website.

3 Lab Environment

In this lab, we use a container to run an encryption oracle. The container is only needed in Task 6.3, so you do not need to start the container for other tasks.

Container Setup and Commands. Please download the `Labsetup.zip` file to your VM from onQ, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

```
$ docker-compose build  # Build the container images
$ docker-compose up     # Start the containers
$ docker-compose down   # Shut down the containers

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the `"docker ps"` command to find out the ID of the container, and then use `"docker exec"` to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

4 Task 0: Encryption Using Different Ciphers and Modes

In the first few tasks, you will use the `openssl enc` command to encrypt and decrypt files. Below is an example usage of the command.

```
$ openssl enc -ciphertype -e  -in plain.txt -out cipher.bin \
              -K  00112233445566778889aabbccddeeff \
              -iv 0102030405060708
```

To try different encryption algorithms and modes, replace `ciphertype` with a specific cipher type, such as `aes-128-cbc` or `aes-128-cfb`. Try at least 3 different ciphers before proceeding to Task 1. For general `openssl` instructions, you can type `man openssl`; for instructions specific to `openssl`enc and to see all the supported cipher types, you can type `man openssl enc`. Common command-line options for `openssl enc` are:

```
  -in <file>     input file
  -out <file>    output file
  -e             encrypt
  -d             decrypt
  -K/-iv         key/iv in hex is the next argument
  -[pP]          print the iv/key (then exit if -P)
```

You do not need to submit any results for this task.

5 Task 1: Encryption Mode: ECB vs. CBC

The files `pic_original1.bmp` and `pic_original2.bmp` (which appear visually identical) are included in the `Labsetup.zip` file. First, encrypt each of these two files using AES in ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes and **explain how you did so**. Then:

1. **Display both encrypted image files** with an image viewer such as `eog`. However, the first 54 bytes of a `.bmp` file must contain header information about the image, and the encrypted files will not have a valid header. So, you will need to use the headers from the original unencrypted files instead. You could use the command-line `hexeditor` program (available on Ubuntu by running `apt install ncurses-hexedit`) to modify the encrypted files. But it is easier to use the following Linux commands to get the header data from `original1.bmp` and the image data from `encrypted1.bmp` and concatenate them into a new file.

   ```
   $ head -c 54 original1.bmp  > header
   $ tail -c +55 encrypted1.bmp > body
   $ cat header body > new1.bmp
   ```

2. **Explain your observations** about what useful information can you derive about the original image from the encrypted image.

3. **Describe the visual differences** between the patterns observable in the encrypted versions of `pic_original1.bmp` and `pic_original2.bmp`. **Explain precisely what is causing the difference** (with an emphasis on the patterns, rather than the exact colours), despite the unencrypted versions of these two files looking the same.

6 Task 2: Padding

Block ciphers may require padding if the plaintext length is not a multiple of the block length. Try the following to observe how the widely-used PKCS#7[1] padding scheme works:

[1] See: `https://en.wikipedia.org/wiki/Padding_(cryptography)#PKCS#5_and_PKCS#7`

1. Use ECB, CBC, CFB, OFB, and CTR modes to encrypt a file using 128-bit AES. Report which modes have padding and which ones do not. For those that do not need padding, explain why.

2. Create three files consisting of 5, 10, and 16 bytes, respectively. The following `echo` command creates a file `f1.txt` with length 5 (without the `-n` option, `echo` appends a newline character):

   ```
   $ echo -n "12345" > f1.txt
   ```

   Then, use `openssl enc -aes-128-cbc -e` to encrypt these three files using 128-bit AES with CBC mode. Describe the size of the encrypted files.

3. Find out the padding that is added to each of the three files during encryption. To do so, you can decrypt these files using `openssl enc -aes-128-cbc -d`. Decryption automatically removes the padding by default, which prevents us from seeing it. However, you can use the `-nopad` option to disable padding during decryption, which prevents the padding from being removed. The padding may contain bytes that are invalid or unprintable characters, so you should display it in hexadecimal format. The following example shows how to display a file's contents with `hexdump`:

   ```
   $ 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.
   ```

4. Repeat steps 2–3 using 256-bit AES instead. State whether this change impacted the padding, and explain why or why not.

7 Task 3: Error Propagation

To observe the error propagation characteristics of various encryption modes, try the following:

1. Create a text file that is at least 1000 bytes long.

2. Encrypt the file using the AES-128 cipher.

3. Flip one bit of the 10th byte in the encrypted file with `hexeditor` to simulate a transmission error.

4. Decrypt the corrupted ciphertext file using the correct key and IV.

How much plaintext can you recover by decrypting the corrupted file if the encryption mode is ECB, CBC, CFB, or OFB, respectively? Explain why. Try answering this question before you start the task, and then verify your answer after completing the task.

8 Task 4: Initial Vector (IV) and Common Mistakes

Most encryption modes require an initial vector (IV). The required properties of an IV depend on the cryptographic algorithm used. Improper IV selection may result in the encrypted data being insecure, even if a secure encryption algorithm and mode of operation are used. The objective of this task is to help understand vulnerabilities that arise from improper IV selection.

8.1 Task 4.1. IV Experiment

A basic requirement for IV is *uniqueness*, which means that no IV may be reused with the same key. To understand why, encrypt the same plaintext using (1) two different IVs, and (2) the same IV. Describe your observations. Based on your observations, explain why the IV must be unique.

8.2 Task 4.2. Common Mistake: Reusing an IV

One may argue that if the plaintext does not repeat, using the same IV is safe. Let us look at the Output Feedback (OFB) mode. Assume that the attacker obtains a plaintext (`P1`) and a ciphertext (`C1`); can they decrypt other encrypted messages if the IV is always the same? Given the following information, try to figure out the actual contents of `P2` based on `C2`, `P1`, and `C1`.

```
Plaintext  (P1): This is a known message!
Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

Plaintext  (P2): (unknown to you)
Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159
```

If you replace OFB in this experiment with CFB (Cipher Feedback), how much of `P2` can be revealed? You only need to answer the question; there is no need to demonstrate that.

The attack used in this experiment is called the *known-plaintext attack*, which is an attack model for cryptanalysis where the attacker has access to both the plaintext and its encrypted version (ciphertext). If this can lead to the revealing of further secret information, the encryption scheme is not considered as secure.

Sample Code. We provide the following sample program `sample_code.py` in `Labsetup/Files`, to demonstrate how to XOR strings:

```
#!/usr/bin/python3

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

MSG   = "A message"
HEX_1 = "aabbccddeeff1122334455"
HEX_2 = "1122334455778800aabbdd"

# Convert ascii/hex string to bytearray
D1 = bytes(MSG, 'utf-8')
D2 = bytearray.fromhex(HEX_1)
D3 = bytearray.fromhex(HEX_2)

r1 = xor(D1, D2)
r2 = xor(D2, D3)
r3 = xor(D2, D2)
print(r1.hex())
print(r2.hex())
print(r3.hex())
```

8.3 Task 4.3. Common Mistake: Using a Predictable IV

Another important requirement of many encryption algorithms is that the IV must be unpredictable (i.e., it should be randomly generated). This task demonstrates what happens if the IV is predictable.

Suppose Bob sends an encrypted message to Alice, and Eve knows that the the message must decrypt to either `Yes` or `No`. Eve can see the ciphertext and the IV, but since AES is a secure cipher, she cannot decrypt the ciphertext. However, since Bob uses predictable IVs, Eve knows the IV that Bob will use next.

A good cipher should not only resist known-plaintext attacks but also *chosen-plaintext attacks*, where the attacker can obtain the ciphertext for an arbitrary plaintext. Since AES is a strong cipher that can resist the chosen-plaintext attack, Bob does not mind encrypting any plaintext given by Eve; he uses a different IV for each plaintext, but unfortunately, the IVs he generates are not random, and are thus predictable.

Your task is to construct a message and ask Bob to encrypt it and give you the ciphertext. Your objective is to use this opportunity to figure out whether Bob's secret message decrypts to `Yes` or `No`. For this task, you are given an encryption oracle which simulates Bob and encrypts message with 128-bit AES with CBC mode. You can access the oracle by running the following command:

```
$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 54601f27c6605da997865f62765117ce
The IV used    : d27d724f59a84d9b61c0f2883efa7bbc

Next IV        : d34c739f59a84d9b61c0f2883efa7bbc
Your plaintext : 11223344aabbccdd
Your ciphertext: 05291d3169b2921f08fe34449ddc3611

Next IV        : cd9f1ee659a84d9b61c0f2883efa7bbc
Your plaintext : <your input>
```

The `nc` command above connects to the IP address `10.9.0.80`[2] on TCP port `3000`, prints out the bytes received, and send the bytes that it reads from your keyboard input. Consult `man nc` for the complete documentation.

After printing the next IV, the oracle will ask you to input a plaintext message as a hexadecimal string. The oracle will encrypt the message with the next IV and output the new ciphertext. You can try different plaintexts, and the IV will change each time but will be predictable. To simplify the task, we let the oracle print the next IV. To exit from the interaction, press Ctrl+C.

9 Task 5: Programming Using a Cryptography Library

Below is a ciphertext and an IV, both in hexadecimal format:

```
Ciphertext (in hex format, with newline added to make it fit on the page):
2a94d3a3df0a72b3535c1bbcd873f09f5f9dd18d68d240e662190bb36277
20498faa86b2cc0ff0d6cf8334a279d482b8

IV (in hex format):
aabbccddeeff00998877665544332211
```

Your task is to crack the encryption key and recover the plaintext. You know that:

[2]The IP address is in the private IPv4 address range (similar to the `192.168.0.0` range). See this article for more information: `https://en.wikipedia.org/wiki/Private_network`

- The `aes-128-cbc` cipher is used for the encryption.
- The encryption key is a sequence of two English dictionary words, with `#` characters (hexadecimal value `0x23`) appended to pad it to 16 bytes (this is a terrible way of choosing a cryptographic key—but that is what makes this task possible). A sample key following this method would be `forexample######`. You may assume that there is padding, which rules out any word or pair of words that exceed 15 characters.

Write a program to **find the encryption key and also print out how long it took to find it**. No credit will be given if you just use the `openssl` commands to complete this task. Sample code can be found at `https://www.openssl.org/docs/man1.1.1/man3/EVP_CipherInit.html`.[3] You will also need an English word list: you may find a suitably large English dictionary (around 100K words) on Ubuntu located in the `/usr/share/dict` directory.

When you compile your code with `gcc`, you must include the `-lcrypto` option to use the `crypto` library, and it is also advisable to use the `-fsanitize=address` option to help catch pointer bugs:

```
$ gcc -o myenc myenc.c -fsanitize=address -lcrypto
```

9.1 Encryption Method

The following commands were used to generate the ciphertext given above; `example` is used in place of the actual secret word that was used to construct the key. When using different words, ensure that you add the correct number of # characters to complete the string length to 16 bytes:

```
$ echo -n "This is an example plaintext." > plaintext.txt
$ echo -n "forexample######" > key
$ xxd -p key
6578616d706c65232323232323232323
$ openssl enc -aes-128-cbc -e -in plaintext.txt -out ciphertext.bin \
      -K  6578616d706c65232323232323232323  \
      -iv 01020304050607080900a0b0c0d0e0f  \
$ xxd -p ciphertext.bin
e5accdb667e8e569b1b34f423508c15422631198454e104ceb658f5918800c22
```

We provide the secret key to the `openssl` command in hexadecimal form (not as an ASCII string). Also, many text editors add a newline character to the end of the file. The easiest way to store a string in a file without appending a newline character is to use `echo` with the `-n` option, as follows:

```
$ echo -n "This is a top secret." > file
```

10 Acknowledgment

We would like to acknowledge the contribution made by the following people and organizations:

- Jiamin Shen developed the following: the code running inside the container, and the container version of the task on predictable IV.

[3] Hint from Prof. Alaca: `unsigned char iv[] = "1234";` instantiates a 4-byte array (5 bytes if you count the null terminator), where each ASCII character in the string represents one byte. In contrast, `unsigned char iv[] = {1, 2, 3, 4};` instantiates a 4-byte array (no null terminator) and will **not** contain the same values as the previous string.

- The US National Science Foundation provided the funding for the SEED project from 2002 to 2020.
- Syracuse University provided the resources for the SEED project from 2001 onwards.

This document was modified by F. Alaca for use in the Fall 2024 offering of CISC 447/866 (Introduction to Cybersecurity) at Queen's University. The original document is found here: `https://seedsecuritylabs.org/Labs_20.04/Crypto/Crypto_Encryption/`