Symmetric Key Encryption Lab rev2

1. Overview

The learning objective of this lab is for students to get familiar with the concepts in the secret-key encryption, block modes for block ciphers, and some common attacks on encryption. From this lab, students will gain a first-hand experience on encryption algorithms, encryption modes, paddings, and initial vector (IV) use and handling. Moreover, students will be able to use tools and write programs to encrypt/decrypt messages using provided templates.

Many common mistakes have been made by developers in using the encryption algorithms and modes. These mistakes weaken the strength of the encryption, and eventually lead to vulnerabilities. This lab exposes students to some of these mistakes and ask students to launch attacks to exploit those vulnerabilities.

This lab covers the following topics:

- Secret-key encryption
- Substitution cipher and frequency analysis
- Encryption modes, IV, and paddings
- Common mistakes in using encryption algorithms
- Programming using the crypto library

Lab environment: This lab has been tested on the SEED[1] Ubuntu 20.04 VM.

The seed Account: In this lab, we need to telnet from one container to another. We have already created an account called seed inside all the containers.
Lab_symmetric.zip is also provided on Canvas with necessary files for this lab.

The username is seed and password is dees for all VMs and docker containers.
Docker Manual for the Lab [here][2]

This lab is part of Assignment 2 so please keep screenshots of your steps while you perform the lab to show your work.

Please make sure that you have completed Lab 1 before you proceed in this lab!

[1] This Lab exercise has been adopted from SEED Labs by Dr. Wenliang Du (https://seedsecuritylabs.org)
[2] https://github.com/seed-labs/seed-labs/blob/master/manuals/docker/SEEDManual-Container.md

2. Lab Environment

In this lab, we use a container to run an encryption oracle and a padding 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 Lab_symmetric.zip file to your VM from the course CANVAS website, unzip it, enter the Lab_symmetric 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.

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).

Container Setup and Commands

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).

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 for you to be able to use the commands faster.

```
$ 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.

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

// 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.

1. [5 pts] Task 1: Encryption using Different Ciphers and Modes

In this task, we will experiment with various encryption algorithms and modes. You can use the following `openssl enc` command to encrypt/decrypt a file. To see the manuals, you can type `man openssl` and `man enc`.

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

Please replace the ciphertype with a specific cipher type, such as -aes-128-cbc, -bf-cbc, -aes-128-cfb, etc. In this task, you should try at least 3 different ciphers. You can find the meaning of the command-line options and all the supported cipher types by typing "man enc". We include some common options for the openssl enc command in the following:

```
-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)
```

2. [10 pts] Task 2: Encryption Mode – ECB vs CBC

The file pic_original.bmp is included in the **Lab_symmetric.zip** file, and it is a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

1. Let us treat the encrypted picture as a picture and use a picture viewing software to display it. However, For the .bmp file, the first 54 bytes contain the header information about the picture, we have to set it correctly, so the encrypted file can be treated as a legitimate .bmp file. We will replace the header of the encrypted picture with that of the original picture. We can use the bless hex editor tool (already installed on our VM) to directly modify binary files. We can also use the following commands to get the header from p1.bmp, the data from p2.bmp (from offset 55 to the end of the file), and then combine the header and data together into a new file.

```
$ head -c 54 p1.bmp  > header
$ tail -c +55 p2.bmp > body
$ cat header body > new.bmp
```

2. Display the encrypted picture using a picture viewing program (we have installed an image viewer program called **eog** on our VM). Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

Select a picture of your choice, repeat the experiment above, and report your observations.

3. [20pts] Task 3: Implementing Different Cipher Modes using basic Encryption Blocks (provided as template)

In this task, you will implement two encryption/decryption systems, one using AES 256 in Cipher Block Chain (CBC) mode and another using AES in Output Feedback (OFB) mode. In both cases the 16-byte encryption both the IV for CBC and the nonce are chosen at random. For the AES CBC encryption you need to implement the PKCS7 (which is the same as PKCS5 only longer) padding scheme. The current templates do not support any padding.

To complete, you can start using the basic AES 256 encryption commands provided in two templates **AES_basic.c and AES_basic.py** for C and Python, respectively. These are in the **Lab_symmetric.zip** file provided with the lab on Canvas.

There are examples of (Key, IV, Ciphertext) and (Key, Nonce, Ciphertext) included see **Task3.txt** file of the lab. Please use them to validate the decryption part of your work. Finally, to validate that you implemented the encryption portion of the algorithm properly, please include a run of the algorithm using a plaintext, IV (or Nonce), and Key of your choice for both AES CBC and OFB with your answer.

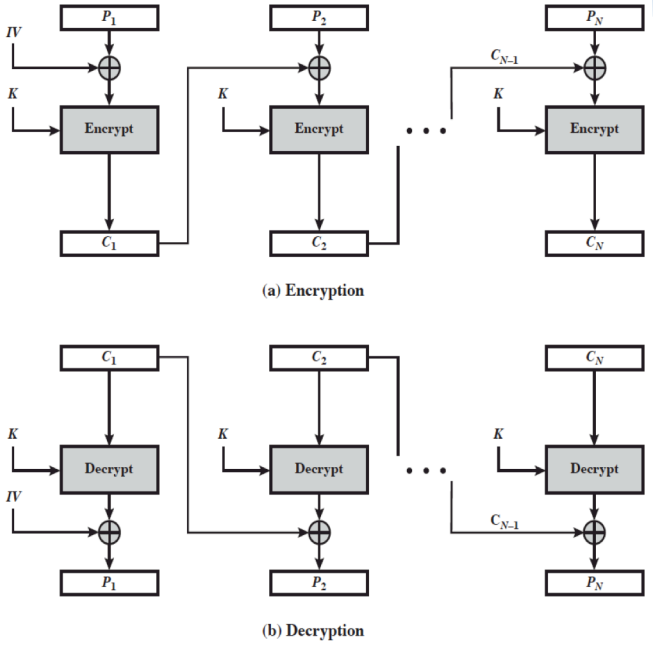

Figure 7.4 Cipher Block Chaining (CBC) Mode

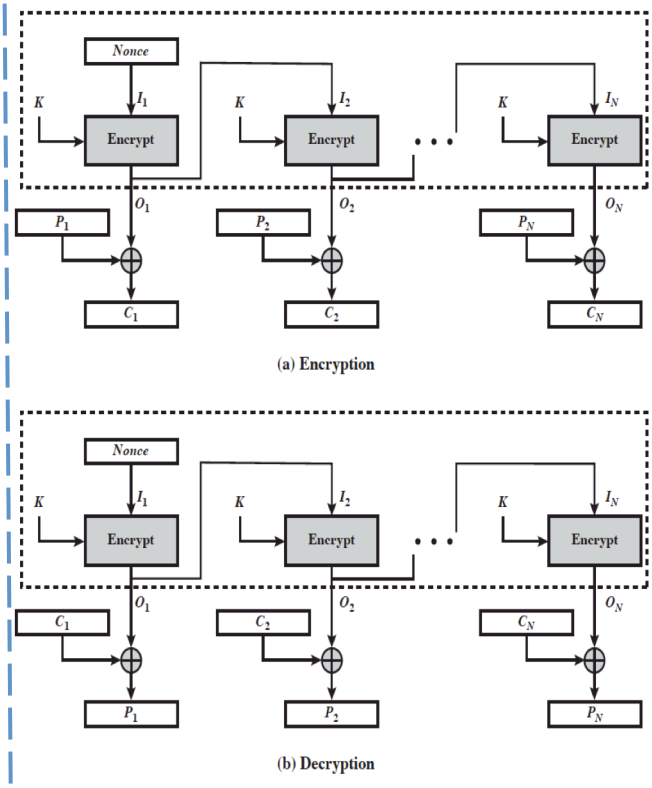

Figure 7.6 Output Feedback (OFB) Mode

Your answer for this task should contain both your code and the decrypted/encrypted values we request.

4. [10 pts] Task 4: Padding

For block ciphers, when the size of a plaintext is not a multiple of the block size, padding may be required. The PKCS#5 padding scheme is widely used by many block ciphers. We will conduct the following experiments to understand how this type of padding works:

1. Use ECB, CBC, CFB, and OFB modes to encrypt a file (you can pick any cipher). Please report which modes have paddings and which ones do not. For those that do not need paddings, please explain why.

2. Let us create three files, which contain 5 bytes, 10 bytes, and 16 bytes, respectively. We can

use the following "echo -n" command to create such files. The following example creates a file f1.txt with length 5 (without the -n option, the length will be 6, because a newline character will be added by echo):

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

We then use "openssl enc -aes-128-cbc -e" to encrypt these three files using 128-bit AES with CBC mode. Please describe the size of the encrypted files.

We would like to see what is added to the padding during the encryption. To achieve this goal, we will decrypt these files using "openssl enc -aes-128-cbc -d".

Unfortunately, decryption by default will automatically remove the padding, making it impossible for us to see the padding. However, the command does have an option called "-nopad", which disables the padding, i.e., during the decryption, the command will not remove the padded data. Therefore, by looking at the decrypted data, we can see what data are used in the padding. Please use this technique to figure out what paddings are added to the three files.

It should be noted that padding data may not be printable, so you need to use a hex tool to display the content. The following example shows how to display a file in the hex format:

```
$ 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.
```

5. [10 pts] Task 5: Error Propagation – Corrupted Cipher Text

To understand the error propagation property of various encryption modes, we would like to do the following exercise:

1. Create a text file that is at least 1000 bytes long.
2. Encrypt the file using the AES-128 cipher.
3. Unfortunately, a single bit of the 42nd byte in the encrypted file got corrupted. You can achieve this corruption using the bless hex editor.
4. Decrypt the corrupted ciphertext file using the correct key and IV.

Please answer the following question: How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively? Please answer this question before you conduct this task, and then find out whether your answer is correct or

wrong after you finish this task. Please provide justification.

6. [15 pts + 5 Extra Credit] Task 6: Initial Vector (IV) and Common Mistakes

Most of the encryption modes require an initial vector (IV). Properties of an IV depend on the cryptographic scheme used. If we are not careful in selecting IVs, the data encrypted by us may not be secure at all, even though we are using a secure encryption algorithm and mode. The objective of this task is to help students understand the problems if an IV is not selected properly.

Task 6.1. [5 pts] IV Experiment

A basic requirement for IV is *uniqueness*, which means that no IV may be reused under the same key. To understand why, please encrypt the same plaintext using (1) two different IVs, and (2) the same IV. Please describe your observation, based on which, explain why IV needs to be unique.

Task 6.2. [5 pts] Common Mistake: Use the Same 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 gets hold of a plaintext (P1) and a ciphertext (C1), can he/she decrypt other encrypted messages if the IV is always the same? You are given the following information, please try to figure out the actual content 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 we 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 a sample program called sample code.py, which can be found inside the Lab_symmetric/Files folder. It shows you how to XOR strings (ascii strings and hex strings). The code is shown in the following:

```
#!/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())
```

Task 6.3. [10 pts] Common Mistake: Use a Predictable IV

From the previous tasks, we now know that IVs cannot repeat. Another important requirement on IV is that IVs need to be unpredictable for many schemes, i.e., IVs need to be randomly generated. In this task, we will see what is going to happen if IVs are predictable.

Assume that Bob just sent out an encrypted message, and Eve knows that its content is either Yes or No; Eve can see the ciphertext and the IV used to encrypt the message, but since the encryption algorithm AES is quite strong, Eve has no idea what the actual content is. However, since Bob uses predictable IVs, Eve knows exactly what IV Bob is going to use next.

A good cipher should not only tolerate the known-plaintext attack described previously, but it should also tolerate the *chosen-plaintext attack*, which is an attack model for cryptanalysis where the attacker can obtain the ciphertext for an arbitrary plaintext. Since AES is a strong cipher that can tolerate the chosen-plaintext attack, Bob does not mind encrypting any plaintext given by Eve; he does use a different IV for each plaintext, but unfortunately, the IVs he generates are not random, and they can always be predictable.

Your job 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 the actual content of Bob's secret message is 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 get access to 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>
```

After showing you the next IV, the oracle will ask you to input a plaintext message (as a hex string). The oracle will encrypt the message with the next IV, and outputs the new ciphertext. You can try different plaintexts, but keep in mind that every time, the IV will change, but it is predictable. To simply your job, we let the oracle print out the next IV. To exit from the interaction, press Ctrl+C.

Additional Readings (Optional)

There are more advanced cryptanalysis papers and studies on IV that is beyond the scope of this lab. Students can read the article posted in this URL:

https://defuse.ca/cbcmodeiv.htm.

Because the requirements on the choice of the IV really depend on cryptographic schemes, it is hard to remember what properties should be maintained when we select an IV. However, we will be safe if we always use a new IV for each encryption, and the new IV needs to be generated using a good pseudo random number generator, so it is unpredictable by adversaries.

7. Lab Report as part of Assignment 2

As part of assignment two that will be posted on Canvas, you need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.