Secret-Key Encryption Lab

1 Overview

The learning objective of this lab is for students to get familiar with the concepts in the secret-key encryption 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). Moreover, students will be able to use tools and write programs to encrypt/decrypt messages.

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

Readings. Detailed coverage of the secret-key encryption can be found in the following:

- Chapter 21 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.

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

Report Requirement.

1. You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed, through CANVAS. "Student Name" and "Student ID" should be shown in the beginning.

2. 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.

3. Do not copy from the Internet, lecture notes, or others. Please use your words in your solutions. **You are required to include an honesty statement in submission (you need to type your name to xxx).**

4. Some files, data, keys are changed from the original SeedLab instructions. Copying the answers from the Internet won't work.

Grading rubrics. The total points are 100. Task 1: 20 points. Task 2: 15 points. Task 3: 15 points. Task 4: 15 points. Task 5: 15 points. Task 6: 20 points. Partial credit will be given if there is no explanation or screenshot, even if you get the right answer.

Due. 01/31/2024, 23:59:59PM PST

2 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 zip file to your VM from the course website, unzip it, enter the 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 (`https://github.com/seed-labs/seed-labs/blob/master/manuals/docker/SEEDManual-Container.md`). 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 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.

3 Task 1: Frequency Analysis

It is well-known that monoalphabetic substitution cipher (also known as monoalphabetic cipher) is not secure, because it can be subjected to frequency analysis. In this lab, you are given a cipher-text that is encrypted using a monoalphabetic cipher; namely, each letter in the original text is replaced by another letter, where the replacement does not vary (i.e., a letter is always replaced by the same letter during the encryption). Your job is to find out the original text using frequency analysis. It is known that the original text is an English article.

In the following, we describe how we encrypt the original article, and what simplification we have made. Instructors can use the same method to encrypt an article of their choices, instead of asking students to use the ciphertext made by us.

- Step 1: let us generate the encryption key, i.e., the substitution table. We will permute the alphabet from `a` to `z` using Python, and use the permuted alphabet as the key. See the following program.

```
#!/bin/env python3

import random
s = "abcdefghijklmnopqrstuvwxyz"
list = random.sample(s, len(s))
key = ''.join(list)
print(key)
```

- Step 2: let us do some simplification to the original article. We convert all upper cases to lower cases, and then removed all the punctuations and numbers. We do keep the spaces between words, so you can still see the boundaries of the words in the ciphertext. In real encryption using monoalphabetic cipher, spaces will be removed. We keep the spaces to simplify the task. We did this using the following command:

```
$ tr [:upper:] [:lower:] < article.txt > lowercase.txt
$ tr -cd '[a-z][\n][:space:]' < lowercase.txt > plaintext.txt
```

- Step 3: we use the `tr` command to do the encryption. We only encrypt letters, while leaving the space and return characters alone.

```
$ tr 'abcdefghijklmnopqrstuvwxyz' 'sxtrwinqbedpvgkfmalhyuojzc' \
      < plaintext.txt > ciphertext.txt
```

We have created a ciphertext **using a different encryption key (not the one described above)**. It is included in `hw1.zip` file, which can be downloaded from the lab's website. Your job is to use the frequency analysis to figure out the encryption key and the original plaintext.

We have also provided a Python program (`freq.py`) inside the `hw1/Files` folder. It reads the `ciphertext.txt` file, and produces the statistics for n-grams, including the single-letter frequencies, bigram frequencies (2-letter sequence), and trigram frequencies (3-letter sequence), etc.

Guidelines. Using the frequency analysis, you can find out the plaintext for some of the characters quite easily. For those characters, you may want to change them back to its plaintext, as you may be able to get more clues. It is better to use capital letters for plaintext, so for the same letter, we know which is plaintext and which is ciphertext. You can use the `tr` command to do this. For example, in the following, we replace letters `a`, `e`, and `t` in `in.txt` with letters `X`, `G`, `E`, respectively; the results are saved in `out.txt`.

```
$ tr 'aet' 'XGE' < in.txt > out.txt
```

There are many online resources that you can use. We list some useful links in the following:

- `https://en.wikipedia.org/wiki/Frequency_analysis`: This Wikipedia page provides frequencies for a typical English plaintext.
- `https://en.wikipedia.org/wiki/Bigram`: Bigram frequency.
- `https://en.wikipedia.org/wiki/Trigram`: Trigram frequency.

4 Task 2: Encryption using Different Ciphers and Modes

In this task, we will play 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)
```

5 Task 3: Encryption Mode – ECB vs. CBC

The file `pic_original.bmp` is included in the `hw1.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.

6 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 (see Chapter 21.4 of the SEED book for details). 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.
   ```

7 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 55th 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.

8 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. The detailed guidelines for this task is provided in Chapter 21.5 of the SEED book.

8.1 Task 6.1. 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.

8.2 Task 6.2. 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 `hw1/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())
```

8.3 Task 6.3. 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, 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, your 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, after the docker (through `dcup`) is brought up:

```
$ 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.

8.4 Additional Readings

There are more advanced cryptanalysis 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 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. See another SEED lab (Random Number Generation Lab) for details on how to generate cryptographically strong pseudo random numbers.

9 Acknowledgment

This lab is adjusted from the SeedLab of Prof. Wenliang Du.