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.
After finishing the lab, students should be able to 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. This lab covers the following topics:

- Secret-key encryption
- Encryption modes and paddings
- Programming using the crypto library
- Random number generation

Lab Environment.   This lab has been tested on our pre-built Ubuntu 12.04 VM and Ubuntu 16.04 VM,
both of which can be downloaded from the SEED website.

2 Lab Tasks

2.1 Task 1: 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`, `-aes-128-cfb`,
`-bf-cbc`, etc. In this task, you should try at least 3 different ciphers and three different modes. 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)
```

2.2 Task 2: Encryption Mode – ECB vs. CBC

The file `pic_original.bmp` contains 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.

2.3 Task 3: Encryption Mode – Corrupted Cipher Text

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

1. Create a text file that is at least 64 bytes long.

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

3. Unfortunately, a single bit of the 30th byte in the encrypted file got corrupted. You can achieve this corruption using a hex editor.

4. Decrypt the corrupted file (encrypted) using the correct key and IV.

Please answer the following questions: (1) 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. (2) Please explain why. (3) What are the implication of these differences?

2.4 Task 4: Padding

For block ciphers, when the size of the plaintex is not the multiple of the block size, padding may be required. In this task, we will study the padding schemes. Please do the following exercises:

1. The `openssl` manual says that `openssl` uses PKCS5 standard for its padding. Please design an experiment to verify this. In particular, use your experiment to figure out the paddings in the AES encryption when the length of the plaintext is 20 bytes and 32 bytes.

2. Please 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.

2.5 Task 5: Pseudo Random Number Generation

Generating random numbers is a quite common task in security software. In many cases, encryption keys are not provided by users, but are instead generated inside the software. Their randomness is extremely important; otherwise, attackers can predict the encryption key, and thus defeat the purpose of encryption. Many developers know how to generate random numbers (e.g. for Monte Carlo simulation) from their prior experiences, so they use the similar methods to generate the random numbers for security purpose. Unfortunately, a sequence of random numbers may be good for Monte Carlo simulation, but they may be bad for encryption keys. Developers need to know how to generate secure random numbers, or they will make mistakes. Similar mistakes have been made in some well-known products, including Netscape and Kerberos.

In this task, students will learn a standard way to generate pseudo random numbers that are good for security purposes.

Task 5.A: Measure the Entropy of Kernel

To generate good pseudo random numbers, we need to start with something that is random; otherwise, the outcome will be quite predictable. Software (i.e. in the virtual world) is not good at creating randomness, so most systems resort to the physical world to gain the randomness. Linux gains the randomness from the following physical resources:

```
void add_keyboard_randomness(unsigned char scancode);
void add_mouse_randomness(__u32 mouse_data);
void add_interrupt_randomness(int irq);
void add_blkdev_randomness(int major);
```

The first two are quite straitforward to understand: the first one uses inter-keypress timing and scancode, and the second one uses mouse movement and interrupt timing . The third one gathers random numbers using the interrupt timing. Of course, not all interrupts are good sources of randomness. For example, the timer interrupt is not a good choice, because it is predictable. However, disk interrupts are a better measure. The last one measures the finishing time of block device requests.

The randomness is measured using *entropy*, which is different from the meaning of entropy in the information theory. Here, it simply means how many bits of random numbers the system currently has. You can find out how much entropy the kernel has at the current moment using the following command.

```
% cat /proc/sys/kernel/random/entropy_avail
```

Please move and click your mouses, type somethings, and run the program again. Please describe your observation in your report.

Task 5.B: Get Pseudo Random Numbers from `/dev/random`

Linux stores the random data collected from the physical resources into a random pool, and then uses two devices to turn the randomness into pseudo random numbers. These two devices have different behaviors. In this subtask, we study the /dev/random device.

You can use the following command to get 16 bytes of pseudo random numbers from /dev/random. We pipe the data to hexdump to print them out.

```
$ head -c 16 /dev/random | hexdump
```

Please run the above command several times, and you will find out that at some point, the program will not print out anything, and instead, it will be waiting. Basically, every time a random number is given out by `/dev/random`, the entropy of the randomness pool will be decreased. When the entropy reaches zero, `/dev/random` will block, until it gains enough randomness. Please show us how you can get `/dev/random` to unblock and to print out random data.

Task 5.C: Get Random Numbers from `/dev/urandom`

`Linux` provides another way to access the random pool via the `/dev/urandom` device, except that this device will not block, even if the entropy of the pool runs low.

You can use the following command to get 1600 bytes of pseudo random numbers from `/dev/urandom`. You should run it several times, and report whether it will block or not.

```
% head -c 1600 /dev/urandom | hexdump
```

Both `/dev/random` and `/dev/urandom` use the random data from the pool to generate pseudo random numbers. When the entropy is not sufficient, `/dev/random` will pause, while `/dev/urandom` will keep generating new numbers. Think of the data in the pool as the "seed", and as we know, you can use a seed to generate as many pseudo random numbers as you want. Theoretically speaking, the `/dev/random` device is more secure, but in practice, there is not much difference, because the "seed" is random and non-predictable. `/dev/urandom` does re-seed whenever new random data become available. The fact that `/dev/random` blocks may lead to denial of service attacks.

It is recommended that you use `/dev/urandom` to get random numbers. To do that in your program, you just need to read directly from this file. The following code snippet shows you how.

```
#define LEN 16   // 128 bits

unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char)*LEN);
FILE* random = fopen("/dev/urandom", "r");
fread(key, sizeof(unsigned char)*LEN, 1, random);
fclose(random);
```

2.6 Task 6: Programming using the Crypto Library

In this task, you are given a plaintext and a ciphertext, and you know that `aes-128-cbc` is used to generate the ciphertext from the plaintext, and you also know that the numbers in the IV are all zeros (not the ASCII character '0'). Another clue that you have learned is that the key used to encrypt this plaintext is an English word shorter than 16 characters; the word that can be found from a typical English dictionary. Since the word has less than 16 characters (i.e. 128 bits), space characters (hexadecimal value 0x20) are appended to the end of the word to form a key of 128 bits. Your goal is to write a program to find out this key. You can download a English word list from the Internet. We have also linked one on the web page of this lab. The plaintext and ciphertext is in the following:

```
Plaintext (total 21 characters): This is a top secret.
Ciphertext (in hex format): 8d20e5056a8d24d0462ce74e4904c1b5
                            13e10d1df4a2ef2ad4540fae1ca0aaf9
```

Note 1: If you choose to store the plaintex message in a file, and feed the file to your program, you need to check whether the file length is 21. Some editors may add a special character to the end of the file. If that

happens, you can use a hex editor tool to remove the special character.

Note 2: In this task, you are supposed to write your own program to invoke the crypto library. No credit will be given if you simply use the openssl commands to do this task.

Note 3: When you compile your code using gcc, do not forget to include the -lcrypto flag, because your code needs the crypto library. See the following example:

```
$ gcc -o enc mycode.c -lcrypto
```

3   Submission

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.