Programming Project 1 for Introduction to Computer Security

1 Overview

The learning objective of this project is for students to get familiar with using secret-key encryption and one-way hash functions in the openssl library. After finishing the assignment, in addition to improving their understanding of secret-key encryption and one-way hash function concepts, you should be able to use tools and write programs to generate one-way hash value and encrypt/decrypt messages.

2 Submission Guidelines

This project has a total of 60 points with another 200 points for optional tasks.

You need to submit a report answer the questions associated with the tasks 3.1-3.3 and 3.5-3.7 describing your observations for each task. Please submit your assignment as a single PDF file. Any submissions other than a single PDF file will not be graded. You also need to provide explanations to the observations that are interesting or surprising.

Tasks 3.4 and 3.8 are optional. For these you have to write your own code in the language of your choice and implement it.

This is due on Wednesday of Week 4 at midnight.

3 Lab Environment and Tasks

Installing OpenSSL. In this MP, you will use openssl commands and libraries. Please install the latest version of openssl compatible with your platform. It should be noted that if you want to use openssl libraries in your programs, you need to install several other things for the programming environment, including the header files, libraries, manuals, etc.

3.1 [10pts] Observation Task: Encryption using different ciphers and modes

In this exercise, you 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 \
-iv 0102030405060708
```

Please replace the ciphertype with a specific cipher type, such as -aes-128-cbc, -aes-128-cfb, -bf-cbc, etc. 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)

What to include in your submission for 3.1:
Include the plaintext used for encryption and the cipher test obtained using 3 different encryption modes/algorithms that you tried

	Plaintext	Encryption Modes	Cipher Result
1.	Hello World	-aes-128-cbc	0000000 260 317 253 214 261 361 256 g 220 372 250 \t $ = R 366 0000010
2.	Hello World	-aes-128-cfb	0000000 351 036 352 Q o 346 Z ; 027 [235 r < 000000d
3.	Hello World	-des3	0000000 030 I E N 221 x 0 9 027 034 327 c 335 370 300 314 0000010

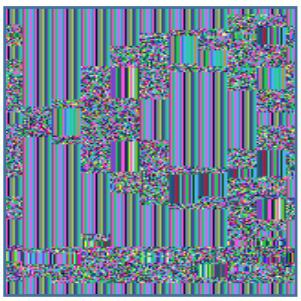
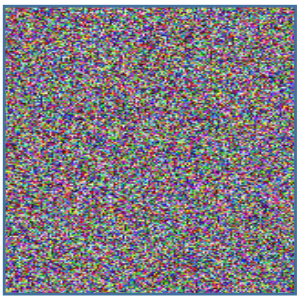
3.2 [10pts] Observation Task: Encryption Mode – ECB vs. CBC

Obtain a simple picture in .bmp format. 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, you have to set it correctly, so the encrypted file can be treated as a legitimate .bmp file. Replace the header of the encrypted picture with that of the original picture. You can use a hex editor tool (e.g. ghex or Bless) to directly modify binary files.
2. Display the encrypted picture using any picture viewing software. Can you derive any useful infor-mation about the original picture from the encrypted picture? Record both encrypted pictures and the original picture for your report.

```
openssl enc -aes-256-ecb -in ProfilePic.bmp -out ProfilePicECB.bmp
openssl enc -aes-256-cbc -in ProfilePic.bmp -out ProfilePicCBC.bmp
$ dd if=ubuntu.bmp of=ubuntu-ecb.bmp bs=1 count=54 conv=notrunc
$ dd if=ubuntu.bmp of=ubuntu-cbc.bmp bs=1 count=54 conv=notrunc
```

Plaintext Original v.s ECB v.s CBC

What to include in your submission for 3.2:
Include the encrypted pictures with CBC and ECB mode along with the original picture.

3.3 [10 pts] Observation Task: CBC Encryption using different IVs

IVs or initialization vectors act as a random seed to the block cipher modes of encryption like CBC. In this task you will understand the role of IVs.

- Step 1: Create any file with one line of text and name it plain.txt, the content of the file doesn't matter.
- Step 2: Now, using the command in the Q3.1, encrypt the file using aes-128-cbc scheme and write the output to the file 'encrypt1.bin'. You are free to choose the Key and IV in this step, but note them down.
- Step 3: Now using the same Key and IV in the step 2, encrypt the file again and write the output to 'encrypt2.bin'.
- Step 4: Now using the same Key in Step 2 and a different IV, encrypt the file again and write your output to 'encrypt3.bin'.

What to include in your submission for 3.3:
Now based on your observations of the above steps and the 3 different files that you generated, answer the following questions:

1. Does the contents of 'encrypt1.bin' match the contents of 'encrypt2.bin'? Explain why or why not.

   Yes it does. Because CBC or cipher-block chaining utilizes the initializing vector to randomize, the key won't matter here. Because each block isn't encrypt with the same algorithm, it returns the same result.

2. Does the contents of 'encrypt1.bin' match the contents of 'encrypt3.bin'? Explain why or why not

   No. Because CBC utilizing IV for randomizing output. The IV is hashed to provide 128 bits output then AES encrypt the hash, the retuning result for the first 128 is then used for the following 128 bits, therefore, always returning seemingly random result.

1. openssl enc -aes-128-cbc -e -in plain.txt -out encrypt1.bin -K 123456789 -iv 013579
2. openssl enc -aes-128-cbc -e -in plain.txt -out encrypt2.bin -K 123456789 -iv 013579
3. openssl enc -aes-128-cbc -e -in plain.txt -out encrypt3.bin -K 123456789 -iv 145853

Results where 1 is same key same IV, 2 is same key same IV and 3. Is same key different iv

1. 0000000 243 270 x 306 345 001 2 017 321 304 035 1 \0 360 \f 0000010

2. 0000000 243 270 x 306 345 001 2 017 321 304 035 1 \0 360 \f 0000010

3. 0000000 252 273 t 4 7 327 341 v 363 211 " \v 027 | 1 U 0000010

3.4. [OPTIONAL] [80pts] Coding Task: Encrypting with OpenSSL

So far, we have learned how to use the tools provided by openssl to encrypt and decrypt messages. In this task, we will learn how to use openssl's crypto library to encrypt/decrypt messages in programs.

OpenSSL provides an API called EVP, which is a high-level interface to cryptographic functions. Although OpenSSL also has direct interfaces for each individual encryption algorithm, the EVP library provides a common interface for various encryption algorithms. To ask EVP to use a specific algorithm, we simply need to pass our choice to the EVP interface. A sample C code is given in http://www.openssl. org/docs/crypto/EVP_EncryptInit.html. Please get yourself familiar with this program, and then do the following exercise.

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 an English word list from the Internet. We have also linked one here http://www.cis.syr.edu/~wedu/seed/Labs/Crypto/Crypto_Encryption/files/words.txt.
The plaintext and ciphertext are 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 plaintext 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: To compile your code, you may need to include the header files in openssl, and link to openssl libraries. To do that, you need to tell your compiler where those files are. In your Makefile, you may want to specify the following:

```
INC=/usr/local/ssl/include/ or the actual location in your system
LIB=/usr/local/ssl/lib/ or the actual location in your system
```

```
all:
        gcc -I$(INC) -L$(LIB) -o enc yourcode.c -lcrypto –ldl
```

3.5 [10pts] Observation Task: Generating Message Digest and MAC

In this exercise, you will generate message digests using a few different one-way hash algorithms. You can use the following openssl dgst command to generate the hash value for a file. To see the manuals, you can type man openssl and man dgst.

```
% openssl dgst dgsttype filename
```

Please replace the dgsttype with a specific one-way hash algorithm, such as -sha1, -sha256, etc. In this exercise, you should try at least 3 different algorithms, and note your observations. You can find the supported one-way hash algorithms by typing "man openssl". 512oe

What to include in your submission for 3.5:
Include the plaintext used and the hash digest obtained using 3 different hash algorithms that you tried.
Plain.txt = “Hello Again Again”

openssl dgst -sha1 plain.txt
SHA1(plain.txt)= dbf279ef4be208cad8adaa8ffc37b8c06e899c3c

openssl dgst -sha256 plain.txt
SHA256(plain.txt)=
3107ed35ab9c31623560b83daf1aa7b97d717ab45d62b9d2e077014c627e7be3

openssl dgst -sha512 plain.txt
SHA512(plain.txt)=
05d0635ff60800d3014fc875076c38a8bf958686c54746b01ff594173d46267fab960fb9874e0faaa5
1b45bec29689af3d1261316db8247e50ace422dde63aeb

3.6 [10pts] Observation Task: Keyed Hash and HMAC

In this exercise, you will generate a keyed hash (i.e. MAC) for a file. Please generate a keyed hash using HMAC-SHA256, and HMAC-SHA1 for any file that you choose. Please try several keys with different length.

What to include in your submission for 3.6:
Answer the following question. Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why?

The key should be the same size as the hash output so if we use sha256, the key size should be 256-bit. You could technically use a key of any size though. I tried not using a key and it did not compile. Also 128 bit keys is sufficient for brute force attack prevention. Note: turns out its okay to enter “” for the key but I didn’t include that part so it didn’t load.

You can use the -hmac option of openssl command line. The following example generates a keyed hash for a file using the HMAC-SHA256 algorithm. The string following the -hmac option is the key.

```
% openssl dgst –sha256 -hmac "abcdefg" filename
```

3.7. [10pts] Observation Task: The Randomness of One-way Hash

To understand the properties of one-way hash functions, do the following exercise for SHA512 and SHA256:

1. Create a text file with the following text -- 'The cow jumps over the moon'
2. Generate the hash value H_1 for this file using SHA512 (SHA256).
3. Flip one bit of the input file. You can achieve this modification using hex editors like ghex or Bless.
4. Generate the hash value H_2 for the modified file.

What to include in your submission for 3.7:
Please observe whether H_1 and H_2 are similar or not. How many bits are the same between H_1 and H_2. Flip bits 1, 49, 73, and 113 and record the number of bits that are different between H_1 and H_2 when using SHA512 and SHA256 in a table. What trend do you see in table? Does this trend change if you flip different bits or flip multiple bits? What if you flipped two bits or all 4 bits at the same time?

openssl dgst -sha256 plain.txt SHA256(plain.txt)= 9ed7b218f3402376f0d2f77b89093704515f68f82857f17c9ba8dc4348ecf81f flip2 ~/cs370/program1/task3.7 1101$ openssl dgst -sha512 plain.txt SHA512(plain.txt)= 3f03cd845ee8f5405d64055c08be02ca1c386b153e8f10ef19fdca85ffe406eb1c038fa4b9220ad414 cc2cb88a1ef0e81b6c91339321d415808d2e0d0b7dd02d

When you compare H1 and H2, you notice that they are different. With the changing of 1 bit, it changes the rest of the code. As asked, by flipping bits 1,49, 73 and 113, you can see that roughly 50% of the code past that bit differs. This means that 1 change in bit result is 50% different in the following code. This trend ffers in different bit or multiple bits and if you flip two bit or all bits

3.8 [OPTIONAL] [120pts] Coding Task: Weak versus Strong Collision Resistance Property

In this task, we will investigate the difference between hash function's two properties: weak collision resistance property versus strong collision-resistance property. You will use the brute-force method to see how long it takes to break each of these properties. Instead of using openssl's command-line tools, you are required to write own programs to invoke the message digest functions in openssl's crypto library.

A sample C code can be found from http://www.openssl.org/docs/crypto/EVP_DigestInit.html. Please get familiar with this sample code.

Since most of the hash functions are quite strong against the brute-force attack on those two properties, it will take us years to break them using the brute-force method. To make the task feasible, we reduce the length of the hash value to 24 bits. We can use any one-way hash function, but we only use the first 24 bits of the hash value in this task. Namely, we are using a modified one-way hash function. Please design an experiment to find out the following:

1. [50pts] How many trials it will take you to break the weak collison resistsance property using the brute-force method? You should repeat your experiment for multiple times (100 or more depending on how long each trial takes), and report your average number of trials.

In order to break the weak collision resistance property using the brute-force method, it would take x methods.

2. [50pts] How many trials it will take you to break the collision-free property using the brute-force method? Similarly, you should report the average.

3. [10pts] Based on your observation, which property is easier to break using the brute-force method?

4. [10pts] Can you explain the difference in your observations?