

Programming Project 1 for Introduction to Computer Security

This project is based on Labs from the SEED project at Syracuse University funded by US National Science Foundation. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. A copy of the license can be found at <http://www.gnu.org/licenses/fdl.html>.

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 200 points.

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.

For Tasks 3.4 and 3.8, you will write your own code in the language of your choice and implement it. It is recommended you write it in Python but C is also acceptable. Be sure to include a README file on exactly how to compile and run your code.

3 Lab Environment and Tasks

Installing OpenSSL. In this lab assignment, 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 \
-K 00112233445566778889aabbccddeeff \
-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
- The ciphertext obtained using 3 different encryption modes/algorithms that you tried
 - o Include the names of the 3 cipher types

The plain text that I used is “This is the plaintext!”.

1. -aes-128-cbc => Salted__^PÁ<90>¾Ö5^Yó<4NX¥^SÇ^Y Z%WĐs&²-,^Bs/b; Ô,~âc^[
2. -aes-256-cbc => Salted__a Ã<^TëAÊ9«<8c>ÔAö<8d>ukÖKɹsòÔ;Nq<μ[Ýăú<82>^^ø^W9<9c>
3. -aes-256-cfb => Salted__Ñ^W<88>ý^B/ûÙ<9a>)d^GĚ<8e>’ă<8f>siJ-íâÑ”l̂<9f>Ă;<8d>;

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 information about the original picture from the encrypted picture? Record both encrypted pictures and the original picture for your report.

What to include in your submission for 3.2:

- Include the original picture



- Include the encrypted pictures with CBC mode



- Include the encrypted pictures with ECB mode



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 does not 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 3.2:

Now based on your observations of the above steps and the 3 different files that you generated, answer the following questions:

- Do the contents of 'encrypt1.bin' match the contents of 'encrypt2.bin'? Explain why or why not.
Yes the files do have the same value this is because when something is encrypted with the same key it will always yield the same result. If it did not always yield the same result when passing the same values it would be impossible to do thing like store passwords in an encrypted form
- Do the contents of 'encrypt1.bin' match the contents of 'encrypt3.bin'? Explain why or why not.
No, the files have different values because the value of vi was altered; it caused the function to evaluate differently and in turn yielded a completely different result. It is similar to changing the value of the public or private key.

3.4. [50pts] 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"`.

What to include in your submission for 3.5:

- Include the plaintext used
 - **The plaintext used here is "This is the message."**
- Include the hash digest obtained using 3 different hash algorithms that you tried
 - Name the 3 different hash algorithms you tried

```
C:\Users\jesse\OneDrive\Documents>openssl dgst -sha1 text.txt
```

```
SHA1(text.txt)= f18e2404e6e73648e2bb28dcff4efb312e325047
```

```
C:\Users\jesse\OneDrive\Documents>openssl dgst -sha256 text.txt
```

```
SHA2-256(text.txt)=
```

```
61bd9a82c57bd546ae776e30393510e21fdce54118d291943198049322afb37c
```

```
C:\Users\jesse\OneDrive\Documents>openssl dgst -sha512 text.txt
```

```
SHA2-512(text.txt)=
```

```
39c55ba08e03a6e9d0f1312245c69c55c450a17238bf04ac4118ab15ac5e051373d3e448
```

```
b1ef516de54468f3b6fcd8ed35431c3cfb8aa2602f7277b4338b0323
```

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?

We do not have to use a key with a fixed size. Although the output has a fixed sized depending on what algorithm is being used for instance sha 256 has a 256 bit output every time. This became very clear after numerous attempts to use the hash algorithm with various key sizes.

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
```

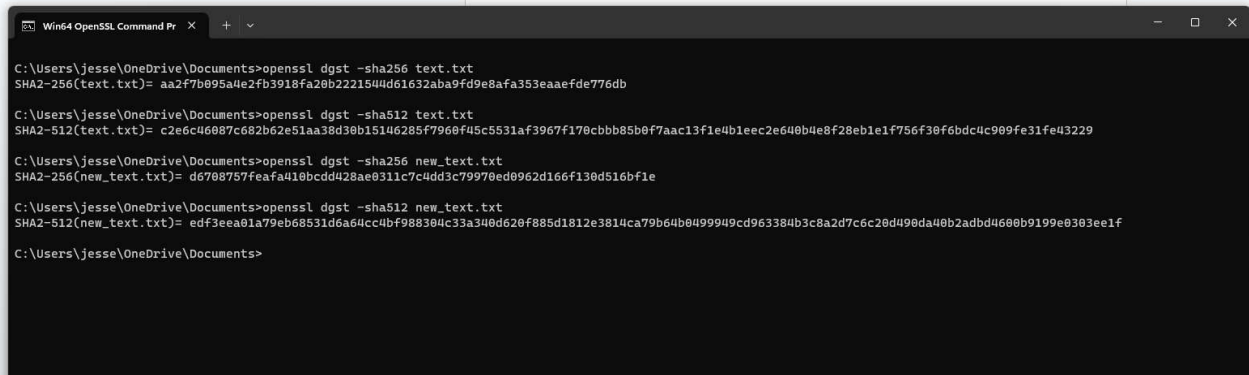

3.7. [10pts] Observation Task: The Randomness of One-way Hash

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?
The trend that seems to be happening is as more bits differ from the original texts, the more things seem to differ with the bits of the overall has values.
- Does this trend change if you flip different bits or flip multiple bits?
No the trend does not change if you flip multiple bits.
- What if you flipped two bits or all 4 bits at the same time?
The behavior is just different every time when you change a bit from a zero to a 1 or vice versa.



```
Win64 OpenSSL Command Pr x + -
C:\Users\jesse\OneDrive\Documents>openssl dgst -sha256 text.txt
SHA2-256(text.txt)= aa2f7b095a4e2fb3918fa20b2221544d61632aba9fd9e8afa353eaaefde776db

C:\Users\jesse\OneDrive\Documents>openssl dgst -sha512 text.txt
SHA2-512(text.txt)= c2e6c46087c682b62e51aa38d30b15146285f7960f45c5531af3967f170cbbb85b0f7aac13f1e4b1eec2e640b4e8f28eb1ef756f30f6bdc9c909fe31fe43229

C:\Users\jesse\OneDrive\Documents>openssl dgst -sha256 new_text.txt
SHA2-256(new_text.txt)= d6708757feafa410bcdd428ae0311c7c4dd3c79970ed0962d166f130d516bf1e

C:\Users\jesse\OneDrive\Documents>openssl dgst -sha512 new_text.txt
SHA2-512(new_text.txt)= edf3eea01a79eb68531d6a64cc4bf988304c33a340d620f885d1812e3814ca79b64b0499949cd963384b3c8a2d7c6c20d490da40b2adbd4600b9199e0303ee1f

C:\Users\jesse\OneDrive\Documents>
```


3.8 [90pts] 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:

- o [35pts] How many trials it will take you to break the weak collision resistance 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. **On average, it took 23 739 483 trials to break the weak collision property.**
- o [35pts] How many trials it will take you to break the collision-free property using the brute-force method? Similarly, you should report the average. **One average, it took 13 trials to break the strong collision resistance property.**
- o [10pts] Based on your observation, which property is easier to break using the brute-force method?

Based on the output, the strong collision property is much easier to break using the brute force method.

- o [10pts] Can you explain the difference in your observations?

There is a vast difference between the amount of trials it took to break each hash algorithm. The weak collision property is hard to break because we are looking for another input that produces the exact same hash. The strong collision property is much easier to break because of the birthday paradox. The chances of finding the same hash input increase tremendous as the amount of store hashes increases. To make it simpler, the weak collision is attempting to find a target that equals a specific hash. Whereas, the strong collision is looking for any two messages that are equal will suffice to break this property.