Crypto Lab – One-Way Hash Function and MAC

# Introduction:

This lab will allow you to get familiar with one-way hash functions and Message Authentication Codes (MAC) by using the **OpenSSL** software library. You will use this library directly from the Linux command line, but also from a C program that you will modify.

# Setting Up The Lab Environment

The lab is started from the Labtainer working directory on your Docker-enabled host, e.g., a Linux VM.  
From there, issue the command:

labtainer onewayhash

The resulting virtual terminals will include a display of a bash shell. The openssl package and other software described below are pre-installed on the system. However, you will have to install the development libraries for **OpenSSL**, more on that later..

# What is Open SSL:

OpenSSL is an open source software library for applications that require secure communications over computer networks. SSL stands for secure sockets layer which is a protocol that was originally developed by Netscape in the late 1990’s. It is a set of rules that governing server and client authentication as well as encrypted communication. SSL is now synonymous with another acronym Transport Layer Security (TLS) as TLS builds upon the now deprecated original SSL specifications. While still being called OpenSSL, since 2018 (version.1.1.1 of open SSL) the library now supports both legacy SSL functionality and TLS 1.3.

OpenSSL is a library of compiled binaries that can be accessed from an operating system such as Linux, MacOS and Linux on the command line. However, by installing both Openssl and LibSSL on Linux one may import the necessary header files into a C or C++ program and utilize open SSL functionality.

# Installing OpenSSL and LibSSL in Linux

Open a Bash terminal on your virtual machine. Your virtual Linux machine list of packages may be a little out of date, so the first thing you must do is update the Advanced Package Tool’s (apt) list of packages. You can do this with the following command:

**sudo apt-get update**

Note: Sudo is a command that provides super user privileges (i.e. do as super user). The super user password for your virtual machine should be **password123**. This is one reason we use a virtual machine, as you could not do this on Agora. Apt should take a couple of minutes to fetch and reference all the most recent packages available to it.

Next, we should make sure the most recent version of **OpenSSL** is installed. Note the docker container of Labtainers is running an older version od Debian 9 and for it the most recent version ofOpenSSL is 1.02g, when never versions of Debian will have 1.11. First check that **OpenSSL**  is installed and functioning with the following linux command:

**openssl version**

To install or update **OpenSSL**  enter the following command:

**Sudo apt-get install openssl-devel**

Next, we shall install the SSL code libraries so we may utilize open SSL from C as these are not installed. Use the following command:

**sudo apt-get install libssl-dev**

# TASK 1: Generating Message Digest and MAC:

In this task, we will play with various one-way hash algorithms. You can use the following openssl dgst command to generate the hash value for a file. To see the manpages, 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 -md5, -sha1, -sha256, etc. And replace [filename] with filetodigest.txt, which is in your home directory (of the container). In this task, you should try at least 3 different algorithms, and describe your observations. You can find the supported one-way hash algorithms by typing "openssl dgst -h" in older versions of OpenSSL (< 1.1.0) or openssl list -digest-algorithms in never versions of OpenSsl (>= 1.1.0). NOTE: the list of algorithms included in the manpages might not be correct.

**Task Results:** Below, type the three Linux Commands used and the corresponding output. Use a color other than black:

ubuntu@onewayhash:~$ openssl dgst -sha1 filetodigest.txt

SHA1(filetodigest.txt)= 3a34eb74a42e99633349ed09f239f46a4d728515

ubuntu@onewayhash:~$ ls

filetodigest.txt

ubuntu@onewayhash:~$ openssl dgst -sha256 filetodigest.txt

SHA256(filetodigest.txt)= cd293847bdf14311c10f1cf3b2932cb375f93508b7d2a1596c8070ba1fe02295

ubuntu@onewayhash:~$ openssl dgst -md5 filetodigest.txt

MD5(filetodigest.txt)= d6818d10168775cfe323e044d97c04ca1.

# TASK 2: Keyed Hash

In this task, we would like to generate a keyed hash (i.e. MAC) for a file. We can use the -hmac option (this option is currently undocumented, but it is supported by openssl). The following example generates a keyed hash for a file using the HMAC with the-MD5 algorithm. The string following the -hmac option is the key.

**openssl dgst -md5 -hmac "abcdefg" [filename]**

Please generate a keyed hash using HMAC-MD5, HMAC-SHA256, and HMAC-SHA1 for any file that you choose. Please try several keys with different length.

**Question:** Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why? Type your response below in a color other than black.

**Type your response below:**

No, we do not have to use a key with a fixed size for HMAC, no matter the key size it will always produce a fixed-length output for any size. If the key size is larger than the function will be hashed down to a smaller size and be that fixed length, however if the key size is larger than the function will pad it to be that fixed length.

# TASK 3: The Randomness of Hash

To understand the properties of one-way hash functions, we would like to do the following exercise for MD5 and SHA256.

* Create a text file named ”edit-this-file.txt” of any length.  
  Generate the hash value ***H1*** for this file using a specific hash algorithm.
* Flip one bit of the input file. You can achieve this modification using **hexedit**.  
  Generate the hash value ***H2*** for the modified file.

Use **man hexedit** to learn how to use hexedit.

* Please observe whether ***H1*** and ***H2*** are similar or not. Please describe your observations below. You can write a short program to count how many bits are the same between H1 and H2.

**Write/Type Observations Below:**

Initial Digest: ubuntu@onewayhash:~$ openssl dgst -sha1 -hmac "abcdef" edit-this-file.txt

HMAC-SHA1(edit-this-file.txt)= 5f7b946ec5e5378a32c56115bc9114978009a751

Result Digest: ubuntu@onewayhash:~$ openssl dgst -sha1 -hmac "abcdef" edit-this-file.txt

HMAC-SHA1(edit-this-file.txt)= 45d2e84fc294caf528e14493739ece2433f513f6

The entire result of the digest changed from a single modification of a bit in the file.

# Task 4:

Take a look at the following C language program code. It is based on a sample program in the documentation of OpenSSL’s EVP digest routines found here: <http://www.openssl.org/docs/crypto/EVP_DigestInit.html>.

This example digests the data "Test Message\n" and "Hello World\n", using the digest name passed on the command line. Take a look at the documentation and use it to understand how the EVP digest routines work in this context.

To make it work we must first install the open SSL development libraries which it seems were not included with the Labtainer instance. To do this first we must update the list of sources. The sudo (do as super user) commands in the Labtainer instances do not require passwords

**sudo apt-get update**

Then we can instsall the development libraries for open SSL and update to the most recent OpenSSL version to match the development libraries.

**sudo apt-get install libssl-dev**

**sudo apt-get install openssl**

To compile the program, we must make sure that we reference the cryptography libraries of OpenSSL:

**gcc mdtest.c -lcrypto -o mdtest**

Once compiled, you can run it with the following commands which uses the md5 digest algorithm.

**mdtest -md5**

#include <stdio.h>

#include <string.h>

#include <openssl/evp.h>

int main**(**int argc**,** char **\***argv**[]){**

//uncomment the line below if running the OpenSSL version 1.02 or less

//OpenSSL\_add\_all\_algorithms**();**

EVP\_MD\_CTX **\***mdctx**;**

const EVP\_MD **\***md**;**

char mess1**[]** **=** "Test Message\n"**;**

char mess2**[]** **=** "Hello World\n"**;**

unsigned char md\_value**[**EVP\_MAX\_MD\_SIZE**];**

unsigned int md\_len**,**i**;**

**if** **(**argv**[**1**]** **==** **NULL){**

printf**(**"usage: mdtest digestname\n"**);**

exit**(**1**);**

**}**

md **=** EVP\_get\_digestbyname**(**argv**[**1**]);**

**if** **(**md **==** **NULL){**

printf**(**"Unknown message diagest %s\n"**,**argv**[**1**]);**

exit**(**1**);**

**}**

mdctx **=** EVP\_MD\_CTX\_create**(\_);**

EVP\_DigestInit\_ex**(**mdctx**,**md**,** **NULL);**

EVP\_DigestUpdate**(**mdctx**,** mess1**,** strlen**(**mess1**)** **);**

EVP\_DigestUpdate**(**mdctx**,** mess2**,** strlen**(**mess2**)** **);**

EVP\_DigestFinal\_ex**(**mdctx**,** md\_value**,&**md\_len**);**

EVP\_MD\_CTX\_destroy**(**mdctx**);**

printf**(**"Digest is: "**);**

**for(**i **=**0**;**i **<** md\_len**;** i**++)**

printf**(**"%02x"**,**md\_value**[**i**]);**

printf**(**"\n"**);**

exit**(**0**);**

**}**

In your virtual machine’s docker container, compile and run the above program. Try with md5, sha1 and sha256 digest algorithms. **Type the commands and corresponding outputs below**:

# Task 5: Modify the C Program and recompile it.

Copy the program made in the last section renaming the copy to HashString.C

Modify the C program from the last task so that when run on the command line it receives two parameters. The first being the name of the hashing algorithm, the second being the string to hash. If the number of parameters is incorrect, print an error informing the user how to use the command.

Compile and test your solution.

You will submit this code when finished with the worksheet.

Tip:

\*\*Note\*\* Python 3 stores strings as Unicode represented as bytes (8 bits) per character. Sometimes the **Python String method encode() is needed to** convert. a string value into a collection of bytes. For example, when passing a string to hashlib’s update method a string will need encoding to utf-8 which is what the methods of the hashlib library support:

string = “No one expects the Spanish inquisition!”.

encoded\_string = string.encode(“utf-8”);

# Task 6 Do it in Python

Python also can utilize the same hashing algorithms available to OpenSSL via its PyCryptodome library. To learn more about this library view their documentation below:

<https://www.pycryptodome.org/>

Rewrite the program from task 5 for Python using the above library. You do not have to do this in our Labtainer virtual machine. You should notice how it is considerably smaller. Make sure you are able to run and use in on the command line. You do not have to do this on the virtual machine.

Using the same algorithm and string as inputs. Compare the output of both the C program running on the guest virtual machine and the and the Python program running on the host.

If you have got it right, you should notice how the of the hex code outputs are the same regardless of operating system and programming language. This demonstrates why hashing algorithms are so effective and dependable.

You will submit this code when finished with this worksheet.

# Task 7: Collisions and One Way Properties

In the domain of hashing algorithms, a collision is where two or more different messages result in the same message digest.

hash(message1) == hash(message2)

The collision free property defines the extent to which different messages may generate the same digest. To break the collision-free property you need to find two or more texts that generate the same hash.

The one way property means that when we use a hash algorithm by passing it a message we cannot then feasibly use the digest to work back and discover the string. To break the one-way property you are given a hash and you search for another text with the same hash.

hash(message) = digest

With a robust hashing algorithm, it is unfeasibly difficult to compute the original message from the digest and therefore the hashing algorithm is one-way. By unfeasible we mean it would take years even for a distributed attack.

In this task we will look at both the collision-free and one way properties of hashing algorithms. 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. Remember that the hashes are shown in hexadecimal, so each character represents 4 bits (i.e 0000 = 0 to 1111 = F), so 24 bits would be the first 6 characters.

In Python, design an experiment to find out the following. Write your responses below:

1. How many trials it will take you to break the one-way property using the brute-force method? You should repeat your experiment for multiple times, and report your average number of trials.

Takes on average about 1 million trials to brute force the first 3 bytes of the one-way property. This was pulled from about 130 datapoints.

1. How many trials it will take you to break the collision-free property using the brute-force method?

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

4. (10 Bonus Points) Can you explain the difference in your observation mathematically?

# 4 Submission

When the lab is finished, provide a zip file containing :

* this worksheet with your answers typed out in a color other than black.
* The C code you developed for task 5.
* The Python code you developed for task 6.
* The Python code you developed for task 7.

Submit your work as a gzipped tar file via send lab on Agora. To compress a directory into a gzip file, assuming you have all your files in a folder called lab2 and you are in the folder above this:

**tar -cvzf lab3.tar.gz lab3**

Then use the following command to submit the file on Agora (this will only work on Agora):

**sendlab.325.1 3 lab3.tar.gz**