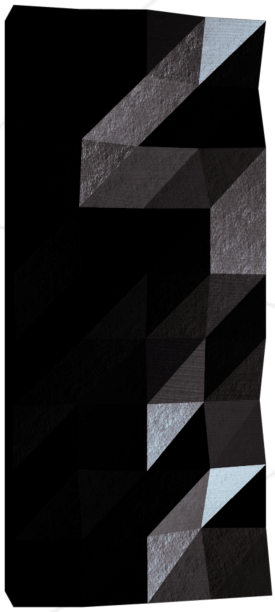

ft_ssl [md5] [sha256]

An introduction to cryptographic hashing algorithms

Summary: This project is the gateway to the encryption branch. You will recode part of the OpenSSL program, specifically the MD5 and SHA-256 hashing algorithms.

Version: 4.1

Contents

I Introduction **2**

II Objectives **3**

III General Instructions **4**

IV Mandatory part **5**

IV.0.1 MD5 6

IV.0.2 SHA 256 8

V Bonus part **9**

VI Submission and peer-evaluation **10**

Chapter I

Introduction

This is what Wikipedia has to say on the subject of cryptographic hash functions:

A cryptographic hash function is a special class of hash function that has certain propertiesi which make it suitable for use in cryptography. It is a mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size. It is designed to also be a `one-way function`, that is, a function which is infeasible to invert. The only way to recreate the input data from an ideal cryptographic hash function's output is to attempt a brute-force search of possible inputs to see if they produce a match, or use a rainbow table of matched hashes. Bruce Schneier has called one-way hash functions "the workhorses of modern cryptography". The input data is often called the message, and the output (the hash value or hash) is often called the message digest or simply the digest.

The ideal cryptographic hash function has five main properties:

- it is deterministic, so the same message always results in the same hash
- it is quick to compute the hash value for any given message
- it is infeasible to generate a message from its digest except by trying all possible messages
- a small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value
- it is infeasible to find two different messages with the same hash value

Cryptographic hash functions have many information-security applications, notably in `digital signatures`, message authentication codes (MACs), and other forms of authentication. They can also be used as ordinary hash functions, to index data in hash tables, for fingerprinting, to detect duplicate data or uniquely identify files, and as `checksums` to detect accidental data corruption. Indeed, in information-security contexts, cryptographic hash values are sometimes called (digital) fingerprints, checksums, or just hash values, even though all these terms stand for more general functions with rather different properties and purposes.

Chapter II

Objectives

This is the first `ft_ssl` project on the path of `Encryption and Security`. You will recode some security technologies you may have already been using, from scratch.

You will want to plan out the structure of your executable before you begin, because you will build directly onto it in later security projects. It is of vital importance that your functions are modular, your code is easy to re-use and adding onto the executable doesn't require rewriting the program.

`Re-read that last sentence again. I'm serious, do it.`

This project will focus specifically on cryptographic hashing algorithms to solidify your understanding of bitwise operations, integer overflow, and one-way functions. A subplot to this project is to emphasize writing clean code that can be efficiently extended.

`If you have ever downloaded a file of significance from the internet (an operating system installation image, for example), you have probably noticed a section with either SHA-1 or MD5 followed by gibberish. I advise you to do some research to learn more about digital fingerprinting so you can better protect yourself from malicious downloads.`

Chapter III

General Instructions

- This project will only be corrected by other human beings. You are therefore free to organize and name your files as you wish, although you need to respect some requirements below.
- The executable file must be named `ft_ssl`.
- The implementation of ft_ssl and all its functionalities must be done in C.
- You must submit a Makefile. The Makefile must contain the usual rules and compile the project as necessary.
- You have to handle errors carefully. In no way can your program quit unexpectedly (Segfault, bus error, double free, etc). If you are unsure, handle the errors like `OpenSSL`.
- You are allowed the following functions:
  - `open`
  - `close`
  - `read`
  - `write`
  - `malloc`
  - `free`
- You are allowed to use other functions as long as their use is justified. (Although they shouldn't be necessary, if you find that you need a `strerror` or an `exit`, that's okay, but using a function or library that does your work is not.)
- To do this project, you will have to use a linux OS, you shall use docker or a VM if needed.

Chapter IV

Mandatory part

`OpenSSL` is a cryptographic toolkit library written in C that is used to secure communications over computer networks. It implements the `Secure Sockets Layer (SSL)` and `Transport Layer Security (TLS)` network protocols to protect against eavesdropping and verify identities during network communication.
You must create a program named `ft_ssl` in C that will recreate part of the `OpenSSL` functionality.

`As you're no longer a beginner, we won't accept a forest of if/else (think of the function pointer array for the dispatching part). You need to make your code clear. Your code will be reviewed.`

For this project, you will implement the `md5` and `sha256` hashing functions.

```
 > ./ft_ssl
usage: ft_ssl command [flags] [file/string]
 >
 > ./ft_ssl foobar
ft_ssl: Error: 'foobar' is an invalid command.

Commands:
md5
sha256

Flags:
-p -q -r -s
```

`You will add Standard and Cipher commands later.`
`You will need to be able to input and output on STDIN/STDOUT and also files as necessary for the commands, but not for the ft_ssl itself.`

IV.0.1 MD5

`MD5` is a cryptographic hash function. It is short for "Message Digest 5".

You must create the `md5` command for your `ft_ssl` program. You must include the following flags:

- `-p`, echo `STDIN` to `STDOUT` and append the checksum to `STDOUT`
- `-q`, quiet mode
- `-r`, reverse the format of the output.
- `-s`, print the sum of the given string

Emulate the behavior of the `md5sum` executable.

`The flags requested here are customised and may work differently to the real program, which is normal; it should be added that these flags do not necessarily have to exist in md5sum.`

If flags are not provided, be prepared to read/write from/to the console.

`Example usage is on the next page.`

```
 > echo "42 is nice" | openssl md5
 (stdin)= 35f1d6de0302e2086a4e472266efb3a9
 >
 >  echo "42 is nice" | md5sum
 35f1d6de0302e2086a4e472266efb3a9 -
 >
 > echo "42 is nice" | ./ft_ssl md5
 (stdin)= 35f1d6de0302e2086a4e472266efb3a9
 >
 > echo "42 is nice" | ./ft_ssl md5 -p
 ("42 is nice")= 35f1d6de0302e2086a4e472266efb3a9
 >
 > echo "Pity the living." | ./ft_ssl md5 -q -r
e20c3b973f63482a778f3fd1869b7f25
 >
 > echo "And above all," > file
 > ./ft_ssl md5 file
MD5 (file) = 53d53ea94217b259c11a5a2d104ec58a
 > ./ft_ssl md5 -r file
53d53ea94217b259c11a5a2d104ec58a file
 >
 > ./ft_ssl md5 -s "pity those that aren't following baerista on spotify."
MD5 ("pity those that aren't following baerista on spotify.") = a3c990a1964705d9bf0e602f44572f5f
 >
 > echo "be sure to handle edge cases carefully" | ./ft_ssl md5 -p file
("be sure to handle edge cases carefully")= 3553dc7dc5963b583c056d1b9fa3349c
MD5 (file) = 53d53ea94217b259c11a5a2d104ec58a
 >
 > echo "some of this will not make sense at first" | ./ft_ssl md5 file
MD5 (file) = 53d53ea94217b259c11a5a2d104ec58a
 >
 > echo "but eventually you will understand" | ./ft_ssl md5 -p -r file
("but eventually you will understand")= dcdd84e0f635694d2a943fa8d3905281
53d53ea94217b259c11a5a2d104ec58a file
 >
 > echo "GL HF let's go" | ./ft_ssl md5 -p -s "foo" file
("GL HF let's go")= d1e3cc342b6da09480b27ec57ff243e2
MD5 ("foo") = acbd18db4cc2f85cedef654fccc4a4d8
MD5 (file) = 53d53ea94217b259c11a5a2d104ec58a
 >
 > echo "one more thing" | ./ft_ssl md5 -r -p -s "foo" file -s "bar"
("one more thing")= a0bd1876c6f011dd50fae52827f445f5
acbd18db4cc2f85cedef654fccc4a4d8 "foo"
53d53ea94217b259c11a5a2d104ec58a file
ft_ssl: md5: -s: No such file or directory
ft_ssl: md5: bar: No such file or directory
 >
 > echo "just to be extra clear" | ./ft_ssl md5 -r -q -p -s "foo" file
just to be extra clear
3ba35f1ea0d170cb3b9a752e3360286c
acbd18db4cc2f85cedef654fccc4a4d8
53d53ea94217b259c11a5a2d104ec58a
```

The error message syntax does not have to match perfectly so long as the error message itself is sensible.

IV.0.2 SHA 256

`SHA-256` is one of the functions in the `SHA` (Secure Hash Algorithms) family. The `SHA` algorithms were designed by the NSA.

You must create the `sha256` command for your `ft_ssl` program. You must include the following flags:

- `-p`, echo `STDIN` to `STDOUT` and append the checksum to `STDOUT`
- `-q`, quiet mode
- `-r`, reverse the format of the output.
- `-s`, print the sum of the given string

`The flags requested here are custom and may function differently compared to the real program. This is normal.`

The behavior should be exactly the same as the command above, with the only difference being the hash algorithm in question.

```
 > echo "https://www.42.fr/" > website
 > ./ft_ssl sha256 -q website
1ceb55d2845d9dd98557b50488db12bbf51aaca5aa9c1199eb795607a2457daf
 > sha256sum website
1ceb55d2845d9dd98557b50488db12bbf51aaca5aa9c1199eb795607a2457daf website
 >
 > ./ft_ssl sha256 -s "42 is nice"
SHA256 ("42 is nice") = b7e44c7a40c5f80139f0a50f3650fb2bd8d00b0d24667c4c2ca32c88e13b758f
> echo -n "42 is nice" | sha256sum
b7e44c7a40c5f80139f0a50f3650fb2bd8d00b0d24667c4c2ca32c88e13b758f -
```

Chapter V

Bonus part

There are several bonuses that can be earned for this project.

- For your first bonus your program must be able to parse commands from `STDIN` the same way that `OpenSSL` does.
- The second and final bonus for this project is simply to implement a more advanced hash algorithm than a simple `MD5`.

You need to implement whirlpool in order to get maximum points.

A hash function that is weaker than MD5 will not grant a bonus.

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Chapter VI

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double-check the names of your folders and files to ensure they are correct.