# **Getting started with OpenSSL: Cryptography basics**

Need a primer on cryptography basics, especially regarding OpenSSL? Read on.

This article is the first of two on cryptography basics using [OpenSSL](https://www.openssl.org/), a production-grade library and toolkit popular on Linux and other systems. (To install the most recent version of OpenSSL, see [here](https://www.howtoforge.com/tutorial/how-to-install-openssl-from-source-on-linux/).) OpenSSL utilities are available at the command line, and programs can call functions from the OpenSSL libraries. The sample program for this article is in C, the source language for the OpenSSL libraries.

The two articles in this series cover—collectively—cryptographic hashes, digital signatures, encryption and decryption, and digital certificates. You can find the code and command-line examples in a ZIP file from [my website](http://condor.depaul.edu/mkalin).

Let’s start with a review of the SSL in the OpenSSL name.

## 1) A quick history

[Secure Socket Layer (SSL)](https://en.wikipedia.org/wiki/Transport_Layer_Security) is a cryptographic protocol that [Netscape](https://en.wikipedia.org/wiki/Netscape) released in 1995. This protocol layer can sit atop HTTP, thereby providing the S for secure in HTTPS. The SSL protocol provides various security services, including two that are central in HTTPS:

* **Peer authentication (aka mutual challenge):** Each side of a connection authenticates the identity of the other side. If Alice and Bob are to exchange messages over SSL, then each first authenticates the identity of the other.
* **Confidentiality**: A sender encrypts messages before sending these over a channel. The receiver then decrypts each received message. This process safeguards network conversations. Even if eavesdropper Eve intercepts an encrypted message from Alice to Bob (a man-in-the-middle attack), Eve finds it computationally infeasible to decrypt this message.

These two key SSL services, in turn, are tied to others that get less attention. For example, SSL supports message integrity, which assures that a received message is the same as the one sent. This feature is implemented with hash functions, which likewise come with the OpenSSL toolkit.

SSL is versioned (e.g., SSLv2 and SSLv3), and in 1999 Transport Layer Security (TLS) emerged as a similar protocol based upon SSLv3. TLSv1 and SSLv3 are alike, but not enough so to work together. Nonetheless, it is common to refer to SSL/TLS as if they are one and the same protocol. For example, OpenSSL functions often have SSL in the name even when TLS rather than SSL is in play. Furthermore, calling OpenSSL command-line utilities begins with the term **openssl**.

The documentation for OpenSSL is spotty beyond the **man** pages, which become unwieldy given how big the OpenSSL toolkit is. Command-line and code examples are one way to bring the main topics into focus together. Let’s start with a familiar example—accessing a web site with HTTPS—and use this example to pick apart the cryptographic pieces of interest.

## 2) An HTTPS client

The **client** program shown here connects over HTTPS to Google:

*/\* compilation: gcc -o client client.c -lssl -lcrypto \*/*  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <openssl/bio.h> /\* BasicInput/Output streams \*/  
#include <openssl/err.h> /\* errors \*/  
#include <openssl/ssl.h> /\* core library \*/  
  
#define BuffSize 1024  
  
void report\_and\_exit(const char\* msg) {  
  [perror](http://www.opengroup.org/onlinepubs/009695399/functions/perror.html)(msg);  
  ERR\_print\_errors\_fp(stderr);  
  [exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(-1);  
}  
  
void init\_ssl() {  
  SSL\_load\_error\_strings();  
  SSL\_library\_init();  
}  
  
void cleanup(SSL\_CTX\* ctx, BIO\* bio) {  
  SSL\_CTX\_free(ctx);  
  BIO\_free\_all(bio);  
}  
  
void secure\_connect(const char\* hostname) {  
  char name[BuffSize];  
  char request[BuffSize];  
  char response[BuffSize];  
  
  const SSL\_METHOD\* method = TLSv1\_2\_client\_method();  
  if (NULL == method) report\_and\_exit("TLSv1\_2\_client\_method...");  
  
  SSL\_CTX\* ctx = SSL\_CTX\_new(method);  
  if (NULL == ctx) report\_and\_exit("SSL\_CTX\_new...");  
  
  BIO\* bio = BIO\_new\_ssl\_connect(ctx);  
  if (NULL == bio) report\_and\_exit("BIO\_new\_ssl\_connect...");  
  
  SSL\* ssl = NULL;  
  
  */\* link bio channel, SSL session, and server endpoint \*/*  
  
  [sprintf](http://www.opengroup.org/onlinepubs/009695399/functions/sprintf.html)(name, "%s:%s", hostname, "https");  
  BIO\_get\_ssl(bio, &ssl); */\* session \*/*  
  SSL\_set\_mode(ssl, SSL\_MODE\_AUTO\_RETRY); */\* robustness \*/*  
  BIO\_set\_conn\_hostname(bio, name); */\* prepare to connect \*/*  
  
  */\* try to connect \*/*  
  if (BIO\_do\_connect(bio) <= 0) {  
    cleanup(ctx, bio);  
    report\_and\_exit("BIO\_do\_connect...");  
  }  
  
  */\* verify truststore, check cert \*/*  
  if (!SSL\_CTX\_load\_verify\_locations(ctx,  
                                      "/etc/ssl/certs/ca-certificates.crt", */\* truststore \*/*  
                                      "/etc/ssl/certs/")) */\* more truststore \*/*  
    report\_and\_exit("SSL\_CTX\_load\_verify\_locations...");  
  
  long verify\_flag = SSL\_get\_verify\_result(ssl);  
  if (verify\_flag != X509\_V\_OK)  
    [fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(stderr,  
            "##### Certificate verification error (%i) but continuing...\n",  
            (int) verify\_flag);  
  
  */\* now fetch the homepage as sample data \*/*  
  [sprintf](http://www.opengroup.org/onlinepubs/009695399/functions/sprintf.html)(request,  
          "GET / HTTP/1.1\x0D\x0AHost: %s\x0D\x0A\x43onnection: Close\x0D\x0A\x0D\x0A",  
          hostname);  
  BIO\_puts(bio, request);  
  
  */\* read HTTP response from server and print to stdout \*/*  
  while (1) {  
    [memset](http://www.opengroup.org/onlinepubs/009695399/functions/memset.html)(response, '\0', sizeof(response));  
    int n = BIO\_read(bio, response, BuffSize);  
    if (n <= 0) break; */\* 0 is end-of-stream, < 0 is an error \*/*  
   [puts](http://www.opengroup.org/onlinepubs/009695399/functions/puts.html)(response);  
  }  
  
  cleanup(ctx, bio);  
}  
  
int main() {  
  init\_ssl();  
  
  const char\* hostname = "www.google.com:443";  
  [fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(stderr, "Trying an HTTPS connection to %s...\n", hostname);  
  secure\_connect(hostname);  
  
return 0;  
}

This program can be compiled and executed from the command line (note the lowercase L in **-lssl** and **-lcrypto**):

**gcc -o client client.c -lssl -lcrypto**

This program tries to open a secure connection to the web site [www.google.com](http://www.google.com/). As part of the TLS handshake with the Google web server, the **client** program receives one or more digital certificates, which the program tries (but, on my system, fails) to verify. Nonetheless, the **client** program goes on to fetch the Google homepage through the secure channel. This program depends on the security artifacts mentioned earlier, although only a digital certificate stands out in the code. The other artifacts remain behind the scenes and are clarified later in detail.

Generally, a client program in C or C++ that opened an HTTP (non-secure) channel would use constructs such as a file descriptor for a network socket, which is an endpoint in a connection between two processes (e.g., the client program and the Google web server). A file descriptor, in turn, is a non-negative integer value that identifies, within a program, any file-like construct that the program opens. Such a program also would use a structure to specify details about the web server’s address.

None of these relatively low-level constructs occurs in the client program, as the OpenSSL library wraps the socket infrastructure and address specification in high-level security constructs. The result is a straightforward API. Here’s a first look at the security details in the example **client** program.

* The program begins by loading the relevant OpenSSL libraries, with my function **init\_ssl** making two calls into OpenSSL:

**SSL\_library\_init();**

**SSL\_load\_error\_strings();**

* The next initialization step tries to get a security **context**, a framework of information required to establish and maintain a secure channel to the web server. **TLS 1.2** is used in the example, as shown in this call to an OpenSSL library function:

**const SSL\_METHOD\* method = TLSv1\_2\_client\_method(); /\* TLS 1.2 \*/**

If the call succeeds, then the **method** pointer is passed to the library function that creates the context of type **SSL\_CTX**:

**SSL\_CTX\* ctx = SSL\_CTX\_new(method);**

The **client** program checks for errors on each of these critical library calls, and then the program terminates if either call fails.

* Two other OpenSSL artifacts now come into play: a security session of type **SSL**, which manages the secure connection from start to finish; and a secured stream of type **BIO** (Basic Input/Output), which is used to communicate with the web server. The **BIO** stream is generated with this call:

**BIO\* bio = BIO\_new\_ssl\_connect(ctx);**

Note that the all-important context is the argument. The **BIO** type is the OpenSSL wrapper for the **FILE** type in C. This wrapper secures the input and output streams between the **client** program and Google's web server.

* With the **SSL\_CTX** and **BIO** in hand, the program then links these together in an **SSL** session. Three library calls do the work:

**BIO\_get\_ssl(bio, &ssl); /\* get a TLS session \*/**

**SSL\_set\_mode(ssl, SSL\_MODE\_AUTO\_RETRY); /\* for robustness \*/**

**BIO\_set\_conn\_hostname(bio, name); /\* prepare to connect to Google \*/**

The secure connection itself is established through this call:

**BIO\_do\_connect(bio);**

If this last call does not succeed, the **client** program terminates; otherwise, the connection is ready to support a confidential conversation between the **client** program and the Google web server.

During the handshake with the web server, the **client** program receives one or more **digital certificates** that authenticate the server’s identity. However, the **client** program does not send a certificate of its own, which means that the authentication is one-way. (Web servers typically are configured not to expect a client certificate.) Despite the failed verification of the web server’s certificate, the **client** program continues by fetching the Google homepage through the secure channel to the web server.

Why does the attempt to verify a Google certificate fail? A typical OpenSSL installation has the directory **/etc/ssl/certs**, which includes the **ca-certificates.crt** file. The directory and the file together contain digital certificates that OpenSSL trusts out of the box and accordingly constitute a truststore. The truststore can be updated as needed, in particular, to include newly trusted certificates and to remove ones no longer trusted.

The client program receives three certificates from the Google web server, but the OpenSSL truststore on my machine does not contain exact matches. As presently written, the **client** program does not pursue the matter by, for example, verifying the digital signature on a Google certificate (a signature that vouches for the certificate). If that signature were trusted, then the certificate containing it should be trusted as well. Nonetheless, the client program goes on to fetch and then to print Google’s homepage. The next section gets into more detail.

## 3) The hidden security pieces in the client program

Let’s start with the visible security artifact in the client example—***the digital certificate***—and consider how other security artifacts relate to it. The dominant layout standard for a digital certificate is X509, and a production-grade certificate is issued by a certificate authority (CA) such as [Verisign](https://www.verisign.com/).

A digital certificate contains various pieces of information (e.g., activation and expiration dates, and a domain name for the owner), including the issuer’s identity and **digital signature,** which is an encrypted cryptographic hash value. A certificate also has an unencrypted hash value that serves as its identifying fingerprint.

A hash value results from mapping an arbitrary number of bits to a fixed-length digest. What the bits represent (an accounting report, a novel, or maybe a digital movie) is irrelevant. For example, the Message Digest version 5 (MD5) hash algorithm maps input bits of whatever length to a 128-bit hash value, whereas the SHA1 (Secure Hash Algorithm version 1) algorithm maps input bits to a 160-bit value. Different input bits result in different—indeed, statistically unique—hash values. The next article goes into further detail and focuses on what makes a hash function cryptographic.

Digital certificates differ in type (e.g., root, intermediate, and end-entity certificates) and form a hierarchy that reflects these types. As the name suggests, a **root certificate** sits atop the hierarchy, and the certificates under it inherit whatever trust the root certificate has. The OpenSSL libraries and most modern programming languages have an X509 type together with functions that deal with such certificates. The certificate from Google has an X509 format, and the **client** program checks whether this certificate is **X509\_V\_OK**.

X509 certificates are based upon public-key infrastructure (PKI), which includes algorithms—RSA is the dominant one—for generating **key pairs**: **a public key and its paired private key**. A public key is an identity: [Amazon’s](https://www.amazon.com/) public key identifies it, and my public key identifies me. A private key is meant to be kept secret by its owner.

**The keys in a pair have standard uses**. *A public key* can be used to encrypt a message, and the *private key* from the same pair can then be used to decrypt the message. *A private key* also can be used to sign a document or other electronic artifact (e.g., a program or an email), and the *public key* from the pair can then be used to verify the signature. The following two examples fill in some details.

In the first example, Alice distributes her public key to the world, including Bob. Bob then encrypts a message with Alice’s public key, sending the encrypted message to Alice. The message encrypted with Alice’s public key is decrypted with her private key, which (by assumption) she alone has, like so:

**+------------------+ encrypted msg  +-------------------+  
Bob's msg--->|Alice's public key|--------------->|Alice's private key|---> Bob's msg  
             +------------------+                +-------------------+**

Decrypting the message without Alice’s private key is possible in principle, but infeasible in practice given a sound cryptographic key-pair system such as RSA.

Now, for the second example, consider signing a document to certify its authenticity. The signature algorithm uses a private key from a pair to process a cryptographic hash of the document to be signed:

**+-------------------+  
Hash of document--->|Alice's private key|--->Alice's digital signature of the document  
                    +-------------------+**

Assume that Alice digitally signs a contract sent to Bob. Bob then can use Alice’s public key from the key pair to verify the signature:

**+------------------+  
Alice's digital signature of the document--->|Alice's public key|--->verified or not  
                                             +------------------+**

It is infeasible to forge Alice’s signature without Alice’s private key: hence, it is in Alice’s interest to keep her private key secret.

None of these security pieces, except for digital certificates, is explicit in the **client** program. The next article fills in the details with examples that use the OpenSSL utilities and library functions.

## 4) OpenSSL from the command line

In the meantime, let’s take a look at OpenSSL command-line utilities: in particular, a utility to inspect the certificates from a web server during the TLS handshake. Invoking the OpenSSL utilities begins with the **openssl** command and then adds a combination of arguments and flags to specify the desired operation.

Consider this command:

**openssl list -cipher-algorithms**

The output is a list of associated algorithms that make up a **cipher suite**. Here’s the start of the list, with comments to clarify the acronyms:

**AES-128-CBC ## Advanced Encryption Standard, Cipher Block Chaining  
AES-128-CBC-HMAC-SHA1 ## Hash-based Message Authentication Code with SHA1 hashes  
AES-128-CBC-HMAC-SHA256 ## ditto, but SHA256 rather than SHA1  
...**

The next command, using the argument **s\_client**, opens a secure connection to [**www.google.com**](http://www.google.com/) and prints screens full of information about this connection:

openssl s\_client -connect [www.google.com:443](http://www.google.com:443/) -showcerts

The port number 443 is the standard one used by web servers for receiving HTTPS rather than HTTP connections. (For HTTP, the standard port is 80.) The network address [**www.google.com:443**](http://www.google.com:443/) also occurs in the **client** program's code. If the attempted connection succeeds, **the three digital certificates** from Google are displayed together with information about the secure session, the cipher suite in play, and related items. For example, here is a slice of output from near the start, which announces that a certificate chain is forthcoming. The encoding for the certificates is base64:

**Certificate chain  
 0 s:/C=US/ST=California/L=Mountain View/O=Google LLC/CN=www.google.com  
 i:/C=US/O=Google Trust Services/CN=Google Internet Authority G3  
-----BEGIN CERTIFICATE-----  
MIIEijCCA3KgAwIBAgIQdCea9tmy/T6rK/dDD1isujANBgkqhkiG9w0BAQsFADBU  
MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2xlIFRydXN0IFNlcnZpY2VzMSUw  
...**

A major web site such as Google usually sends multiple certificates for authentication.

The output ends with summary information about the TLS session, including specifics on the cipher suite:

**SSL-Session:  
    Protocol : TLSv1.2  
    Cipher : ECDHE-RSA-AES128-GCM-SHA256  
    Session-ID: A2BBF0E4991E6BBBC318774EEE37CFCB23095CC7640FFC752448D07C7F438573  
...**

The protocol **TLS 1.2** is used in the **client** program, and the **Session-ID** uniquely identifies the connection between the **openssl** utility and the Google web server. The **Cipher** entry can be parsed as follows:

* **ECDHE** (Elliptic Curve Diffie Hellman Ephemeral) is an effective and efficient algorithm for managing the TLS handshake. In particular, ECDHE solves the **key-distribution problem** by ensuring that both parties in a connection (e.g., the client program and the Google web server) use the same encryption/decryption key, which is known as the **session key**. The follow-up article digs into the details.
* **RSA** (Rivest Shamir Adleman) is the dominant public-key cryptosystem and named after the three academics who first described the system in the late 1970s. The key-pairs in play are generated with the RSA algorithm.
* **AES128** (Advanced Encryption Standard) is a **block cipher**that encrypts and decrypts blocks of bits. (The alternative is a **stream cipher**, which encrypts and decrypts bits one at a time.) The cipher is **symmetric**in that the same key is used to encrypt and to decrypt, which raises the key-distribution problem in the first place. AES supports key sizes of 128 (used here), 192, and 256 bits: the larger the key, the better the protection.

Key sizes for symmetric cryptosystems such as AES are, in general, smaller than those for asymmetric (key-pair based) systems such as RSA. For example, a 1024-bit RSA key is relatively small, whereas a 256-bit key is currently the largest for AES.

* **GCM** (Galois Counter Mode) handles the repeated application of a cipher (in this case, AES128) during a secured conversation. AES128 blocks are only 128-bits in size, and a secure conversation is likely to consist of multiple AES128 blocks from one side to the other. GCM is efficient and commonly paired with AES128.
* **SHA256** (Secure Hash Algorithm 256 bits) is the cryptographic hash algorithm in play. The hash values produced are 256 bits in size, although even larger values are possible with SHA.

Cipher suites are in continual development. Not so long ago, for example, Google used the RC4 stream cipher (Ron’s Cipher version 4 after Ron Rivest from RSA). RC4 now has known vulnerabilities, which presumably accounts, at least in part, for Google’s switch to AES128.

## Wrapping up

This first look at OpenSSL, through a secure C web client and various command-line examples, has brought to the fore a handful of topics in need of more clarification. [The next article gets into the details](https://opensource.com/article/19/6/cryptography-basics-openssl-part-2), starting with cryptographic hashes and ending with a fuller discussion of how digital certificates address the key distribution challenge.

# **How to use OpenSSL: Hashes, digital signatures, and more**

Dig deeper into the details of cryptography with OpenSSL: Hashes, digital signatures, digital certificates, and more

The [first article in this series](https://opensource.com/article/19/6/cryptography-basics-openssl-part-1) introduced **hashes, encryption/decryption, digital signatures, and digital certificates** through the OpenSSL libraries and command-line utilities. This second article drills down into the details. Let’s begin with hashes, which are ubiquitous in computing, and consider what makes a hash function ***cryptographic***.

**1) Cryptographic hashes**

The download page for the OpenSSL source code (<https://www.openssl.org/source/>) contains a table with recent versions. Each version comes with two hash values: 160-bit SHA1 and 256-bit SHA256. These values can be used to verify that the downloaded file matches the original in the repository: The downloader re-computes the hash values locally on the downloaded file and then compares the results against the originals. Modern systems have utilities for computing such hashes. Linux, for instance, has **md5sum** and **sha256sum**. OpenSSL itself provides similar command-line utilities.

Hashes are used in many areas of computing. For example, the *Bitcoin block-chain* uses SHA256 hash values as block identifiers. To mine a Bitcoin is to generate a SHA256 hash value that falls below a specified threshold, which means a hash value with at least N leading zeroes. (The value of N can go up or down depending on how productive the mining is at a particular time.) As a point of interest, today’s miners are hardware clusters designed for generating SHA256 hashes in parallel. During a peak time in 2018, Bitcoin miners worldwide generated about 75 million tera hashes per second—yet another incomprehensible number.

Network protocols use hash values as well—often under the name **checksum**—to support message integrity; that is, to assure that a received message is the same as the one sent. The message sender computes the message’s checksum and sends the results along with the message. The receiver re-computes the checksum when the message arrives. If the sent and the recomputed checksum do not match, then something happened to the message in transit, or to the sent checksum, or to both. In this case, the message and its checksum should be sent again, or at least an error condition should be raised. (Low-level network protocols such as UDP do not bother with checksums.)

Other examples of hashes are familiar. Consider a website that requires users to authenticate with a password, which the user enters in their browser. Their password is then sent, encrypted, from the browser to the server via an HTTPS connection to the server. Once the password arrives at the server, it's decrypted for a database table lookup.

What should be stored in this lookup table? Storing the passwords themselves is risky. It’s far less risky is to store a hash generated from a password, perhaps with some *salt* (extra bits) added to taste before the hash value is computed. Your password may be sent to the web server, but the site can assure you that the password is not stored there.

Hash values also occur in various areas of security. For example, hash-based message authentication code ([HMAC](https://en.wikipedia.org/wiki/HMAC)) uses a hash value and a secret cryptographic key to authenticate a message sent over a network. HMAC codes, which are lightweight and easy to use in programs, are popular in web services. An X509 digital certificate includes a hash value known as the ***fingerprint***, which can facilitate certificate verification. An in-memory truststore could be implemented as a lookup table keyed on such fingerprints—as a ***hash map***, which supports constant-time lookups. The fingerprint from an incoming certificate can be compared against the truststore keys for a match.

What special property should a ***cryptographic hash function*** have? It should be *one-way*, which means very difficult to invert. A cryptographic hash function should be relatively straightforward to compute, but computing its inverse—the function that maps the hash value back to the input bitstring—should be computationally intractable. Here is a depiction, with **chf** as a cryptographic hash function and my password **foobar** as the sample input:

**+---+  
foobar—>|chf|—>hash value ## straightforward  
        +--–+**

By contrast, the inverse operation is infeasible:

**+-----------+  
hash value—>|chf inverse|—>foobar ## intractable  
            +-----------+**

Recall, for example, the SHA256 hash function. For an input bitstring of any length N > 0, this function generates a fixed-length hash value of 256 bits; hence, this hash value does not reveal even the input bitstring’s length N, let alone the value of each bit in the string. By the way, SHA256 is not susceptible to a [*length extension attack*](https://en.wikipedia.org/wiki/Length_extension_attack). The only effective way to reverse engineer a computed SHA256 hash value back to the input bitstring is through a brute-force search, which means trying every possible input bitstring until a match with the target hash value is found. Such a search is infeasible on a sound cryptographic hash function such as SHA256.

Now, a final review point is in order. Cryptographic hash values are statistically rather than unconditionally unique, which means that it is unlikely but not impossible for two different input bitstrings to yield the same hash value—a *collision*. The [*birthday problem*](https://en.wikipedia.org/wiki/Birthday_problem) offers a nicely counter-intuitive example of collisions. There is extensive research on various hash algorithms’ *collision resistance*. For example, MD5 (128-bit hash values) has a breakdown in collision resistance after roughly 221 hashes. For SHA1 (160-bit hash values), the breakdown starts at about 261 hashes.

A good estimate of the breakdown in collision resistance for SHA256 is not yet in hand. This fact is not surprising. SHA256 has a range of 2256 distinct hash values, a number whose decimal representation has a whopping 78 digits! So, can collisions occur with SHA256 hashing? Of course, but they are extremely unlikely.

In the command-line examples that follow, two input files are used as bitstring sources: **hashIn1.txt** and **hashIn2.txt**. The first file contains **abc** and the second contains **1a2b3c**.

These files contain text for readability, but binary files could be used instead.

Using the Linux **sha256sum** utility on these two files at the command line—with the percent sign (**%**) as the prompt—produces the following hash values (in hex):

% sha256sum hashIn1.txt  
9e83e05bbf9b5db17ac0deec3b7ce6cba983f6dc50531c7a919f28d5fb3696c3 hashIn1.txt  
  
% sha256sum hashIn2.txt  
3eaac518777682bf4e8840dd012c0b104c2e16009083877675f00e995906ed13 hashIn2.txt

The OpenSSL hashing counterparts yield the same results, as expected:

% openssl dgst -sha256 hashIn1.txt  
SHA256(hashIn1.txt)= 9e83e05bbf9b5db17ac0deec3b7ce6cba983f6dc50531c7a919f28d5fb3696c3  
  
% openssl dgst -sha256 hashIn2.txt  
SHA256(hashIn2.txt)= 3eaac518777682bf4e8840dd012c0b104c2e16009083877675f00e995906ed13

This examination of cryptographic hash functions sets up a closer look at digital signatures and their relationship to key pairs.

**2) Digital signatures**

As the name suggests, a **digital signature** can be attached to a document or some other electronic artifact (e.g., a program) to vouch for its authenticity. *Such a signature is thus analogous to a hand-written signature on a paper document.* To verify the digital signature is to confirm two things. *First*, that the vouched-for artifact has not changed since the signature was attached because it is based, in part, on a cryptographic *hash* of the document. *Second*, that the signature belongs to the person (e.g., Alice) who alone has access to the private key in a pair. By the way, digitally signing code (source or compiled) has become a common practice among programmers.

Let’s walk through how a **digital signature** is created. As mentioned before, there is no digital signature without a public and private key pair. When using OpenSSL to create these keys, there are two separate commands: *one to create a private key, and another to extract the matching public key from the private one.* These key pairs are encoded in base64, and their sizes can be specified during this process.

The private key consists of numeric values, two of which (a modulus and an exponent) make up the public key. Although the private key file contains the public key, the extracted public key does *not* reveal the value of the corresponding private key.

The resulting file with the private key thus contains the full key pair. Extracting the public key into its own file is practical because the two keys have distinct uses, but this extraction also minimizes the danger that the private key might be publicized by accident.

*Next, the pair’s private key is used to process a hash value for the target artifact (e.g., an email), thereby creating the signature. On the other end, the receiver’s system uses the pair’s public key to verify the signature attached to the artifact.*

Now for an example. To begin, generate a 2048-bit RSA key pair with OpenSSL:

$ openssl genpkey -out privkey.pem -algorithm rsa -pkeyopt rsa\_keygen\_bits:2048

We can drop the **-algorithm rsa** flag in this example because **genpkey** defaults to the type RSA. The file’s name (**privkey.pem**) is arbitrary, but the **Privacy Enhanced Mail (PEM)** extension **pem** is customary for the default PEM format. (OpenSSL has commands to convert among formats if needed.) If a larger key size (e.g., 4096) is in order, then the last argument of **2048** could be changed to **4096**. These sizes are always powers of two.

Here’s a slice of the resulting **privkey.pem** file, which is in base64:

-----BEGIN PRIVATE KEY-----  
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBANnlAh4jSKgcNj/Z  
JF4J4WdhkljP2R+TXVGuKVRtPkGAiLWE4BDbgsyKVLfs2EdjKL1U+/qtfhYsqhkK  
…  
-----END PRIVATE KEY-----

The next command then extracts the pair’s public key from the private one:

openssl rsa -in privkey.pem -outform PEM -pubout -out pubkey.pem

The resulting **pubkey.pem** file is small enough to show here in full:

-----BEGIN PUBLIC KEY-----  
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDZ5QIeI0ioHDY/2SReCeFnYZJY  
z9kfk11RrilUbT5BgIi1hOAQ24LMilS37NhHYyi9VPv6rX4WLKoZCmkeYaWk/TR5  
4nbH1E/AkniwRoXpeh5VncwWMuMsL5qPWGY8fuuTE27GhwqBiKQGBOmU+MYlZonO  
O0xnAKpAvysMy7G7qQIDAQAB  
-----END PUBLIC KEY-----

Now, with the key pair at hand, the digital signing is easy—in this case with the source file **client.c** as the artifact to be signed:

openssl dgst -sha256 -sign privkey.pem -out sign.sha256 client.c

The digest for the **client.c** source file is SHA256, and the private key resides in the **privkey.pem** file created earlier. The resulting binary signature file is **sign.sha256**, an arbitrary name. To get a readable (if base64) version of this file, the follow-up command is:

openssl enc -base64 -in sign.sha256 -out sign.sha256.base64

The file **sign.sha256.base64** now contains:

**h+e+3UPx++KKSlWKIk34fQ1g91XKHOGFRmjc0ZHPEyyjP6/lJ05SfjpAJxAPm075  
VNfFwysvqRGmL0jkp/TTdwnDTwt756Ej4X3OwAVeYM7i5DCcjVsQf5+h7JycHKlM  
o/Jd3kUIWUkZ8+Lk0ZwzNzhKJu6LM5KWtL+MhJ2DpVc=**

Or, the executable file **client** could be signed instead, and the resulting base64-encoded signature would differ as expected:

**VMVImPgVLKHxVBapJ8DgLNJUKb98GbXgehRPD8o0ImADhLqlEKVy0HKRm/51m9IX  
xRAN7DoL4Q3uuVmWWi749Vampong/uT5qjgVNTnRt9jON112fzchgEoMb8CHNsCT  
XIMdyaPtnJZdLALw6rwMM55MoLamSc6M/MV1OrJnk/g=**

The final step in this process is to verify the digital signature with the public key. The hash used to sign the artifact (in this case, the executable **client** program) should be recomputed as an essential step in the verification since the verification process should indicate whether the artifact has changed since being signed.

There are two OpenSSL commands used for this purpose. The first decodes the base64 signature:

openssl enc -base64 -d -in sign.sha256.base64 -out sign.sha256

The second verifies the signature:

openssl dgst -sha256 -verify pubkey.pem -signature sign.sha256 client.c

The output from this second command is, as it should be:

Verified OK

To understand what happens when verification fails, a short but useful exercise is to replace the executable **client** file in the last OpenSSL command with the source file **client.c** and then try to verify. Another exercise is to change the **client** program, however slightly, and try again.

**3) Digital certificates**

A **digital certificate** brings together the pieces analyzed so far: ***hash values, key pairs, digital signatures, and encryption/decryption***. The first step toward a production-grade certificate is to create a **certificate signing request (CSR)**, which is then sent to a certificate authority (CA). To do this for the example with OpenSSL, run:

openssl req -out myserver.csr -new -newkey rsa:4096 -nodes -keyout myserverkey.pem

This example generates a CSR document and stores the document in the file **myserver.csr** (base64 text). The purpose here is this: the CSR document requests that the CA vouch for the identity associated with the specified domain name—the common name (CN) in CA-speak.

A new key pair also is generated by this command, although an existing pair could be used. Note that the use of **server** in names such as **myserver.csr** and **myserverkey.pem** hints at the typical use of digital certificates: as vouchers for the identity of a web server associated with a domain such as [www.google.com](http://www.google.com/).

The same command, however, creates a CSR regardless of how the digital certificate might be used. It also starts an interactive question/answer session that prompts for relevant information about the domain name to link with the requester’s digital certificate. This interactive session can be short-circuited by providing the essentials as part of the command, with backslashes as continuations across line breaks. The **-subj** flag introduces the required information:

**% openssl req -new  
-newkey rsa:2048 -nodes -keyout privkeyDC.pem  
-out myserver.csr  
-subj "/C=US/ST=Illinois/L=Chicago/O=Faulty Consulting/OU=IT/CN=myserver.com"**

The resulting CSR document can be inspected and verified before being sent to a CA. This process creates the digital certificate with the desired format (e.g., X509), signature, validity dates, and so on:

openssl req -text -in myserver.csr -noout -verify

Here’s a slice of the output:

**verify OK  
Certificate Request:  
Data:  
Version: 0 (0x0)  
Subject: C=US, ST=Illinois, L=Chicago, O=Faulty Consulting, OU=IT, CN=myserver.com  
Subject Public Key Info:  
Public Key Algorithm: rsaEncryption  
Public-Key: (2048 bit)  
Modulus:  
00:ba:36:fb:57:17:65:bc:40:30:96:1b:6e:de:73:  
…  
Exponent: 65537 (0x10001)  
Attributes:  
a0:00  
Signature Algorithm: sha256WithRSAEncryption  
…**

**4) A self-signed certificate**

During the development of an HTTPS web site, it is convenient to have a digital certificate on hand without going through the CA process. A self-signed certificate fills the bill during the HTTPS handshake’s authentication phase, although any modern browser warns that such a certificate is worthless. Continuing the example, the OpenSSL command for a self-signed certificate—valid for a year and with an RSA public key—is:

**openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:4096 -keyout myserver.pem -out myserver.crt**

The OpenSSL command below presents a readable version of the generated certificate:

**openssl x509 -in myserver.crt -text -noout**

Here’s part of the output for the self-signed certificate:

**Certificate:  
Data:  
Version: 3 (0x2)  
Serial Number: 13951598013130016090 (0xc19e087965a9055a)  
Signature Algorithm: sha256WithRSAEncryption  
Issuer: C=US, ST=Illinois, L=Chicago, O=Faulty Consulting, OU=IT, CN=myserver.com  
Validity  
Not Before: Apr 11 17:22:18 2019 GMT  
Not After : Apr 10 17:22:18 2020 GMT  
Subject: C=US, ST=Illinois, L=Chicago, O=Faulty Consulting, OU=IT, CN=myserver.com  
Subject Public Key Info:  
Public Key Algorithm: rsaEncryption  
Public-Key: (4096 bit)  
Modulus:  
00:ba:36:fb:57:17:65:bc:40:30:96:1b:6e:de:73:  
…  
Exponent: 65537 (0x10001)  
X509v3 extensions:  
X509v3 Subject Key Identifier:  
3A:32:EF:3D:EB:DF:65:E5:A8:96:D7:D7:16:2C:1B:29:AF:46:C4:91  
X509v3 Authority Key Identifier:  
keyid:3A:32:EF:3D:EB:DF:65:E5:A8:96:D7:D7:16:2C:1B:29:AF:46:C4:91  
  
        X509v3 Basic Constraints:  
            CA:TRUE  
Signature Algorithm: sha256WithRSAEncryption  
     3a:eb:8d:09:53:3b:5c:2e:48:ed:14:ce:f9:20:01:4e:90:c9:  
     ...**

As mentioned earlier, an RSA private key contains values from which the public key is generated. However, a given public key does *not* give away the matching private key. For an introduction to the underlying mathematics, see <https://simple.wikipedia.org/wiki/RSA_algorithm>.

There is an important correspondence between a digital certificate and the key pair used to generate the certificate, even if the certificate is only self-signed:

* The digital certificate contains the *exponent* and *modulus* values that make up the public key. These values are part of the key pair in the originally-generated PEM file, in this case, the file **myserver.pem**.
* The exponent is almost always 65,537 (as in this case) and so can be ignored.
* The modulus from the key pair should match the modulus from the digital certificate.

The modulus is a large value and, for readability, can be hashed. Here are two OpenSSL commands that check for the same modulus, thereby confirming that the digital certificate is based upon the key pair in the PEM file:

**% openssl x509 -noout -modulus -in myserver.crt | openssl sha1 ## modulus from CRT  
(stdin)= 364d21d5e53a59d482395b1885aa2c3a5d2e3769  
  
% openssl rsa -noout -modulus -in myserver.pem | openssl sha1 ## modulus from PEM  
(stdin)= 364d21d5e53a59d482395b1885aa2c3a5d2e3769**

The resulting hash values match, thereby confirming that the digital certificate is based upon the specified key pair.

**5) Back to the key distribution problem**

Let’s return to an issue raised at the end of Part 1: the TLS handshake between the **client** program and the Google web server. There are various handshake protocols, and even the Diffie-Hellman version at work in the **client** example offers wiggle room. Nonetheless, the **client** example follows a common pattern.

To start, during the TLS handshake, the **client** program and the web server agree on a cipher suite, which consists of the algorithms to use. In this case, the suite is **ECDHE-RSA-AES128-GCM-SHA256**.

The two elements of interest now are the RSA key-pair algorithm and the AES128 block cipher used for encrypting and decrypting messages if the handshake succeeds. Regarding encryption/decryption, this process comes in two flavors: symmetric and asymmetric. In the symmetric flavor, the *same* key is used to encrypt and decrypt, which raises the *key distribution problem* in the first place: How is the key to be distributed securely to both parties? In the asymmetric flavor, one key is used to encrypt (in this case, the RSA public key) but a different key is used to decrypt (in this case, the RSA private key from the same pair).

The **client** program has the Google web server’s public key from an authenticating certificate, and the web server has the private key from the same pair. Accordingly, the **client** program can send an encrypted message to the web server, which alone can readily decrypt this message.

In the TLS situation, the symmetric approach has two significant advantages:

* In the interaction between the **client** program and the Google web server, the authentication is one-way. The Google web server sends three certificates to the **client** program, but the **client** program does not send a certificate to the web server; hence, the web server has no public key from the client and can’t encrypt messages to the client.
* Symmetric encryption/decryption with AES128 is nearly a *thousand times faster* than the asymmetric alternative using RSA keys.

The TLS handshake combines the two flavors of encryption/decryption in a clever way. During the handshake, the **client** program generates random bits known as the pre-master secret (PMS). Then the **client** program encrypts the PMS with the server’s public key and sends the encrypted PMS to the server, which in turn decrypts the PMS message with its private key from the RSA pair:

**+-------------------+ encrypted PMS  +--------------------+  
client PMS--->|server’s public key|--------------->|server’s private key|--->server PMS  
              +-------------------+                +--------------------+**

At the end of this process, the **client** program and the Google web server now have the same PMS bits. Each side uses these bits to generate a *master secret* and, in short order, a symmetric encryption/decryption key known as the *session key*. There are now two distinct but identical session keys, one on each side of the connection. In the **client** example, the session key is of the AES128 variety. Once generated on both the **client** program’s and Google web server’s sides, the session key on each side keeps the conversation between the two sides confidential. A handshake protocol such as Diffie-Hellman allows the entire PMS process to be repeated if either side (e.g., the **client** program) or the other (in this case, the Google web server) calls for a restart of the handshake.

**Wrapping up**

The OpenSSL operations illustrated at the command line are available, too, through the API for the underlying libraries. These two articles have emphasized the utilities to keep the examples short and to focus on the cryptographic topics. If you have an interest in security issues, OpenSSL is a fine place to start—and to stay.

A cipher is a method (*algorithm*) used for encryption of some text. But English speakers have that habit of making verbs from nouns... hence ciphering became a synonym of encrypting.

Now, the fun part. If you consider decrypt and decipher, now they have different meanings.

* **decrypt** means applying the decryption key to some code
* **decipher** means finding the meaning of some text that was not deliberately encrypted.

In France (I'm french) we also have funny confusion with similar words. We have "chiffrer" (very similar to "cipher") that is the correct word and means encrypt, but we also use the verb "crypter" that means the same thing but is considered as an anglicism (verb built from english "crypted"). When we go for the opposite words "décrypter" and "dechiffrer" we also have different meanings but not like the english ones... "déchiffrer" means the same that both english words decrypt and decipher depending on the case, but "décrypter" is used when you try to get the clear text without the code (it means breaking the code). I believe there is no english word that means that.

Looking at my answer, I wonder if things were not clearer before it.... natural language is definitely some kind of encryption.