A Practical Introduction To OpenSSL

Produced by: Michele La Manna

Department of Information Engineering (DII)

University of Pisa

Table of Contents

About This File

This file was created for the benefit of all teachers and students wanting to use OpenSSL for tests/exams/lessons/thesis/articles etc.

Special Thanks to Dr. Pericle Perazzo for greatly contributing to this guide, and to Prof. Gianluca Dini for reviewing and editing the final version of this guide.

SECTION 1

Brief Introduction to OpenSSL

The OpenSSL Project develops and maintains the OpenSSL software - a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication.
In this brief compendium, you will find a list of useful APIs to begin your adventure in programming with cryptography!

1.1 What's OpenSSL?

The OpenSSL project includes a core C library and a toolkit. The core C library provides many cryptographic APIs at different layers, while the toolkit provides several command-line tools for cryptographic primitives, in particular `Key Generation` primitives.

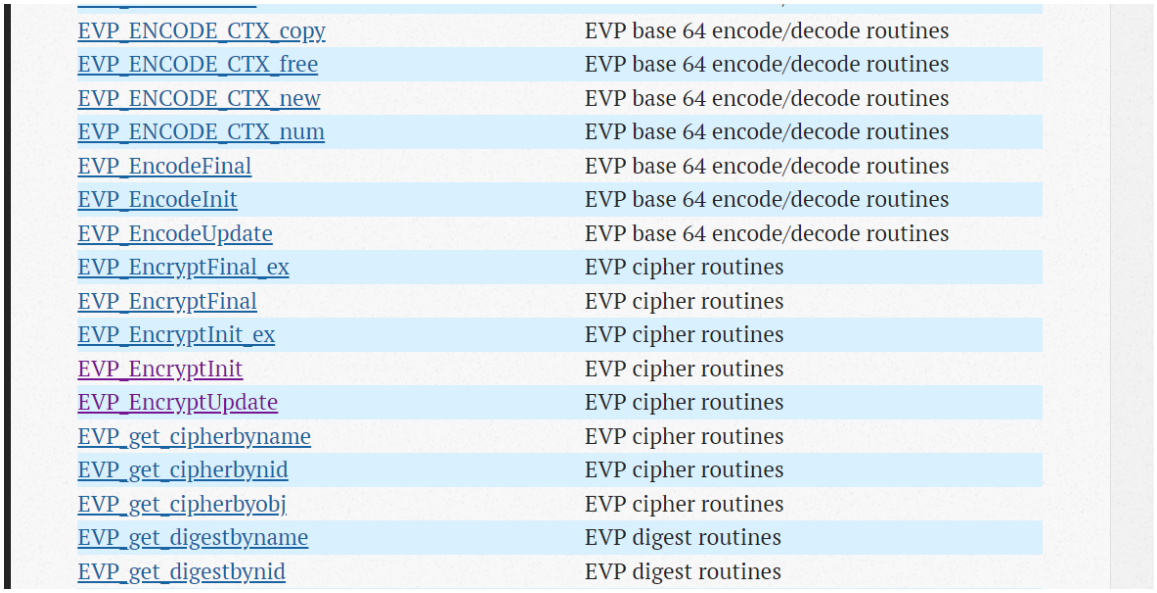
1.2 Installation & Linking

On Unix-like platforms you simply need to install `libssl-dev` packages. On Windows, you can simply download the binary from the repository https://wiki.openssl.org/index.php/Binaries and launch the installer. On any platform, when you compile a `C/C++` file using OpenSSL APIs, you need to link the `lcrypto` library. For example:
`gcc mysource.c -lcrypto`.

1.3 OpenSSL Documentation

At the following link you can search for any API provided by OpenSSL: https://www.openssl.org/docs/man1.1.1/man3/. You will be presented with a literal list of all the APIs. With a simple `CTRL+F`, you can search for the API you are looking for, and click on its hyperlink. As an ex-

ample, you can see in Fig. 1 the hyperlink related to the API `EVP_Encrypt_Init()`. At this point



EVP_ENCODE_CTX_copy	EVP base 64 encode/decode routines
EVP_ENCODE_CTX_free	EVP base 64 encode/decode routines
EVP_ENCODE_CTX_new	EVP base 64 encode/decode routines
EVP_ENCODE_CTX_num	EVP base 64 encode/decode routines
EVP_EncodeFinal	EVP base 64 encode/decode routines
EVP_EncodeInit	EVP base 64 encode/decode routines
EVP_EncodeUpdate	EVP base 64 encode/decode routines
EVP_EncryptFinal_ex	EVP cipher routines
EVP_EncryptFinal	EVP cipher routines
EVP_EncryptInit_ex	EVP cipher routines
EVP_EncryptInit	EVP cipher routines
EVP_EncryptUpdate	EVP cipher routines
EVP_get_cipherbyname	EVP cipher routines
EVP_get_cipherbynid	EVP cipher routines
EVP_get_cipherbyobj	EVP cipher routines
EVP_get_digestbyname	EVP digest routines
EVP_get_digestbynid	EVP digest routines

Figure 1: Search for the `EVP_EncryptInit()` API.

you would expect a clean description of the selected API. Instead, as you can see in Fig. 2, you will find a plethora of somewhat "affine" APIs, including the one you have clicked on. In green text with black background, you can see the definition of the APIs. In the plaintext description under the definitions, instead, you will find aggregate explanations for the APIs arguments, behavior, and returns.

Good luck! (You'll need it!)

EVP_EncryptInit

NAME

EVP_CIPHER_CTX_new, EVP_CIPHER_CTX_reset, EVP_CIPHER_CTX_free, EVP_EncryptInit_ex, EVP_EncryptUpdate, EVP_EncryptFinal_ex, EVP_DecryptInit_ex, EVP_DecryptUpdate, EVP_DecryptFinal_ex, EVP_CipherInit_ex, EVP_CipherUpdate, EVP_CipherFinal_ex, EVP_CIPHER_CTX_set_key_length, EVP_CIPHER_CTX_ctrl, EVP_EncryptInit, EVP_EncryptFinal, EVP_DecryptInit, EVP_DecryptFinal, EVP_CipherInit, EVP_CipherFinal, EVP_get_cipherbyname, EVP_get_cipherbynid, EVP_get_cipherbyobj, EVP_CIPHER_nid, EVP_CIPHER_block_size, EVP_CIPHER_key_length, EVP_CIPHER_iv_length, EVP_CIPHER_flags, EVP_CIPHER_mode, EVP_CIPHER_type, EVP_CIPHER_CTX_cipher, EVP_CIPHER_CTX_nid, EVP_CIPHER_CTX_block_size, EVP_CIPHER_CTX_key_length, EVP_CIPHER_CTX_iv_length, EVP_CIPHER_CTX_get_app_data, EVP_CIPHER_CTX_set_app_data, EVP_CIPHER_CTX_type, EVP_CIPHER_CTX_flags, EVP_CIPHER_CTX_mode, EVP_CIPHER_param_to_asn1, EVP_CIPHER_asn1_to_param, EVP_CIPHER_CTX_set_padding, EVP_enc_null - EVP cipher routines

SYNOPSIS

```
#include <openssl/evp.h>

EVP_CIPHER_CTX *EVP_CIPHER_CTX_new(void);
int EVP_CIPHER_CTX_reset(EVP_CIPHER_CTX *ctx);
void EVP_CIPHER_CTX_free(EVP_CIPHER_CTX *ctx);

int EVP_EncryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                       ENGINE *impl, const unsigned char *key, const unsigned char *iv);
int EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx, unsigned char *out,
                      int *outl, const unsigned char *in, int inl);
int EVP_EncryptFinal_ex(EVP_CIPHER_CTX *ctx, unsigned char *out, int *outl);

int EVP_DecryptInit_ex(EVP_CIPHER_CTX *ctx, const EVP_CIPHER *type,
                       ENGINE *impl, const unsigned char *key, const unsigned char *iv);
```

Figure 2: OpenSSL documentation for a group of APIs.

SECTION 2

Useful Concepts

In this chapter you will not find any “cryptogrphic" API, rather some high-level description of structure, paradigms, or – more in general – concepts, as well as “utility" APIs. Whenever you do not understand a keyword, refer to this chapter and re-read the section dedicated to it.

2.1 OpenSSL Context

OpenSSL APIs often make use of a so-called *context*. A context is a structure that “manages" the low-level cryptographic operation for the programmer. A realistic pseudo-code on the use of a context is the following:

```
1. create context;
2. initialize context; // Tell the context what you want to do, give it some inputs.
3. cycle
     output_buffer = context_update(input_data_fragment);
end
4. output_buffer = context_finalize; // End the cryptographic operation.
```

The creation and the initialization of a context are two separate steps because in OpenSSL there are many kind of context and each of them can be initialized in almost infinite ways. The creation of the context determines the type of cryptographic operation (e.g., symmetric encryption, hashing, signature, etc...). The initialization of the context consists in providing to the context the needed “tools" to perform the operation needed.
For example, assume that you need to symmetrically encrypt a file. You will need to create a context that is able to manage symmetric encryption. During the initialization you will need to

specify the *algorithm* (e.g., AES, DES, 3DES,...), the *encryption mode* (e.g., ECB, CBC, GCM,...), the *symmetric key*, the *Initialization Vector* (IV, if required by the encryption mode).
Then, and only then, you are able to encrypt the file. However, if the file is huge, you cannot encrypt it with a single iteration, you need to divide the plaintext in fragments, and encrypt it a little bit at a time. This process is called *context update*, and as you may have guessed, is executed inside a cycle.
Finally, when the whole file has been passed to the context update process, you have to end the operation by *finalizing the context*. In our example, the finalize will add the padding (if required by the encryption mode) and output the last chunk of the ciphertext.

2.2 Blocks Vs. Fragments

Other two useful concepts regarding symmetric encryption are *blocks* and *fragments*. Blocks are "slices" of data of fixed size, determined by the encryption algorithm. As an example, DES has block size 8 bytes. Fig. 3 shows a plaintext that has to be encrypted with DES. In the figure, each square represents a byte and each row represents a block.
A *fragment*, instead, is still a "slice" of data but with an arbitrary amount of bytes, choosen by the programmer. Blocks and fragments are involved in the context update process: the API takes *one* fragment of data in input, and gives *zero or more* blocks in output. The amounts of blocks given in output depends, of course, on the size of the input fragment.
Fig. 4 shows the first iteration example of an encryption context update. In Fig. 4a the bytes in light blue constitute the input fragment: such a fragment is composed of two 8-byte blocks and two bytes. As a consequence, the update produces two blocks whereas two bytes are left in the context. With reference to Fig. 4b, the bytes in the output blocks are in green whereas the bytes in the context are in yellow. Fig. 5 shows the second iteration. In Fig. 5a two types of bytes are visible: the yellow bytes are the "leftovers" from the previous iteration, while the light blue bytes are the new fragment provided to the context. This time, the input fragment has different size (9 bytes), therefore the context can give as output only one encrypted block, as shown in Fig. 5b.

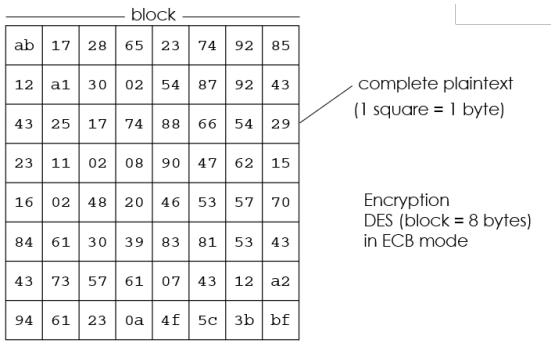
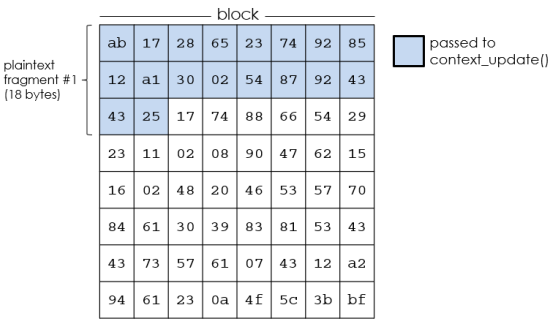
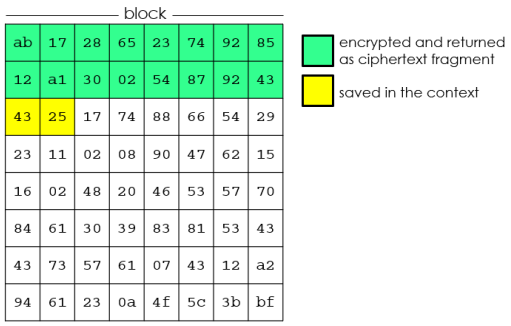


Figure 3: A representation of a plaintext. Each square is a byte, each row is a block in DES.



ab	17	28	65	23	74	92	85
12	a1	30	02	54	87	92	43
43	25	17	74	88	66	54	29
23	11	02	08	90	47	62	15
16	02	48	20	46	53	57	70
84	61	30	39	83	81	53	43
43	73	57	61	07	43	12	a2
94	61	23	0a	4f	5c	3b	bf

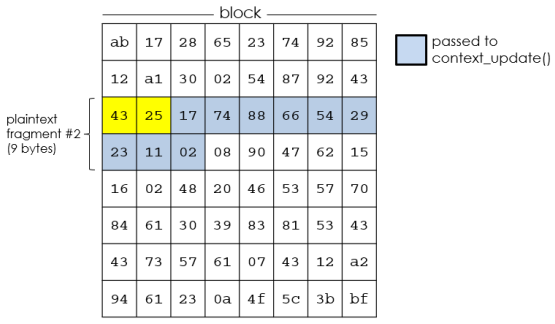
(a) First Iteration Input.



ab	17	28	65	23	74	92	85
12	a1	30	02	54	87	92	43
43	25	17	74	88	66	54	29
23	11	02	08	90	47	62	15
16	02	48	20	46	53	57	70
84	61	30	39	83	81	53	43
43	73	57	61	07	43	12	a2
94	61	23	0a	4f	5c	3b	bf

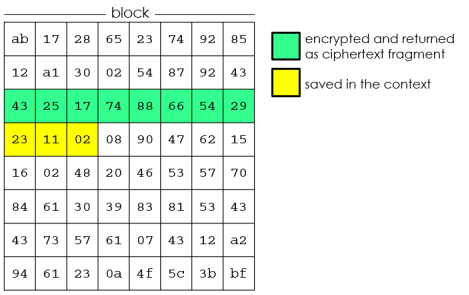
(b) First Iteration Output.

Figure 4: First Encryption Iteration.



ab	17	28	65	23	74	92	85
12	a1	30	02	54	87	92	43
43	25	17	74	88	66	54	29
23	11	02	08	90	47	62	15
16	02	48	20	46	53	57	70
84	61	30	39	83	81	53	43
43	73	57	61	07	43	12	a2
94	61	23	0a	4f	5c	3b	bf

(a) Second Iteration Input.



ab	17	28	65	23	74	92	85
12	a1	30	02	54	87	92	43
43	25	17	74	88	66	54	29
23	11	02	08	90	47	62	15
16	02	48	20	46	53	57	70
84	61	30	39	83	81	53	43
43	73	57	61	07	43	12	a2
94	61	23	0a	4f	5c	3b	bf

(b) Second Iteration Output.

Figure 5: Second Encryption Iteration.

2.3 PKCS#7 Padding

Since in the ECB and CBC modes the encryption works on plaintext blocks of fixed size, the *context finalize* step adds the necessary padding to the plaintext last block before encrypting it. OpenSSL uses the PKCS#7 standard for padding, by which the padding bytes have the same value of the padding length. For example, Fig. 6 shows 2 bytes padding each of value 0x02. The padding is ALWAYS added, so if the plaintext length is already a multiple of the block, a block-long padding is added (e.g., 8 bytes of value 0x08 in DES). As a consequence, the ciphertext length is always (strictly) greater than the plaintext length. On the other hand, the length of the ciphertext fragments (i.e., the context update's output) is always a multiple of the block size. The "context finalization" will encrypt the remaining bytes stored in the context (plus the padding), and it will return a final ciphertext fragment. During *decryption*, the context finalize step also checks for the validity of the padding, returns an error in case, and deletes it. If the padding is not valid, the ciphertext is considered corrupted and the decrypted data must be discarded. Please remember that non every encryption mode requires padding.

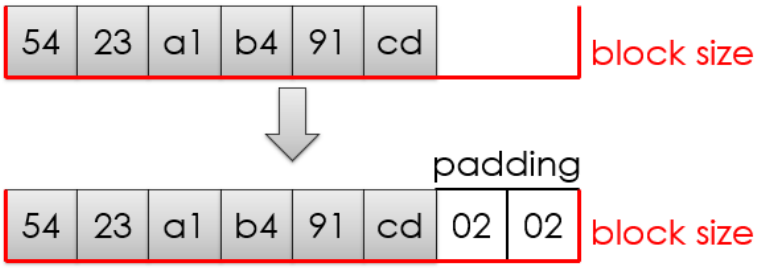


Figure 6: An example of 2-byte padding in a 8-byte block.

2.4 Randomness

The generation of unpredictable random numbers is often an underrated aspect of security systems, causing many vulnerabilities. Generating practically unpredictable random numbers requires to select a good Pseudo-Random Number Generator (PRNG) and good seeds for it. It is always preferable to use a cryptographic library like OpenSSL to generate unpredictable random numbers. The following example code shows the generation of a random key and a random IV for successive encryption.

```
1  #include <openssl/rand.h>
2  int main() {
3    RAND_poll();
4    unsigned char key[16];
5    unsigned char iv[16];
6    RAND_bytes(key, 16);
7    RAND_bytes(iv, 16);
8    /* proceeds with encryption */
9    return 0;
10  }
```

At line 3, `RAND_poll()` seeds the PRNG with a good seed, extracted from the /dev/urandom virtual device on UNIX-like operating systems, or from a combination of `CryptGenRandom()` and other randomicity sources on Windows. Calling `RAND_poll()` is not *strictly* necessary, because it is automatically called at the program start. However, the PRNG should be reseeded by calling `RAND_poll()` after generating large quantities of random bytes. `RAND_poll()` is available since OpenSSL 1.1.0. In previous versions of OpenSSL (1.0.x), the PRNG was automatically seeded from /dev/urandom only if available. Otherwise, whenever /dev/urandom is not available, the PRNG had to be seeded by hand, which is a very risky operation. For instance, in OpenSSL 1.0.x over the Win32 operating systems, a good way to seed the PRNG is to call `RAND_screen()` which takes randomicity from the current content on the display. However, `RAND_screen()` exists only for compatibility reasons and it is deprecated, so avoid it if possible. At lines 6 and 7, `RAND_bytes(buffer, amount)` generates a number `amount` of random bytes, and stores them in the specified (preallocated) `buffer`.

WARNING!
Random bytes (e.g., keys, IVs, ciphertexts...) ARE NOT STRINGS!
DO NOT use `strlen()`, `strcpy()`, etc. on them!
Those functions interact with the null terminator, therefore if a sequence of random bytes does not have a null terminator, or if a null terminator is contained in the middle of the sequence, your program will malfunction.
Use SEPARATE VARIABLES to save keys', IVs', or ciphertexts' lengths.
Use `memcpy()` to copy sequence of random bytes, and use binary mode to load/save them from/to files.

2.5 The PEM Format

PEM (Privacy-Enhanced Mail) is a 1993 IETF standard for securing eMail communications using asymmetric cryptography. The standard became obsolete once PGP was published, however, the corresponding file format was later adopted in many applications (mainly serialization of non-textual, non-audio, and non-video files). PEM is a textual format, in which cryptographic quantities are surrounded by tags, for example, `"---BEGIN PUBLIC KEY---"`, `"---END PUBLIC KEY---"`. PEM can represent public or private keys (both RSA and EC), digital certificates, Diffie-Hellman parameters, and so on. In the following, you can see a representation of an RSA public key stored in PEM format.

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDerpZiO7yI0Gp7i+EathC74vXv
hEri18e3isbVc0oMO/06cpBZ+8+kvMSOxSrxkz12CBDRkyhob0+0lLpz2PzXx1Gl
qfpMddnulZnaXRYh/UtcUZL5VLq2rMyQ4yeZ1hA6gNfw/5M9me97heyy4gogLJAq
zx52ol6c/thpDWQecQIDAQAB
-----END PUBLIC KEY-----
```

2.5.1 (De)Serialization of Data Structures in OpenSSL

In OpenSSL, you will need to send many data structures (e.g., public keys, certificates, etc...) to your peer over the Internet. To do so, one option is to serialize the data structure you want to transmit in the PEM format. In OpenSSL there is a special "auxiliary" structure that is useful to serialize data structure into PEM format and vice-versa. This structure is basically a character stream and it is called a `BIO`.

`struct BIO`
An instance of this structure represents a generic character stream. Such instance can be associated to a file (`file BIO`), a socket (`socket BIO`), or a memory buffer (`memory BIO`), to create different types of stream.

`BIO* BIO_new(BIO_s_mem());`
Creates a new *memory BIO*. A *memory BIO* keeps an internal memory buffer, structured as a queue of bytes. Writing on the BIO results in data to be inserted in the queue. Reading from

the BIO results in data to be extracted from the queue. The BIO automatically manages memory allocation issues.

`BIO_free(BIO* bio)`
Deallocates a BIO.

`int PEM_write_bio_PUBKEY(BIO* bio, EVP_PKEY* pubkey);`
Serializes a public key (saved in an `EVP_PKEY` structure) into PEM format and writes it in the BIO.

`int PEM_write_bio_X509(BIO* bio, X509* cert);`
Serializes an X.509 certificate data structure into PEM format and stores it in the BIO.

`int BIO_read(BIO* bio, void* data, int dlen);`
Reads and extracts data from a BIO.

`long BIO_get_mem_data(BIO* mbio, char** pp);`
This API works only with memory BIOs. Sets the pointer `pp` to point to the internal buffer of the memory BIO, and returns its size. Useful to get the whole memory BIO content. It does not extract data from the BIO.

`int BIO_write(BIO* bio, const void* data, int dlen);`
Writes `dlen` bytes from the buffer *data* on the BIO `bio`.

`EVP_PKEY* PEM_read_bio_PUBKEY(BIO* bio, NULL, NULL, NULL);`
Reads a public key written in PEM format from the BIO `bio` and deserializes it. It returns a pointer to the deserialized data structure containing the public key.

`X509* PEM_read_bio_X509(BIO* bio, NULL, NULL, NULL);`
Reads a certificate written in PEM format from the BIO `bio` and deserializes it. It returns a pointer to the deserialized data structure containing the certificate.

2.6 Error Management in OpenSSL

To aid the programmer in the debugging process, OpenSSL provides some API to translate return error codes into natural language. If you want to use this feature, at the beginning of the main program you have to execute the following API:

`void SSL_load_error_strings();`
The function initializes the internal OpenSSL table of error descriptions.

The following code snippet represents a typical case in which you may want to use this feature:

```
int ret;
ret = SOME_OPENSSL_API(...);
```

```
if(ret != 1) { /* 1 means success in OpenSSL convention */
    ERR_print_errors_fp(stderr);
    /* ... Handle error ... */
}
```

In OpenSSL, errors are organized as a queue of error codes, which are integers.

`void ERR_print_errors_fp(stderr);`
Prints on standard error a human-readable description of all the errors in the error queue, and empties the queue.

Moreover, OpenSSL provides other two APIs if you want to manually extracts and analyze errors in the queue:

`int ERR_get_error();`
Extracts an error code from the queue and returns it, or returns 0 if there are no errors.

`char* ERR_error_string(int errcode, NULL);`
Obtains a human-readable description of an error code. The returned string must not be deallocated. The description has the following format:
`error:[error code]:[library name]:[function name]:[reason string]`

SECTION 3

Symmetric Encryption

The logical representation of symmetric encryption is shown in Fig. 7. Considering the encryption function as a block, it needs two inputs (the plaintext p and the key k), and one output (the ciphertext ct).

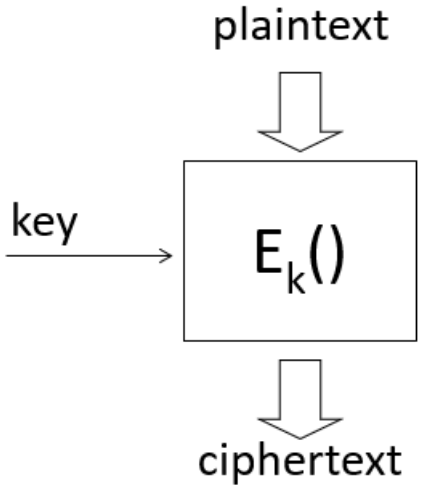


Figure 7: Logical representation of the symmetric encryption operation.

The logical representation suggests that we can encrypt the whole plaintext in one iteration, but for large files, this is not practical. Indeed, when a plaintext is encrypted in one iteration, you need to allocate in the memory two buffers (at least) as large as the plaintext itself: one for the input, and one for the output of the function. If you encrypt a 4GB file, you need *at least* 8GB of RAM!

With reference to section 2.2, instead, we can split a large plaintext in multiple fragments. At each encryption iteration, we load a fragment from mass storage to memory, perform the context update, and store the resulting piece of ciphertext in mass memory. In this way, you can efficiently manage your RAM.

3.1 OpenSSL Symmetric Encryption APIs

To use the APIs contained in this section, you will need to include the `openssl/evp.h` library in your source code.
Each macrocategory of operation is managed by a specific context. The following APIs refers to the `EVP_CIPHER_CTX` structure, used – among other operations – to manage symmetric encryption and decryption operations.

```
#Context creation:
    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
#Encryption:
    EVP_EncryptInit(ctx, Algorithm&Mode, key, iv);
    EVP_EncryptUpdate(ctx, outbuf, &outlen, inbuf, inlen);
    EVP_EncryptFinal(ctx, outbuf, &outlen);
#Decryption:
    EVP_DecryptInit(ctx, EVP_aes_128_ecb(), key, iv);
    EVP_DecryptUpdate(ctx, outbuf, &outlen, inbuf, inlen);
    EVP_DecryptFinal(ctx, outbuf, &outlen);
#Context destruction:
    EVP_CIPHER_CTX_free(ctx);
```

`EVP_CIPHER_CTX_new()`
It allocates a context object, and it returns a pointer to it.
`EVP_EncryptInit(ctx, Algorithm&Mode, key, iv)`

- `ctx`: a pointer to the created `EVP_CIPHER_CTX` object;
- `Algorithm&Mode`: a specifier of the algorithm and the encryption mode to perform. An example is `EVP_aes_128_ecb()`.
- `key`: the symmetric key, of opportune length. It can be a `char*` pointing to a buffer.
- `iv`: must be provided if required by the encryption mode, otherwise it must be set to `NULL`.

`EVP_EncryptUpdate(ctx, outbuf, &outlen, inbuf, inlen)`

- `ctx`: a pointer to the created and initialized `EVP_CIPHER_CTX` object;
- `outbuf`: a pointer to an already allocated output buffer (the ciphertext blocks are written in here);
- `outlen`: a `size_t` variable in which there will be written – in bytes – the amount of data written to `outbuf`;

- `inbuf`: pointer to the buffer containing the input fragment;
- `inlen`: length – in bytes – of the input fragment.
- **Notes:** to be safe, outbuf must have size equal to `inlen + blocksize`. Moreover, don't forget to pass to this API the *address* of outlen, and not his value!

`EVP_EncryptFinal(ctx, outbuf, &outlen)`

- `ctx`: pointer to the context to which has been given all of the plaintext;
- `outbuf`: a pointer to an already allocated output buffer. The final ciphertext block is written in here, containing the "leftover" bytes in the context, if any, plus the padding, if required;
- `outlen`: a `size_t` variable in which there will be written – in bytes – the amount of data written to `outbuf`;

`EVP_DecryptInit(ctx, Algorithm&Mode, key, iv)`

- `ctx`: a pointer to the created `EVP_CIPHER_CTX` object;
- `Algorithm&Mode`: a specifier of the algorithm and the encryption mode to perform. An example is `EVP_aes_128_ecb()`.
- `key`: the symmetric key, of opportune length. It can be a `char*` pointing to a buffer.
- `iv`: must be provided if required by the encryption mode, otherwise it must be set to `NULL`.

`EVP_DecryptUpdate(ctx, outbuf, &outlen, inbuf, inlen)`

- `ctx`: a pointer to the created and initialized `EVP_CIPHER_CTX` object;
- `outbuf`: a pointer to an already allocated output buffer (the plaintext blocks are written in here);
- `outlen`: a `size_t` variable in which there will be returned – in bytes – the amount of data written to `outbuf`;
- `inbuf`: pointer to the buffer containing the input fragment of the ciphertext;
- `inlen`: length – in bytes – of the input fragment.
- **Notes:** to be safe, outbuf must have size equal to `inlen + blocksize`. Moreover, don't forget to pass to this API the *address* of outlen, and not his value!

`EVP_DecryptFinal(ctx, outbuf, &outlen)`

- `ctx`: pointer to the context to which has been given all of the ciphertext;
- `outbuf`: a pointer to an already allocated output buffer. The final plaintext block is written in here, containing the "leftover" bytes in the context without the padding, if present;

- `outlen`: a `size_t` variable in which there will be written – in bytes – the amount of data written to `outbuf`;
- **Note:** this function checks the correctness of the padding, if any. It will return an error in case of wrong padding.

`EVP_CIPHER_CTX_free(ctx)`
It deallocates the context object. It MUST be called, not for memory efficiency, but for security reasons. Leaving the context in the heap means leaving sensible information, such as the symmetric key, inside the memory. This can lead to key compromise.

3.2 Symmetric Encryption Essentials

```
1  #include <openssl/evp.h>
2  int main() {
3        char msg[] = "Lorem ipsum dolor sit amet.";
4        unsigned char key[] = {0x79, 0x33, 0x64, /* more bytes needed */};
5        unsigned char* ciphertext;
6        EVP_CIPHER_CTX* ctx;
7        int cipherlen;
8        int outlen;
9
10       /* Buffer allocation for the ciphertext */
11       ciphertext = (unsigned char*)malloc(sizeof(msg) + 16);
12
13       /* Context allocation */
14       ctx = EVP_CIPHER_CTX_new();
15       /* Encryption (initialization + single update + finalization */
16       EVP_EncryptInit(ctx, EVP_aes_128_ecb(), key, NULL);
17       EVP_EncryptUpdate(ctx, ciphertext, &outlen,
18         (unsigned char*)msg, sizeof(msg));
19       cipherlen = outlen;
20       EVP_EncryptFinal(ctx, ciphertext + cipherlen, &outlen);
21       cipherlen += outlen;
22
23       /* Context deallocation */
24       EVP_CIPHER_CTX_free(ctx);
25
26       /* ... Print ciphertext on screen in hexadecimal digits ... */
27       printf("Ciphertext is:\n");
28       BIO_dump_fp(stdout, (const char *)ciphertext, cipherlen);
29       return 0;
30     }
```

First of all, let's address something that is here just for didactic purposes:

- Line 3. The plaintext is hard-coded for simplicity reasons. Most of the time you will retrieve your plaintext from a file or from the standard input. You know how to do that.
- Line 4. The key is also hard-coded and *clearly* this is a problem. Refer to Section 2.4 to see how appropriate random bytes are generated.
- Line 4. The key is also too short, you need 16 bytes in this example.
- Lines 16, 17, 20. There is no error checking on the return value of OpenSSL function. Refer to section 2.6 to learn how to manage errors in OpenSSL.

Now, we can focus on the purpose of this example, namely the structure of the code. Besides the variable allocation, the most of the OpenSSL operations' core will follow this scheme: *Creation*, *Initialization*, *Update*, *Finalize*, *Deallocation*.
Now, some small details. We are using AES_128 as encryption scheme and ECB as encryption mode. This decisions impact on the following things:

- At Line 11 we allocate the output buffer that will contain the ciphertext. It has the size of the plaintext, plus one block which is 16 bytes.
- At Line 16, there is no IV (the last parameter is set to `NULL`), since ECB does not require it.
- At Lines 20, the `"outbuf"` parameter points to the first empty byte of the ciphertext array, since the `Update` API has already written the first blocks.
- At Line 21, the variable `cipherlen` is summed to the return variable `outlen`. This is because we have encrypted the file in a single `Update` process, and the output buffer is still the same.
- At Line 28, you can find a useful OpenSSL API that writes, in hexadecimal, the bytes values of a buffer.

The example of a simple decryption is left to the student as an exercise.

3.3 Other Ciphers & Utility Functions

In the following you find the most common ciphers (plus the modes) used in OpenSSL. AES is the recommended choice, since the other ciphers are obsolete and insecure. AES with 128-bit keys (in CBC mode) is the best choice for the majority of the applications. AES with 256-bit keys (in CBC mode) represents TOP SECRET security, but it is less efficient than 128-bit keys.

```
struct EVP_CIPHER;
```
Represents a cipher algorithm (AES-128, etc.) plus a mode (ECB, CBC, etc.).

```
EVP_CIPHER* EVP_des_ede3_cbc();
```
Triple DES (EDE) with 3 keys in CBC mode. It uses 168-bit key (mapped on 192 bits) with block size of 64-bit. 3DES is bbsolete since AES provides more security with less key bits.

```
EVP_CIPHER* EVP_aes_128_cbc();
```

AES with 128-bit keys in CBC mode, with 128-bit block size. It is the best tradeoff between security and performance 99% of the time.

`EVP_CIPHER* EVP_aes_256_cbc();`
AES with 256-bit keys in CBC mode. Counter-intuitively, the block size is still 128-bit. TOP-SECRET security.

Moreover, with the following utility functions, you can retrieve useful information from any given `EVP_CIPHER` object.

`EVP_CIPHER_key_length(EVP_aes_128_cbc());`
Returns the size (in bytes) of the needed key.

`EVP_CIPHER_block_size(EVP_aes_128_cbc());`
Returns the size (in bytes) of the block.

`EVP_CIPHER_iv_length(EVP_aes_128_cbc());`
Returns the size (in bytes) of the initialization vector (0 for ECB modes).

SECTION 4

Asymmetric Encryption

An asymmetric cryptosystem uses two keys, called *key-pair*: a private key, and a public key. An asymmetric key-pair is not a simple string of bits like a symmetric key, but it has an internal structure. Fig 8 shows how an RSA public/private key-pair is internally represented in OpenSSL.

```
typedef struct {
        BIGNUM *n; // public modulus        } public key
        BIGNUM *e; // public exponent       }
        BIGNUM *d; // private exponent       } private key
        BIGNUM *p; // secret prime factor    }
        BIGNUM *q; // secret prime factor    }
        BIGNUM *dmp1; // d mod (p-1)         }
        BIGNUM *dmq1; // d mod (q-1)         }
        BIGNUM *iqmp; // q^-1 mod p          }
// ...
}RSA;
```

Figure 8: Low-level structure that represents an RSA keypair.

`BIGNUM` is a data structure that represents an integer represented on a variable number of bits. The first two `BIGNUM`'s represent the public key: the modulus `n`, and the public exponent `e`. The eight instances of the structure `BIGNUM` represent the private key, in particular the private exponent `d`. Luckily, the high-level OpenSSL APIs included in the `<openssl/evp.h>` library manage this low level structure for you.

The RSA scheme (Rivest-Shamir-Adleman, from the names of its inventors) is one of the oldest and by far the most famous type of asymmetric scheme. It is based on the NP-hardness of the factorization problem. RSA is very famous because it is quite simple to understand and imple-

ment, therefore it is widespread in many applications. However, to obtain high levels of security it requires very long keys (the length of an RSA key is given by the number of bits of the modulus), which make the private-key operations quite inefficient. For example, TOP-SECRET security requires a 7680 bits modulus.

4.1 Logical Representation of the Digital Envelope

How can we encrypt a plaintext with asymmetric encryption? A straightforward way to do that is to encrypt the whole plaintext with the public key of the recipient. However, this is very inefficient, because asymmetric encryption is extremely slow compared to symmetric one. A better solution is to encrypt the whole message with a randomly generated symmetric key, and then encrypt only the symmetric key with the public key. Fig. 9 shows this technique, called *Digital Envelope*. In the figure, *k* is a symmetric key chosen at random from a PRNG, $E_k()$ is a symmetric encryption function that uses *k*, *kpub* is an asymmetric public key, and $E_{kpub}()$ is an asymmetric encryption function that uses *kpub*. The digital envelope technique encrypts a message in such a way that only who knows a particular private key can decrypt it. In contrast to symmetric encryption, no pre-shared secret is needed.

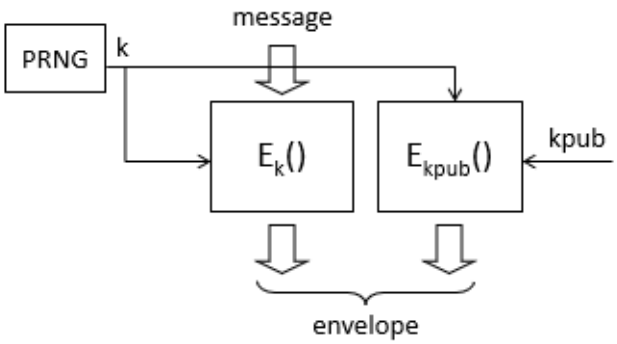


Figure 9: Logical representation of the digital envelope technique.

When you want to send the same message to multiple recipients, you can simply encrypt the same symmetric key with the public keys of each intended recipient, as shown in Fig. 10.

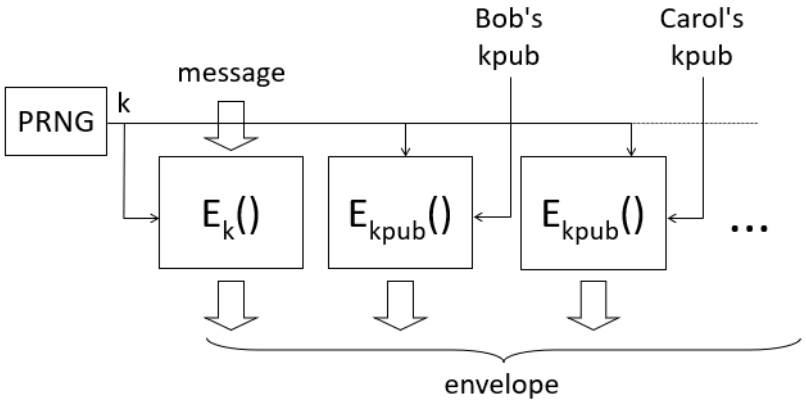


Figure 10: Logical representation of the multi-addressed digital envelope technique.

4.2 OpenSSL Asymmetric Encryption

The following are the OpenSSL command-line tools to create a private key and to extract the public key from a private key. All the keys are saved in PEM-format files. In case a passphrase is used to encrypt the private key, such a passphrase is asked to the user when generating the private key, and asked again when extracting the public key. A public key is never encrypted with passphrases (it's *public*!).

`openssl genrsa -out rsa_privkey.pem 3072`
Generates an RSA private key with modulus over 3072 bits, and stores it in the file named `rsa_privkey.pem`.

`openssl genrsa -aes128 -out rsa_privkey.pem 3072`
Same, but the generated private key is encrypted with a password, using `AES_128` as encryption algorithm. The password is taken *blindly* (i.e., it won't echo on the prompt) from `stdin`.

`openssl rsa -pubout -in rsa_privkey.pem -out rsa_pubkey.pem`
Extracts the public key from an RSA private key file named `rsa_privkey.pem` and saves it in `rsa_pubkey.pem`.

4.2.1 Managing Asymmetric Keys with OpenSSL

In OpenSSL, an `EVP_PKEY` data structure represents a private or a public key. These API functions allocate, deallocate, and, more in general, manage an `EVP_PKEY` data structure.

`EVP_PKEY` (data structure)
It represents a public key or a private key of any supported asymmetric cryptosystem.

```
EVP_PKEY* EVP_PKEY_new();
```
Allocates an EVP_PKEY.

```
void EVP_PKEY_free(EVP_PKEY* key);
```
Deallocates an EVP_PKEY.

```
int EVP_PKEY_size(EVP_PKEY* key);
```
Returns the maximum size of an encrypted symmetric key (useful for allocating buffers).

```
#include <openssl/pem.h>
```
Library containing API functions for reading/writing from/to PEM-format files.

```
EVP_PKEY* PEM_read_PrivateKey(FILE* fp, NULL, NULL, pwd);
```
This function allocates an `EVP_PKEY` structure, then loads a private key from a PEM file.

- `fp`: File descriptor, e.g., opened with `fopen()`.
- `pwd`: It can be an hardcoded string (e.g., "p4$$w0rD"), a variable, or `NULL`. If the private key file is password protected and `NULL` is passed as argument, the API will blindly request the password from `stdin`. Therefore, `NULL` is the recommended option.
- `RETURNS` the `EVP_PKEY` structure (or `NULL` if error).

`EVP_PKEY* PEM_read_PUBKEY(FILE* fp, NULL, NULL, NULL);`
Allocates a public key and loads its value from the PEM file whose file descriptor is pointed by `fp`.

- `fp`: File descriptor, e.g., opened with `fopen()`;
- `RETURNS` a pointer to the newly allocated `EVP_PKEY` structure (or NULL if error).

Do you remember how OpenSSL contexts work? *Creation, Initialization, Update, Finalize, Deallocation.* We have already seen how to create a context in Sec. 3, in the following we'll see how to apply the same context to asymmetric operations.

WARNING! All the OpenSSL API functions for *digital envelope* support ONLY RSA cryptosystem. Although digital envelope technique based on EC is technically possible (cfr. the standard ECIES: Elliptic-Curve Integrated Encryption Scheme), it is NOT implemented by OpenSSL (version 1.1.1).

`int EVP_SealInit(EVP_CIPHER_CTX* ctx, const EVP_CIPHER* type, unsigned char** ek, int* ekl, unsigned char* iv, EVP_PKEY** pubk, int npubk);`
Initializes a context for (possibly multi-addressed) digital envelope. It also generates the symmetric key for encryption and a random IV.

- `ctx` (input/output): The `EVP_CIPHER_CTX*` context already created.
- `type` (input): The **symmetric** cipher to use in the digital envelope.
- `ek` (output): Preallocated (array of) output buffer(s) that will store the encrypted symmetric key(s). In case of multi-addressed envelope, the other encrypted symmetric keys are stored in ek[1], ek[2], etc. The buffer `ek[0]` must accommodate at least `EVP_PKEY_size(pubk[0])` bytes.
- `ekl` (output): Preallocated (array of) integer(s) that will store the length(s) of the encrypted symmetric key(s).
- `iv` (output): Preallocated output buffer that will store the generated initialization vector.
- `pubk` (input): The (array of) public key(s). In case of multi-addressed envelope, the other public keys are pubk[1], pubk[2], etc.
- `npubk` (input): The number of public keys (1 if single-address envelope, >1 if multi-address envelope).
- `RETURNS` 0 on error, non-0 on success.

4.3 Digital Envelope Essentials

```
#include <openssl/evp.h>
#include <openssl/pem.h>
int main() {
    char msg[] = "Lorem ipsum dolor sit amet.";
    unsigned char* encrypted_key, *iv;
    int encrypted_key_len;
    FILE* pubkey_file = fopen("pubkey_filename", "r");
    EVP_PKEY* pubkey = PEM_read_PUBKEY(pubkey_file, NULL, NULL, NULL);
    fclose(pubkey_file);
    encrypted_key = malloc(EVP_PKEY_size(pubkey));
    unsigned char* ciphertext = malloc(sizeof(msg) + 16);
    int outlen, cipherlen;
    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    iv = (unsigned char*) malloc(EVP_CIPHER_iv_length(EVP_aes_128_cbc()));
    ret = EVP_SealInit(ctx, EVP_aes_128_cbc(),
                &encrypted_key, &encrypted_key_len, iv, &pubkey, 1);
    if(ret == 0) { /* handle error */ }
    EVP_SealUpdate(ctx, ciphertext, &outlen, (unsigned char*)msg, sizeof(msg));
    cipherlen = outlen;
    EVP_SealFinal(ctx, ciphertext + cipherlen, &outlen);
    cipherlen += outlen;
    EVP_CIPHER_CTX_free(ctx);
    return 0;
}

```

5

SECTION

Diffie-Hellman Key Exchange

Diffie-Hellman protocol (or Diffie-Hellman-Merkle protocol) allows two parties to establish a shared secret over a non-confidential channel without using any *pre-shared* secret. It was published by Whitfield Diffie and Martin Hellman in 1976, inspired by the work of Ralph Merkle. It is one of the first examples of public-key cryptography. As shown in Fig. 11, in Diffie-Hellman (DH), first the parties agree on two parameters: p (the modulus, a large prime number) and g (the generator, usually 2 or 5). Then, they both generate a pair of quantities: a private key and a public key. Alice generates a (private key) and $Y_a = g^a mod p$ (public key). Bob generates b (private key) and $Y_b = g^b mod p$ (public key). They both send their public keys to the other party. Finally, Alice and Bob independently retrieve the secret, by computing $K_{ab} = Y_b^a mod p$ (Alice) and $K_{ab} = Y_a^b mod p$ (Bob). Diffie-Hellman resists to an eavesdropper adversary who wants to discover the shared secret. The security is based on the hardness of the discrete logarithm problem.

The "pure" version of Diffie-Hellman, also called "Anonymous" Diffie-Hellman, does not provide for the authentication of the parties. This means that an active adversary could pretend to be one of the parties. Moreover, an adversary could mount a man-in-the-middle attack, performing two distinct Diffie-Hellman protocols, one with Alice and another with Bob. She could then transparently decrypt and re-encrypt the messages that Alice and Bob send to each other, thus intercepting the whole communication. These attacks can be avoided by using authenticated versions of Diffie-Hellman, which are currently used in state-of-the-art security protocols (TLS, IPsec, etc.).

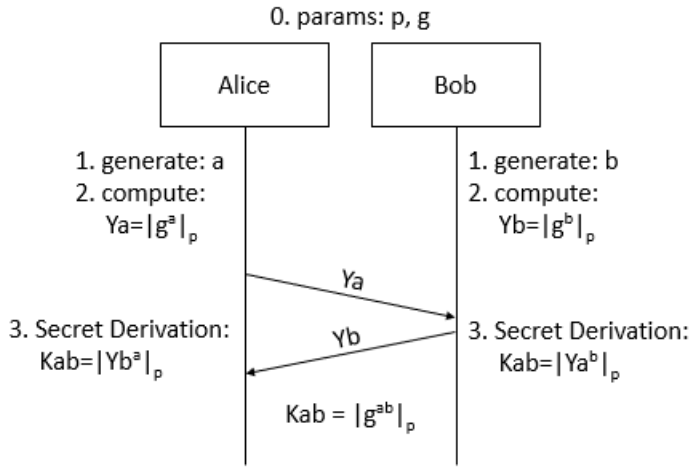


Figure 11: Logical representation of the Diffie-Hellman Key Exchange.

5.1 Diffie-Hellman in OpenSSL

5.1.1 Parameter Generation and Parameter Management

OpenSSL provides a command-line tool for generating **custom Diffie-Hellman parameters**:

`openssl dhparam -C 2048`

Command-line tool that creates Diffie-Hellman parameters p,g with public keys on 2048 bits, and prints on stdout the source code of a C function called `get_dh2048()` that allocates and returns a low-level DH structure containing such parameters. In the following code, you can see an example of the generated function.

```
static DH *get_dh2048(void){
    static unsigned char dhp_2048[] = {
        0x91, 0xB7, 0x38, 0x6F, 0x27, 0xC1, 0x91, 0xDC, 0xD9, 0x86,
        0x23, 0x8A, 0xF0, 0x26, 0xB8, 0xD2, 0x84, 0x19, 0x94, 0x43,
        0xD2, 0xE4, 0x55, 0x55, 0xC7, 0xE4, 0xCD, 0x2F, 0x12, 0xF7,
        0xDE, 0xE0, 0xAB, 0x6E, 0xC4, 0x15, 0xB0, 0xB1, 0x8D, 0x59,
        0xCD, 0xF8, 0xD6, 0xE9, 0xF3, 0x73, 0x6F, 0x1C, 0x1E, 0x9E,
        0x1F, 0x1D, 0xEE, 0x4B, 0x09, 0xCA, 0x5C, 0xCA, 0x49, 0xE7,
        0xE9, 0x8A, 0x9A, 0x16, 0x63, 0x91, 0x43, 0x12, 0x12, 0x20,
        0xFA, 0x27, 0x78, 0xC3, 0xA4, 0x38, 0x4B, 0xA3, 0x17, 0x76,
        0xE5, 0xE6, 0x55, 0x37, 0x77, 0x44, 0x0F, 0x3C, 0x4A, 0x74,
        0x75, 0x83, 0xB0, 0x4B, 0x57, 0xF1, 0x30, 0xAE, 0x3B, 0xD7,
        0xDA, 0xC9, 0x70, 0x31, 0x4B, 0xBB, 0xCA, 0x94, 0x35, 0x92,
        0xE2, 0x0F, 0x4F, 0x67, 0x05, 0x72, 0xB5, 0xC2, 0x7B, 0x0A,
        0xBC, 0x12, 0x5B, 0x8B, 0x62, 0x29, 0x73, 0x5C, 0x19, 0x86,
        0xB2, 0x7C, 0x8D, 0x55, 0x9D, 0xC5, 0x6D, 0x5E, 0x09, 0xDA,
        0x7C, 0x86, 0x22, 0xDC, 0x73, 0x83, 0xCF, 0x29, 0xEB, 0x05,
        0xEC, 0x97, 0xE5, 0x64, 0x02, 0x39, 0x5B, 0xDD, 0x24, 0x2E,
```

```
        0x93, 0x36, 0x46, 0xE8, 0x5F, 0xA3, 0xB5, 0x77, 0xE2, 0xD0,
        0x76, 0x4A, 0x67, 0xCB, 0x9D, 0x81, 0x73, 0xB8, 0x49, 0x3D,
        0xF4, 0xA7, 0xD0, 0xF3, 0x58, 0x34, 0xF3, 0xE6, 0x7D, 0xDC,
        0xE2, 0x02, 0x0C, 0x8F, 0x17, 0x9F, 0x58, 0xAF, 0xE2, 0xBF,
        0x12, 0x91, 0xF4, 0x05, 0xE8, 0x86, 0x24, 0x8C, 0x35, 0x6E,
        0xBF, 0x99, 0xD8, 0xF2, 0xB3, 0xF3, 0xDF, 0x0A, 0x0D, 0x7A,
        0x65, 0x5C, 0x33, 0x13, 0xB8, 0x3B, 0xE4, 0x2D, 0x50, 0x91,
        0x5F, 0x6F, 0x41, 0xB7, 0x18, 0xFE, 0xA4, 0xDB, 0xB8, 0x0E,
        0xFE, 0x47, 0x08, 0x26, 0xB2, 0x4B, 0x57, 0xEC, 0x52, 0x61,
        0xFF, 0xD7, 0x95, 0xE8, 0xDF, 0x5B
    };
    static unsigned char dhg_2048[] = {
        0x02
    };
    DH *dh = DH_new();
    BIGNUM *p, *g;

    if (dh == NULL)
        return NULL;
    p = BN_bin2bn(dhp_2048, sizeof(dhp_2048), NULL);
    g = BN_bin2bn(dhg_2048, sizeof(dhg_2048), NULL);
    if (p == NULL || g == NULL
            || !DH_set0_pqg(dh, p, NULL, g)) {
        DH_free(dh);
        BN_free(p);
        BN_free(g);
        return NULL;
    }
    return dh;
}
```

Instead, if you want to use **standard parameters** for the DH protocol, you can use the following APIs (recommended option):

`DH* DH_get_1024_160();`
Returns preallocated (low-level) object containing DH parameters standardized by IETF RFC 5114, with public keys on 1024 bits and private keys on 160 bits (80 bits of equivalent security strength).

`DH* DH_get_2048_224();`
Same as before, but public keys are on 2048 bits and private keys on 224 bits (112 bits security strength).

`EVP_PKEY_set1_DH(EVP_PKEY* dh_params, DH* low_level_dh);`

Copies the low-level Diffie-Hellman parameter `low_level_dh` into the (previously allocated) high-level Diffie-Hellman parameters `dh_params`. Returns 1 on success.

`DH_free(DH*);`
Deallocates a DH structure.

5.1.2 Custom Parameters Example

In the following you can see a setup example using custom DH parameters generated through command-line.

```
1 // From command line: openssl dhparam -C 2048
2
3 static DH *get_dh2048(void){/*...command-line generated code...*/}
4 /* ... */
5 EVP_PKEY* dh_params;
6 DH* tmp = get_dh2048();
7 dh_params = EVP_PKEY_new();
8 EVP_PKEY_set1_DH(dh_params, tmp);
9 DH_free(tmp);
```

5.1.3 Standard Parameters Example

In the following you can see a setup example using standard DH parameters provided by OpenSSL. This is the recommended approach.

```
1 EVP_PKEY* dh_params;
2
3 dh_params = EVP_PKEY_new();
4 EVP_PKEY_set1_DH(dh_params, DH_get_2048_224());
```

5.1.4 Diffie-Hellman Key Generation

`EVP_PKEY` (data structure)
The same structure may contain either: (i) a set of Diffie-Hellman parameters; or (ii) a pair of Diffie-Hellman public/private keys.

`EVP_PKEY_CTX` (data structure)
A context for Diffie-Hellman public-key operations.

`EVP_PKEY_CTX* EVP_PKEY_CTX_new(EVP_PKEY* pkey, NULL);`
Allocates a context for public-key operations according to the content of the parameter pkey: (i) if pkey represents DH parameters, the operation is a key generation; (ii) if pkey represents DH private key, the operation is a secret derivation.

`EVP_PKEY_CTX_free(EVP_PKEY_CTX* pkey);`

Deallocates the context for public-key operations.

`int EVP_PKEY_keygen_init(ctx);`
Initializes a context for Diffie-Hellman key generation. Returns 1 on success.

`int EVP_PKEY_keygen(ctx, EVP_PKEY** pkey);`
Allocates memory for and generates a Diffie-Hellman private key (which also contains the public key) and stores it in **pkey. Remember to free **pkey right after the key exchange! Returns 1 on success.

`EVP_PKEY_free(EVP_PKEY* pkey);`
Deallocates a PKEY structure.

5.1.5 Diffie-Hellman Key Generation Example

The following code snippet represent the first step to realize the Diffie-Hellman key exchange, both with discrete-log and with elliptic-curve mathematics. We assume the Diffie-Hellman parameters to be already generated and stored in a structure. In OpenSSL, the `EVP_PKEY` structure represents a generic quantity for asymmetric cryptography, for example private and public RSA keys. In this example, it represents Diffie-Hellman parameters. The `EVP_PKEY_CTX` structure represents a context of a generic public-key algorithm, in this case an algorithm to generate a pair of public/private keys. To generate a new key pair from the parameters, we have to use the `EVP_PKEY_keygen_init()` and the `EVP_PKEY_keygen()` functions.

```
1  #include <openssl/evp.h>
2  int main() {
3    EVP_PKEY* dh_params;
4
5    /* ... Load Diffie-Hellman parameters in dh_params */
6
7    /* Generation of private/public key pair */
8    EVP_PKEY_CTX* ctx = EVP_PKEY_CTX_new(dh_params, NULL);
9    EVP_PKEY* my_prvkey = NULL;
10   EVP_PKEY_keygen_init(ctx);
11   EVP_PKEY_keygen(ctx, &my_prvkey);
12
13   /* ... Send the public key inside my_prvkey to the peer */
```

5.1.6 Save/Load DH Public Keys in PEM Format

`int PEM_write_PUBKEY(FILE* fp, EVP_PKEY* pkey);`
Saves a DH public key on a PEM file. Usually this function is used to extract and convert the user's own public key into PEM format, so that it is ready to be sent to the DH peer.

- `fp` (output): File where to write (file pointer). It should have been previously opened with fopen().
- *pkey* (input): If pkey represents a DH private key, it extracts the public one and saves it.
- `returns`: 1 on success.

`EVP_PKEY* PEM_read_PUBKEY(FILE* fp, NULL, NULL, NULL);`
Allocates a DH public key and loads it from a PEM file. Usually this function is used to load in memory the peer's DH public key received in PEM format.

- `fp` (input): File where to read (file pointer). It should have been previously opened with fopen().
- `returns`: the `EVP_PKEY` structure (or NULL if error).

5.1.7 Diffie-Hellman Key Derivation

`int EVP_PKEY_derive_init(ctx);`
Initializes a context for Diffie-Hellman secret derivation. Returns 1 on success.

`int EVP_PKEY_derive_set_peer(ctx, EVP_PKEY* peer_pubkey);`
Set the peer's public key for Diffie-Hellman secret derivation. Returns 1 on success.

`int EVP_PKEY_derive(ctx, NULL, size_t* secretlen);`
Returns the maximum size of the Diffie-Hellman shared secret to be derived in `*secretlen`. Returns 1 on success.

`int EVP_PKEY_derive(ctx, unsigned char* secret, size_t* secretlen);`
Derives a Diffie-Hellman shared secret in `secret`, and returns its size in `*secretlen`. Returns 1 on success.

5.1.8 Diffie-Hellman Key Derivation Example

The second step to perform the DH key exchange is to derive the shared secret from the user's private key and the peer's public key. We use the `EVP_PKEY_derive_init()`, `EVP_PKEY_derive_set_peer()` and `EVP_PKEY_derive()` functions to do that. Note that we call the `EVP_PKEY_derive()` function twice: the first time to get the maximum size of the shared secret to allocate enough space for it, and the second time to actually compute the shared secret. Note that you should never use a truncation of the shared secret as a key for symmetric encryption, because it has not the necessary entropy. It is always preferable to use a hash of the shared secret.

```
1 EVP_PKEY* peer_pubkey;
2   /* ... Retrieve the public key of the peer and store it in peer_pubkey */
3
4   /* Initializing shared secret derivation context */
5   EVP_PKEY_CTX* ctx_drv = EVP_PKEY_CTX_new(my_prvkey, NULL);
```

```
  EVP_PKEY_derive_init(ctx_drv);
  EVP_PKEY_derive_set_peer(drv_ctx, peer_pubkey);
  unsigned char* secret;

  /* Retrieving shared secret's length */
  size_t secretlen;
  EVP_PKEY_derive(drv_ctx, NULL, &secretlen);

  /* Deriving shared secret */
  secret = (unsigned char*)malloc(secretlen);
  EVP_PKEY_derive(drv_ctx, secret, &secretlen);
```

WARNING! Never use directly the shared secret as a key for symmetric encryption! You should hash the derived secret, and use the resulting digest as a symmetric key. You will learn how to compute digests in the next Chapter.

5.1.9 Elliptic-Curve Diffie-Hellman Parameters Example

The following code snippet retrieves Elliptic-Curve Diffie-Hellman (ECDH) parameters relative to the standard prime256v1 elliptic curve, providing for 128 bits of effective security strength. The operations seen in this snippet are an alternative to those in Section 5.1.5. From Section 5.1.6 to Section 5.1.8, all the operations are exactly the same.

```
EVP_PKEY* dh_params = NULL;
EVP_PKEY_CTX* pctx;
pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_EC, NULL);
EVP_PKEY_paramgen_init(pctx);
EVP_PKEY_CTX_set_ec_paramgen_curve_nid(
 pctx,   NID_X9_62_prime256v1);
EVP_PKEY_paramgen(pctx, &dh_params);
EVP_PKEY_CTX_free(pctx);
```

SECTION 6

Hash, HMAC, Authenticated Encryption

6.1 Simple Hash Functions

The logical representation of a simple hash algorithm (Fig.12) is a function, taking a variable-sized message as input, and returning a fixed-sized *digest* as output. Implementing digest creation in this way is not efficient. Indeed, if the message is big, we have to maintain in memory a big quantity of data at once. The majority of cryptographic libraries uses instead incremental functions, which update a hashing context step-by-step. This is done in higher-level languages as well, for example Java, C#, Python. A realistic pseudo-code that computes an hash value is the

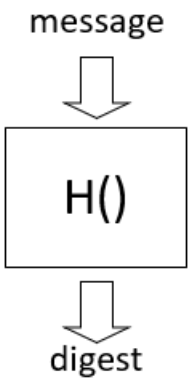


Figure 12: Logical Representation of an Hash Function.

following:

```
1. md_ctx = context_initialize(hash_algorithm);
2. cycle:
```

```
   context_update(md_ctx, message_fragment);
end
3. digest = context_finalize(md_ctx);
```

Instead, a realistic pseudo-code that verifies an hash value is the following:

```
1. received_digest = /*retrieves digest from the message*/
2. md_ctx = context_initialize(hash_algorithm);
3. cycle:
   context_update(md_ctx, message_fragment);
end
4. computed_digest = context_finalize(md_ctx);
compare(computed_digest, received_digest);
```

Clearly, this pseudo-code is very similar to the encryption/decryption procedures seen in previous chapters. However, please note that during the context update no data is returned, as the output of the hash function is of fixed size and created entirely at the end.

6.1.1 Hash Functions in OpenSSL

At the beginning of the program, you have to include the library `<openssl/evp.h>`, which contains the high-level API functions useful for computing and verifying digests.

```
#Data Structure
    EVP_MD_CTX
#Context creation:
    EVP_MD_CTX* md_ctx;
    md_ctx = EVP_MD_CTX_new();
#Hashing:
    EVP_DigestInit(md_ctx, <Hash_type>);
    EVP_DigestUpdate(md_ctx, inbuf, inlen);
    EVP_DigestFinal(md_ctx, outbuf, &outlen);
#Context destruction:
    EVP_MD_CTX_free(md_ctx);
#<Hash_type> Values:
    EVP_MD* EVP_md5();
    EVP_MD* EVP_sha1();
    EVP_MD* EVP_sha256();
```

The most common hash algorithms used in OpenSSL are `MD5`, `SHA-1`, `SHA-2`. It is recommended to use `SHA-256`, since the other algorithms are obsolete or will become obsolete soon. `MD5` (*Message Digest 5*) is an obsolete algorithm, completely broken from the security point of view. In 2013, a research showed how to find colliding texts (birthday attack) for `MD5` in less than 1 second of processing time on a common PC. Preimage attacks are known too, even if they are

not realized in practice yet. `SHA-1` (*Secure Hash Algorithm 1*) offers medium security. Theoretical attacks are known, and the first attack developed in practice was announced in February 2017. However, this attack is difficult to realize: it requires 6,500 years of CPU computation to complete the first phase of the attack, and 110 years of GPU computation to complete the second phase. `SHA-256` (part of the SHA-2 family) offers good security. No practical or theoretical attacks are known.

`EVP_MD_size(<Hash_type>);` (Utility Function)
Returns the size (in bytes) of any digest computed with said hash function (remember, the output is over a fixed amount of bytes).

`int CRYPTO_memcmp(computed_digest, received_digest, digest_len);`
Compares two portions of memory in constant time. Returns 0 if they are equal. This function is defined in `<openssl/crypto.h>`.

WARNING! It is NOT safe to use the standard `memcmp()` function to compare two digests, because it makes the system vulnerable to *timing attacks*. Indeed, the runtime of *memcmp()* depends on the inputs: if they differ in the first bytes, the runtime will be short; if they differ in the last bytes only, it will be long. An adversary can make the system check several (wrong) digests. By measuring the runtime each time, the adversary can learn how many initial bytes are correct. In this way, the complexity of guessing the correct digest is linear with the length of the digest, instead of exponential. Instead, *CRYPTO_memcmp()* has a constant runtime, therefore it is recommended to check digests.

6.1.2 Hash Computation Example

The following code snippet realizes a simple hash of a static text with SHA-256. Note that the output buffer "digest" is allocated with the utility function `EVP_MD_size(<hashing_algorithm>)`, so that the programmer does not have to check the size of the output. At the end, in the «digest» buffer, there will be the digest. You must always deallocate the context!

```
1  #include <openssl/evp.h>
2  int main() {
3    char msg[] = "Lorem ipsum dolor sit amet.";
4    unsigned char* digest;
5    int digestlen;
6    EVP_MD_CTX* ctx;
7
8    /* Buffer allocation for the digest */
9    digest = (unsigned char*)malloc(EVP_MD_size(EVP_sha256()));
10
11   /* Context allocation */
12   ctx = EVP_MD_CTX_new();
13
14   /* Hashing (initialization + single update + finalization */
```

```
  EVP_DigestInit(ctx, EVP_sha256());
  EVP_DigestUpdate(ctx, (unsigned char*)msg, sizeof(msg));
  EVP_DigestFinal(ctx, digest, &digestlen);

 /* Context deallocation */
  EVP_MD_CTX_free(ctx);

  /* ... Print digest on screen in hexadecimal digits ... */
  return 0;
}
```

6.1.3 Hash Verification Example

The following code snippet realizes a digest verification. Differently from Encrypt/Decrypt and Seal/Open, in case of an hash both the sender and the receiver use the same APIs. Clearly, the receiver needs also to invoke the `CRYPTO_memp` function to «verify» the correctness of the hash.

```
#include <openssl/evp.h>
int main() {
  /* ...receives the message and the message digest... */
  unsigned char* digest;
  int digestlen;
  EVP_MD_CTX* ctx;

  /* Buffer allocation for the digest */
  digest = (unsigned char*)malloc(EVP_MD_size(EVP_sha256()));

  /* Context allocation */
  ctx = EVP_MD_CTX_new();

  /* Hashing (initialization + single update + finalization */
  EVP_DigestInit(ctx, EVP_sha256());
  EVP_DigestUpdate(ctx, (unsigned char*)received_msg, sizeof(received_msg));
  EVP_DigestFinal(ctx, digest, &digestlen);

  /* Context deallocation */
  EVP_MD_CTX_free(ctx);

  if(CRYPTO_memcmp(digest, received_digest, EVP_MD_size(EVP_sha256())) == 0)
    /* Digest OK */
  else
    /* Invalid digest */

  return 0;
```

```
28 }
```

6.2 Keyed Hash Functions in OpenSSL

Within security applications, *keyed hash algorithms* (HMAC) are more useful than "pure" ones, because they are used for authenticating communications. The logical representation of a keyed hash algorithm is a function taking a key and a variable-sized message as input, and returning a fixed-size digest as output, as shown in Fig. 13. The majority of cryptographic libraries use

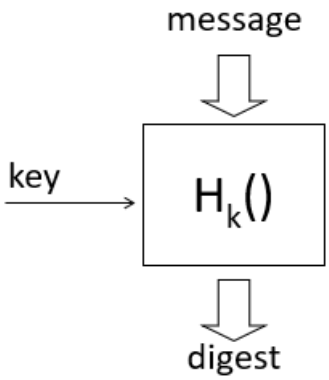


Figure 13: Logical Representation of a keyed hash function.

incremental functions for keyed hash algorithms as well. Note that HMAC algorithms do not impose constraints on the key length. However, keys of the same size of the digests are implicitly recommended by the HMAC RFC (rfc2104). This is because if the key is shorter than the digest, then it will be easier to guess the key, thus the security is weaker. Otherwise, a key longer than the digest is useless, since it makes more convenient to guess directly the digest. Every hash algorithm can be extended to be keyed. There are HMAC-MD5, HMAC-SHA1, HMAC-SHA256, etc. The HMAC digests have the same size of the simple hash algorithm digests.
At the beginning of the program, you have to include the library `<openssl/hmac.h>`, which contains the high-level API functions useful for computing and verifying keyed digests.

```
#Data Structure
    HMAC_CTX*
#Context creation:
    HMAC_CTX* hmac_ctx;
    hmac_ctx = HMAC_CTX_new();
#Keyed-hashing:
    HMAC_Init_ex(hmac_ctx, key, keylen, <Hash_type>);
    HMAC_Update(hmac_ctx, inbuf, inlen);
    HMAC_Final(hmac_ctx, outbuf, &outlen);
#Context destruction:
    HMAC_CTX_free(hmac_ctx);
```

The `HMAC` computation and verification follow the the same structure as the ones shown in Section 6.1.2 and Section 6.1.3, respectively.

There is also a function to compute an HMAC on-the-fly, without initializing and destroying the context. This function is useful to simplify the code when the message to be authenticated has a short and fixed size (for example a nonce).

```
HMAC(<Hash_type>, key, keylen, inbuf, inlen, outbuf, &outlen);
```

WARNING! HMAC are often used to provide *authenticity and integrity* to messages that only provide for *confidentiality*, as symmetrically encrypted messages do. For example, one can use `AES` plus `HMAC` to secure a communication. Those two mechanisms (encryption and hashing) need each a key, but, in theory, one could use the same key for both. Be aware that is preferable to use **different keys** for encryption and authentication. Indeed, if one key is compromised, then only one mechanism is compromised (damage limitation). Moreover, if a mechanism is not used anymore, its key must be destroyed to respect the principle of the *end of the cryptoperiod*. However, this cannot be done if the key is used by another mechanism!

6.3 Authenticated Encryption

Often, data must be both encrypted and authenticated with symmetric-key encryption. One way is to mount together two independent mechanisms: one for encryption (e.g., AES-CBC) and one for authentication (e.g., HMAC). However, if the two mechanisms are not properly mounted, this approach can lead to vulnerabilities. An example is the padding oracle attack, which affects the CBC encryption modes mounted with some MAC mechanism. This is the reason why *Authenticated Encryption with Associated Data* (AEAD) mechanisms have been developed. As shown in Fig. 14, AEAD can be represented as a function which takes three inputs: a key, a plaintext, and an additional authenticated data (AAD); moreover, it returns two outputs: a ciphertext, and a tag. The ciphertext is the encrypted plaintext. The tag acts as a MAC and assures the authenticity of the plaintext together with the AAD. The same key is used for encrypting and authenticating.

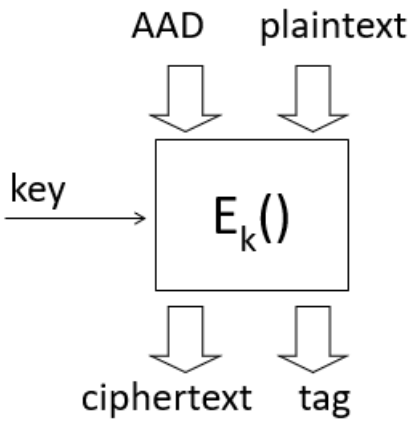


Figure 14: Logical Representation of Authenticated Encryption mechanism.

The presence of the AAD means that the authenticity mechanism covers a superset of what is covered by the encryption mechanism. Fig. 15 shows the result of an authenticated encryption. This is useful in many common situations, for example when a message must be *authenticated* and *encrypted* in such a way that some of the header is left in the clear. Such an in-the-clear header may contain information needed for decryption (a very clear example is the initialization

vector!). Common authenticated encryption modes implemented in OpenSSL are `GCM` (Galois-Counter Mode), `CCM` (Counter and CBC-MAC Mode), and `OCB` (Offset CodeBook mode).

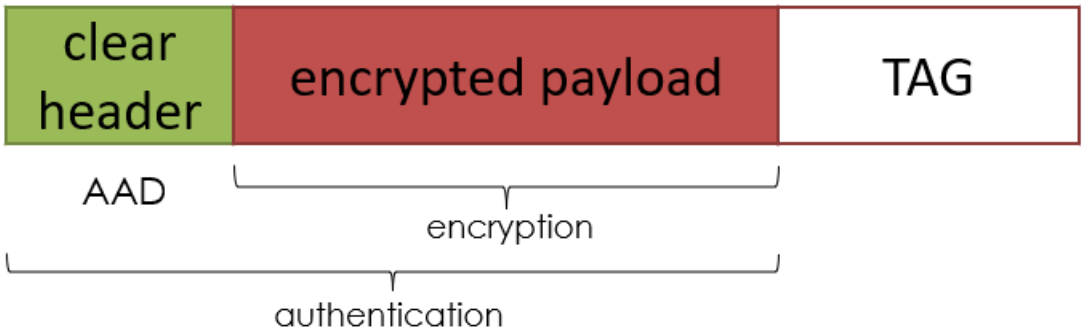


Figure 15: Authenticated Encryption message representation. In green authenticated field, in red authenticated and encrypted fields

A realistic pseudo-code that performs an authenticated encryption is the following:

```
1. ctx = context_initialize(encrypt, cipher, mode, key, iv, etc...);
2. cycle:
   context_update(ctx, AAD_fragment);
end
3. cycle:
   ciphertext_fragment = context_update(ctx, plaintext_fragment);
end
4. ciphertext_fragment = context_finalize(ctx);
5. tag = retrieve_tag(ctx);
```

We must first initialize the context and giving the various parameters (i.e., cipher, mode, key, iv). Then, we perform a *first* cycle, giving a series of AAD fragments (context update). Please note that, similarly to the hash context update, the api does not return any ciphertext fragment. After this, we perform a *second* cycle, with the plaintext fragments (as we have seen in Sec. 3). Finally, we finalize the context, retrieving the last ciphertext fragment, and we retrieve the computed tag.
A realistic pseudo-code that performs an authenticated decryption is the following:

```
1. ctx = context_initialize(decrypt, cipher, mode, key, iv, etc...);
2. cycle:
   context_update(ctx, AAD_fragment);
end
3. cycle:
   plaintext_fragment = context_update(ctx, ciphertext_fragment);
end
4. set_received_tag(ctx, received_tag);
5. plaintext_fragment = context_finalize(ctx);
```

Note that the context must be informed about the received tag before executing the context finalization. Such a context finalization returns an error in case the received tag does not match with the computed one.

6.3.1 Authentication Encryption in OpenSSL

OpenSSL provides AEAD APIs for, among others, the following algorithms:
`EVP_aes_128_gcm();`
`EVP_aes_128_ccm();`
`EVP_aes_128_ocb();`
The following APIs:

```
EVP_EncryptInit(ctx, Algorithm&Mode, key, iv);
EVP_DecryptInit(ctx, Algorithm&Mode, key, iv);
EVP_EncryptUpdate(ctx, outbuf, &outlen, inbuf, inlen);
EVP_DecryptUpdate(ctx, outbuf, &outlen, inbuf, inlen);
EVP_EncryptFinal(ctx, outbuf, &outlen);
EVP_DecryptFinal(ctx, outbuf, &outlen);
```

behave exactly as described in Sec. 3.1. OpenSSL will initialize `ctx` accordingly when `Algorithm&Mode` belongs to the authenticated encryption set listed above, with some exceptions.

`EVP_(En|De)cryptUpdate(ctx, NULL, outlen, inbuf, inlen);`
To specify additional authenticated data (AAD), a call to `EVP_EncryptUpdate()` or `EVP_DecryptUpdate()` should be made with the output parameter `outbuf` set to `NULL`.

`int EVP_DecryptFinal(ctx, outbuf, outlen);`
During decryption, the return value of *EVP_DecryptFinal()* indicates whether the operation was successful (`return 1`) or not (`return 0`). If the decryption failed, the authentication operation has failed too, and any output data **MUST NOT** be used, as it is corrupted.

Ok, so, how do we manage the `tag`? OpenSSL provides for an API that helps the programmer to extract and set the `tag` during encryption and decryption.
`EVP_CIPHER_CTX_ctrl(ctx, FLAG, taglen, tag);`

- `ctx`: the context of encryption or decryption;
- `FLAG`: this field can assume two distinct values:
  - `EVP_CTRL_AEAD_GET_TAG`: it is used to *extract* the tag from the context. In this case the programmer must call the function **after** the `EVP_EncryptFinal();`
  - `EVP_CTRL_AEAD_SET_TAG`: it is used to *set* the *received* tag in the context. In this case the programmer must call the function **before** the `EVP_DecryptFinal();`
- `taglen`: by default, it is 16 bytes for `AES_GCM` and `AES_OCB`, while it is 12 bytes for `AES_CCM`;

- `tag`: this value assume a different role depending on the value of `FLAG`:
  - `EVP_CTRL_AEAD_GET_TAG`: in this case `tag` is a *preallocated* buffer that will contain the extracted tag;
  - `EVP_CTRL_AEAD_SET_TAG`: in this case `tag` is a buffer containing the tag *received* from the sender (that will be compared to the one calculated by the `EVP_DecryptFinal()` API);

6.3.2 Authenticated Encryption vs. Encryption + Authentication

Table 1 shows the pros and cons in adopting an Authenticated Encryption mechanism (e.g., `AES_GCM`, or adopting two separate Encryption mechanism (e.g., `AES_CBC`) plus an Authentication mechanism (e.g., `SHA-2 HMAC`).

Authenticated Encryption	Encryption + Authentication
✓Less error prone (mounting, key management...)	✓if a key is compromised, the other mechanism is still safe
✓Code easier to implement and to maintain	✓Flexible, either one of the mechanism can be changed without changing the other
✓Faster	✓When not required, you may not perform one of the two mechanism
×If the key is compromised, both encryption and authentication are lost.	×The code is harder to implement and to maintain.
×Must always compute and communicate the tag, even when not necessary	×Inexperienced developers can introduce vulnerabilities

Table 1: Pros and cons of two different strategies to achieve *Confidentiality, Authenticity* and *Integrity* of a communication.

SECTION 7

Digital Signature

The digital signature technique (Fig. 16) authenticates a message in such a way that everyone can verify its authenticity. To do so, it leverages two *Asymmetric Keys*, the first called *private key*, and the second called *public key*. Usually, we do not authenticate the message itself, but the digest of the message, relying on the collision-resistance property of the hash algorithm. The digest is authenticated by signing it with the private key of the authenticating entity.

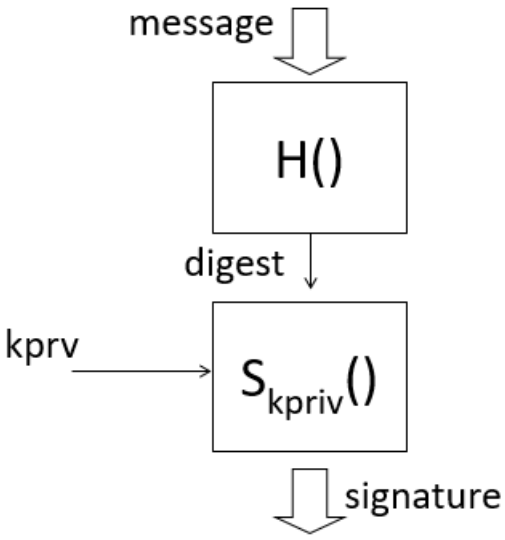


Figure 16: Logical Representation of a Digital Signature.

The verification procedure (Fig. 17) extracts the digest from the signature by means of the public key of the authenticating entity. Then, computes the digest from the message received, then

it compares the computed digest with the one previously extracted. If they are the same, the verification process is successful.

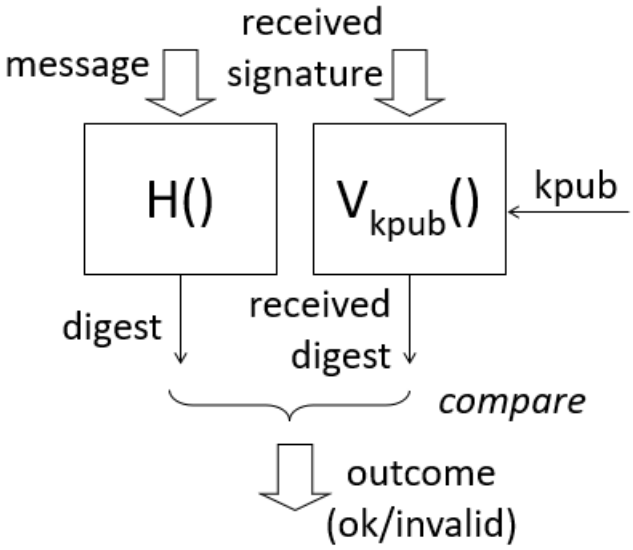


Figure 17: Logical Representation of a Digital Signature Verification.

7.1 Digital Signatures in OpenSSL

The following API functions initialize and update a context for digital signature and signature verification.

```
EVP_SignInit(ctx, <HashType>)
EVP_SignUpdate(ctx, buf, len)
EVP_VerifyInit(ctx, <HashType>)
EVP_VerifyUpdate(ctx, <HashType>)
```

They work exactly as their hashing counterparts in Sec. 6.1.1. Remember that the hash context must be previously allocated with `EVP_MD_CTX_new()`, and finally deallocated with `EVP_MD_CTX_free()`. Moreover, OpenSSL makes use of two more APIs:

```
int EVP_SignFinal(EVP_MD_CTX* ctx, unsigned char* sgnt, unsigned int* sgntl,
EVP_PKEY* prvkey);
```

Finalizes the context for digital signature. It also produces the digital signature by means of the private key.

- `ctx`: the digital signature context;
- `sgnt`: the buffer that will contain the digital signature. It must be *preallocated* using the number of bytes returned by the `EVP_PKEY_size(prvkey)` function;
- `sgntl` it will contain the actual size of the signature.

- `prvkey` the private key with which the function will compute the signature.

`int EVP_VerifyFinal(EVP_MD_CTX* ctx, unsigned char* sgnt, unsigned int sgntl, EVP_PKEY* pubkey);`

- `ctx`: the signature verification context;
- `sgnt` the buffer containing the received signature;
- `sgntl` the received signature length;
- `pubkey` the public key that will be used for the signature verification process.
- `returns`: 0 on invalid signature, -1 on other errors, 1 for success. *Do not use* the received data if the signature is invalid or other errors occurred.

7.2 Digital Signature Example

The following code snippet shows how to compute a signature with a private key within OpenSSL.

```
1  #include <openssl/evp.h>
2
3  int main(){
4  /*..retrieve private key..*/
5  char msg[] = "Lorem ipsum dolor sit amet.";
6  unsigned char* signature;
7  int signature_len;
8  signature = malloc(EVP_PKEY_size(prvkey));
9  EVP_MD_CTX* ctx = EVP_MD_CTX_new();
10 EVP_SignInit(ctx, EVP_sha256());
11 EVP_SignUpdate(ctx, (unsigned char*)msg, sizeof(msg));
12 EVP_SignFinal(ctx, signature, &signature_len, prvkey);
13 EVP_MD_CTX_free(ctx);
14 return 0;
15 }
```

The following code snippet shows how to verify a signature with a public key within OpenSSL.

```
1  #include <openssl/evp.h>
2
3  int main(){
4  /*... retrieve public key ..*/
5  unsigned char* msg = /* ... retrieve it ... */;
6  int msg_len = /* ... retrieve it ... */;
7  unsigned char* signature = /* ... retrieve it ... */;
8  int signature_len =  /* ... retrieve it ... */;
9  EVP_MD_CTX* ctx = EVP_MD_CTX_new();
10 EVP_VerifyInit(ctx, EVP_sha256());
```

SECTION 8

Certificates

A main problem in asymmetric cryptography is to guarantee the association between an entity (Alice, Bob, a bank) and its cryptographic keys. For example, Alice *needs* to be sure of the public key that the server reachable at the domain "www.server.com" uses. Sending the public key over Internet is not safe, because a man in the middle could change it. We need a trusted third entity called certification authority (CA), with the following features:

- everyone trusts the CA;
- everyone knows the CA's public key;
- the CA releases signed certificates, each one binding a given subject to a unique cryptographic quantity.

The most common type of certificate is a public key certificate, which binds a given subject (usually an Internet domain or a company) to a unique public key.
To obtain a certificate, a subject (a person, a company, a domain), must request a certificate from a CA. The request must contain the subject name (for example, the name of the company) and may contain also the public key of the server that will use the certificate. Before issuing the

certificate, the CA makes sure that the subject really exists and really owns that cryptographic quantity (validation process).
How can the CA verify that the public key in the request is used by the subject in the request? There are several types of certificates: the most common ones are certificates with domain validation (*DV certificates*) and certificates with extended validation (*EV certificates*). To issue a domain validation certificate, the CA only needs to assure that the requesting subject owns a particular Internet domain. Usually it is done by sending a challenge email to an email server running on that domain. The subject proves the domain's ownership by responding to the email. With extended validation, the CA assures the physical existence and the identity of the requesting subject. In particular, *three* checks must be performed: (i) legal existence; (ii) physical existence; (iii) operational existence (that is, the subject is a still-operating company). The extended validation requires a face-to-face identification with a CA's employee, a notary, or a lawyer. Not all CA's are allowed to issue EV certificates. Web browsers usually signals the presence of an EV certificate with green colors (a green lock in Firefox/Chrome, a green address bar in Internet Explorer).
There are two methods of creating and deploying a certificate. The simplest way is to create the key pair at the certification authority, create the certificate, and then send the private key and the certificate to the subject. Of course, the subject must trust the certification authority to create a good key pair, and to guarantee the confidentiality of the created private key. This means that the subject trusts that the CA securely destroys the generated private key. Moreover, a secure channel is needed to send the private key fro the CA to the subject. This approach is suitable for small organizations, when the CA and the subject are administrated by the same entity.
A more secure method requires the subject to create its own key pair (i.e., public and private keys). Then, the subject creates also a *certificate signing request* (CSR) signed with the newly created private key. The CA receives the CSR and performs the validation (either DV or EV). If the validation is sccessful, the CA creates the certificate, and sends it to the subject. In this way, the subject does not have to trust the CA in creating good key pairs and in destroying the private key, and it does not need a secure channel. This method is suitable when the CA and the subject belong to different organizations.

8.1 Certificates in OpenSSL

At the beginning of the program, you have to include the library `<openssl/x509.h>`, which contains the high-level API functions useful for storing and managing `x509`-compliant certificates. Moreover, the library `<openssl/x509_vfy.h>` contains APIs useful for verifying certificates.

`X509; X509_CRL; (data structures)`
Represent an x509-compliant certificate, and a x509-compliant Certificate Revocation List (CRL), respectively.

8.1.1 Managing Certificates

```
X509* PEM_read_X509(FILE* fp, NULL, NULL, NULL);
```

Allocates an x509 certificate structure and loads it from a PEM file. The file pointer fp is the file from which the API reads the certificate. Clearly the file must have been already opened with fopen() or a similar function. The API returns the X509 structure (or NULL if error).

```
X509_CRL* PEM_read_X509_CRL(FILE* fp, NULL, NULL, NULL);
```
Allocates an x509 certificate revocation list structure and loads it from a PEM file. The file pointer fp is the file from which the API reads the CRL. Clearly the file must have been already opened with fopen() or a similar function. The API returns the X509_CRL structure (or NULL if error).

```
X509_NAME* X509_get_subject_name(X509* cert);
```
The variable cert contains the certificate. The API returns an X509_NAME structure representing the subject's distinguished name.

```
X509_NAME* X509_get_issuer_name(X509* cert);
```
The variable cert contains the certificate. The API returns an X509_NAME structure representing the issuer's distinguished name. The issuer is usually a certification authority.

```
char* X509_NAME_oneline(X509_NAME* name, NULL, 0);
```
Allocates and returns a NULL-terminated string containing a one-line representation of a distinguished name, in the following form:
/C=<Country>/ST=<State>/L=<Location>/O=<Organization>/CN=<Common_Name>.
An example is a certificate from Google:
/C=US/ST=California/L=Mountain View/O=Google Inc/CN=*.google.com.
The string must be freed afterwards with free(). The variable name is the distinguished name to convert.

8.1.2 Certificate Verification

A store, represented in Fig. 18, is a collection of CA's root certificates, other trusted certificates, and a CRL. Everything that is in the store is considered trusted by the subject. When a subject (Alice) wants to verify another subject's identity (Bob), she verifies the certificate (Bob's certificate) against her store. If it is possible to verify Bob's certificate from Alice's store, then Alice will consider Bob's certificate as verified.
X509_STORE (data structure)
Represents a store.

```
X509_STORE* X509_STORE_new();
```
Allocates an empty store and returns it (or returns NULL if an error occurred). You must deallocate the store at the end of the program.

```
void X509_STORE_free(X509_STORE* s);
```
Deallocates a store.

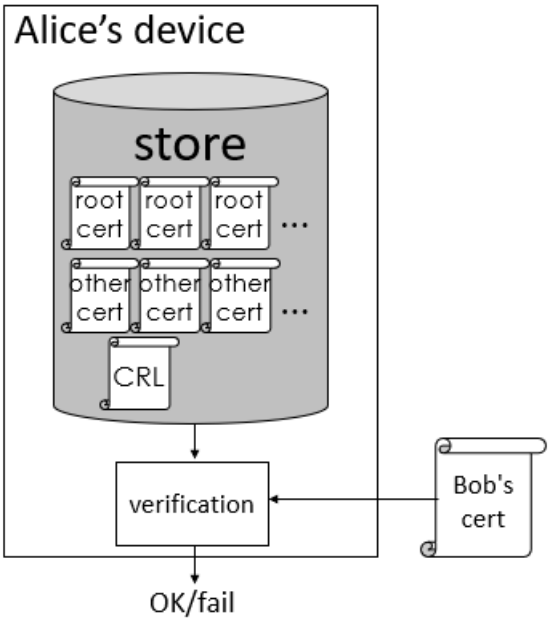


Figure 18: An overview of a Store in OpenSSL.

`int X509_STORE_add_cert(X509_STORE* s, X509* x);`
Adds a trusted certificate to the store. From now on the certificate is memory-managed by the store: deallocating the store automatically deallocates the certificate.

- `s`: the previously allocated store;
- `x`: the certificate to add;
- `return`: 1 on success, non-1 on error.

`int X509_STORE_add_crl(X509_STORE* s, X509_CRL* x);`
Adds a trusted CRL to the store. Afterwards, the CRL is memory-managed by the store: deallocating the store automatically deallocates the CRL.

- `s`: the previously allocated store;
- `x`: the `CRL` to add;
- `return`: 1 on success, non-1 on error.

`void X509_STORE_set_flags(X509_STORE* s, X509_V_FLAG_CRL_CHECK);`
It configures the store `s` to check against the `CRL` every valid certificate before returning a successful validation.

With the APIs shown above, one can build and setup the store. In the following, we show the APIs needed to *verify* any certificate that Alice may receive.

`X509_STORE_CTX` (data structure)
Represents a context for certificate verification.

`X509_STORE_CTX* X509_STORE_CTX_new();`

Allocates a new certificate-verification context. It returns the allocated context, or NULL if an error occurs.

`int X509_STORE_CTX_init(X509_STORE_CTX* ctx, X509_STORE* s, X509* cert, NULL);`
Initializes the certificate-verification context.

- `ctx`: the context to be initialized;
- `s`: the store against which the certificate will be verified;
- `cert`: the certificate to be verified;
- `return`: 1 on success, non-1 on error.

`int X509_verify_cert(ctx);`
Verifies the certificate passed at initialization time. It returns 1 if the certificate has been verified, 0 if it cannot be verified, <0 if some other error.

`void X509_STORE_CTX_free(X509_STORE_CTX* ctx);`
Deallocates the certificate-verification context `ctx`.

`EVP_PKEY* X509_get_pubkey(X509* cert);`
Extracts the public key from the certificate `cert`. It returns the `EVP_PKEY` structure representing the public key, or `NULL` if an error occurs. You should call this API only after having successfully verified the certificate.

SECTION 9

Transport Layer Security (TLS)

In OpenSSL, a Transport Layer Security (TLS) connection is represented by an SSL object. An SSL object is created by a *factory* object called `SSL_CTX`. A factory holds a *store* to authenticate the peer. The store contains the certificates of one or more Certification Authorities. The *store* may also contain a certificate plus a private key belonging to the client, used to authenticate the client itself to another peer. A single factory can create several TLS connections, as shown in Fig 19. Each TLS connection sends and reads bytes from a character stream, represented by a `socket BIO` object, which is in turn attached to a socket.

9.1 TLS in OpenSSL

The concept of character stream is represented in OpenSSL by BIO objects. As seen in section 2.5.1, the BIO is a structure that is used as an intermediate step in the conversion from a buffer to an OpenSSL structure, or vice-versa. To this end, a `socket BIO` can be associated to the TCP sockets, so that the programmer can read/write directly from/on it[1], while the OpenSSL API manages encryption and authenticity.

```
BIO* BIO_new_socket(int socket, BIO_NOCLOSE);
```

Allocates a new *socket BIO*, i.e., a BIO sending data to and receiving data from the network. The *BIO_NOCLOSE* flag specifies that when the BIO is deallocated, the wrapped socket **is not** automatically closed. On the server, the BIO must be associated to the connected socket, not to the listening socket. Thus, you have first to call `accept()` to retrieve a connected socket, and then associate the retrieved socket to a new BIO. Remember to deallocate the BIO as described

[1] Actually, it writes on the connection, as we will see in few pages.

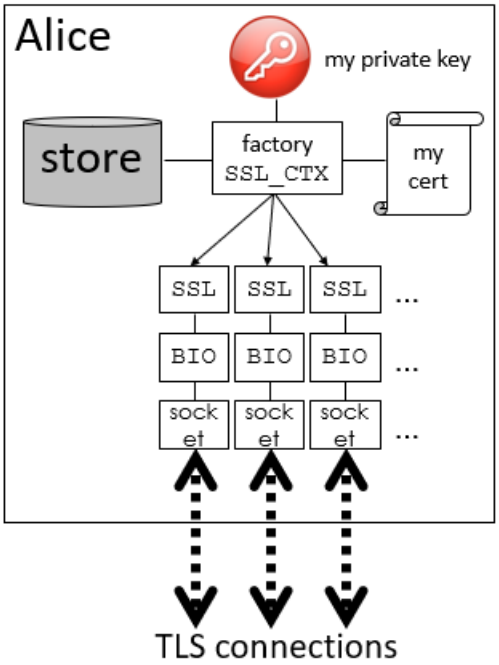


Figure 19: Representation of a factory and its TLS connection in OpenSSL.

in section 2.5.1.

9.1.1 Creation and Setup of a Factory

Before talking about a TLS connection, we have to perform some setup tasks. The following functions are defined in `openssl/ssl.h`, which you must include in your program. First of all, the functions `SSL_library_init()` and `SSL_load_error_strings()` have to be called to manage connections and errors, respectively. At the time of writing[2] TLSv1.1 is the higher version protocol to have been officially deprecated by the IETF in march 2021.

`void SSL_library_init();`
Initializes the internal OpenSSL data structures for managing TLS connections.

`void SSL_load_error_strings();`
Initializes the internal OpenSSL table of error descriptions.

`SSL_CTX` data structure
Represents a factory of `SSL` objects.

`SSL_CTX* SSL_CTX_new(TLS_method());`
Allocates a new factory implementing the TLS/SSL protocol. The parameter *TLS_method()* tells the factory to always negotiate with the peer the highest safe version of TLS supported by them both, among SSLv3, TLSv1, TLSv1.1, TLSv1.2 and TLSv1.3. However, TLSv1.1 and below are all **deprecated**. In place of the parameter `TLS_method()` the programmer can set a specific version.

[2] February 2022

Unluckily, to specify a single protocol is a deprecated behaviour by the OpenSSL documentation. Since 3 out of the 5 negotiable protocols are deprecated by the IETF, what should a programmer do?

OpenSSL provides utility functions for the factory. In particular it can set the minimum or maximum protocol version that it wants to use.

```
int SSL_CTX_set_min_proto_version(SSL_CTX *ctx, int version);
int SSL_CTX_set_max_proto_version(SSL_CTX *ctx, int version);
```

These functions set the minimum and maximum supported protocol versions for the ctx. Currently supported versions are `SSL3_VERSION`, `TLS1_VERSION`, `TLS1_1_VERSION`, `TLS1_2_VERSION`, `TLS1_3_VERSION` for TLS.

Therefore, to solve such problem, the programmer should first invoke the `SSL_CTX_new(TLS_method())` API, and then invoke the `SSL_CTX_set_min_proto_version(ctx, TLS1_2_VERSION)`, in this order.

WARNING! TLSv1.3 has a mechanism called "tickets" very useful for session resumptions. In some applications, however, this mechanism is hindering, since it disables the possibility of a connection in which the server does not send any data to the client.
`int SSL_CTX_set_num_tickets(SSL_CTX* ctx, 0);` Called by a server, disables the ticket functionalities in TLSv1.3. It makes TLS less efficient in case of session resumption. In case the server uses TLSv1.3 and never sends data to the client, calling this is **mandatory**.

9.1.2 Populating and Configuring a Factory

Now, there is a factory... but it is empty! The factory must be "populated" by first retrieving the store with the `SSL_CTX_get_cert_store(ctx)` API. Then, the programmer can initialize the store by adding to said store the RootCA certificates and CRLs in the same way we have seen in Sec. 8.
In case the programmer is coding a server (or any client that has a certificates, although this rarely happens) the factory needs to know the certificate and the private key used by the server, as shown in Fig. 19.

```
X509_STORE* SSL_CTX_get_cert_store(ctx);
```

Returns a reference to the store of a factory (or `NULL` if error). The store can be modified to add certificates, CRL's, and so on.

```
int SSL_CTX_use_certificate(SSL_CTX* ctx, X509* cert);
```

Sets the certificate of the factory's owner. Usually it is called only by servers (rarely clients have certificates). It returns 1 on success, non-1 on error.

```
int SSL_CTX_use_PrivateKey(SSL_CTX* ctx, EVP_PKEY* prvkey);
```

Sets the private key of the factory's owner, associated to the certificate set before. If the certificate has been set, then the API also checks the validity of the public key-private key coupling. It returns 1 on success, non-1 on error.

After initializing the store, the programmer should now properly configure the settings of the factory.

`void SSL_CTX_set_verify(SSL_CTX* ctx, int mode, NULL);`
Sets the flags to tell to a factory whether to request and verify the other peer's certificate. The variable `mode` is a set of logical flag in `OR`. It should be set differently if it is called server side or client side.

- `SSL_VERIFY_NONE` (client side): the client does not request nor verify the server's certificate;
- `SSL_VERIFY_PEER` (client side): the client requests and verifies the server's certificate;
- `SSL_VERIFY_NONE` (server side): the server does not request nor verify the client's certificate.
- `SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT` (server side): the server requests and verifies the client's certificate. The flag `SSL_VERIFY_FAIL_IF_NO_PEER_CERT` is meaningful only on server side and MUST be used with the `SSL_VERIFY_PEER` flag. It results in the factory aborting the connection if the client does not possess a certificate.

`void SSL_CTX_free(SSL_CTX* ctx);`
Deallocates a factory.

9.1.3 TLS Connection Management

Now, we are ready to make a TLS connection! The only prerequisite is that the programmer should have already generated a socket bio associated to the socket used by TCP.
From a factory the programmer can generate many TLS connections, through the `SSL_new` API. Then, the TLS connection needs a `socket BIO` where to write or read data to/from the network. Usually, the same BIO is used both in read and write mode.
You should think about a TLS connection and a TCP one as a Matrioska-doll. Before opening a TLS connection between two parties, they **have to** be already connected through TCP. The programmer should call the `SSL_connect(SSL* ssl);` API (`or the SSL_accept(SSL* ssl);` API) on the Client (or the Server) only after the respective TCP functions have been invoked. Indeed, the programmer should associate the `socket BIO` to the TLS connection *before* calling the latter two APIs.

`SSL* SSL_new(SSL_CTX* ctx);`
Creates a new TLS session from a factory, and returns the created TLS session.

`void SSL_set_bio(SSL *ssl, BIO *rbio, BIO *wbio);`

Sets the input (`rbio`) and the output (`wbio`) BIO's for a TLS connection. Usually the same `socket BIO` is used in both direction.

`int SSL_connect(SSL* ssl);`
Initiates a TLS connection from the client side, and verifies the server's certificate[3]. The function is blocking if the underlying BIO is read-blocking (it is, by default). The wrapped socket must be already connected, so the "classic" `connect()` function must be called before `SSL_connect()`. The function returns 1 if the connection was successful, 0 if it was gracefully shut down by the peer, <0 if a fatal error has occurred.

`int SSL_accept(SSL* ssl);`
Initiates a TLS connection from the server side, and verifies the client's certificate[4]. The function is blocking if the underlying BIO is read-blocking (it is, by default). The wrapped socket must be already connected, so the "classic" accept() function must be called before *SSL_accept()*. The function returns 1 if the connection was successful, 0 if it was gracefully shut down by the peer, <0 if a fatal error has occurred.

After the connection has been established, the programmer can check the negotiated protocol version and see the peer certificate (if any has been sent). The factory controls the validity of a certificates, but the programmer may also want to check the common name of a certificate.
`const char* SSL_get_version(const SSL* ssl);`
Retrieves a string specifying the negotiated version of the TLS protocol, for example "TLSv1.3".

`X509* SSL_get_peer_certificate(const SSL* ssl);`
Retrieves the peer's certificate if present, or `NULL` if no certificate has been received.

Now, it is time to read and write on this God-blessed secure channel! OpenSSL provides two APIs to send/receive a buffer to/from a peer. This data will be encrypted and, if the peers have authenticated themselves, authenticated.

`int SSL_write(SSL *ssl, const void *buf, int num);`
Sends `num` bytes from the buffer `buf` to the TLS connection. It returns the number of bytes sent, or a negative value on error.

`int SSL_read(SSL *ssl, void *buf, int num);`
Receives at most `num` bytes from the TLS connection to the buffer `buf`. It is blocking if the underlying `BIO` is read-blocking (which it is, by default). The function returns the number of bytes actually received, or a negative value if an error occurs. Note that `SSL_read()` could receive less than `num` bytes, as usually happens in TCP-based communications. If you want to receive exactly

[3] If the certificate verification has been set with the flag `SSL_VERIFY_PEER`

[4] If the certificate verification has been set with the flag `SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT`

num bytes, you have to arrange a cycle with the SSL_read().

Once the communication is over, you need to gracefully shutdown the connection and deallocate the employed resources. Please note that you **do not have to** deallocate the factory once a connection is over! The factory **must** be freed only when terminating the main program.

`int SSL_shutdown(SSL *ssl);`
Closes a TLS connection. It is blocking if the underlying BIO is read-blocking (which it is, by default). This function must be called before the "classic" close() function on the socket. It returns 1 on success, non-1 on error.

`void SSL_free(SSL *ssl);`
Deallocates a TLS connection. It also frees the associated *BIO*s, so there is no need to invoke BIO_free().

9.2 TLS Communication Session Example

```
1  #include <openssl/evp.h>
2
3  int main(){
4  ////////////////////////////////////////////////
5  //* This Code is the same for client and server *//
6  ////////////////////////////////////////////////
7  /* Open a connection (client & server sides) */
8  BIO* bio;
9  SSL_CTX* ctx;
10 SSL* ssl;
11 int ret;
12 SSL_library_init();
13 SSL_load_error_strings();
14 ctx = SSL_CTX_new(TLS_method());
15 SSL_CTX_set_num_tickets(ctx, 0); /* <-- only on the server
16 /* ... Possibly bind the SSL_CTX with CA certificates and/or my own certificate and privat
17 int sk = /* ... create a traditional socket ... */
18 /* ... perform a traditional connection ... */
19 bio = BIO_new_socket(sk, BIO_NOCLOSE);
20 ssl = SSL_new(ctx);
21 SSL_set_bio(ssl, bio, bio);
22
23 ////////////////////////////////////
24 //* Start a connection client-side *//
25 ////////////////////////////////////
26 /* ... Client-side connect the traditional socket sk ... */
```

```
27 ret = SSL_connect(ssl);
28 ret = SSL_write(ssl, "Some data", strlen("Some data"));
29 /* ... Rest of the protocol ... */
30
31 ////////////////////////////////////
32 //* Start a connection server-side *//
33 ////////////////////////////////////
34 /* ... Server-side connect the traditional socket sk ... */
35 ret = SSL_accept(ssl);
36 ret = SSL_read(ssl, buffer, buffer_len);
37 /* ... Rest of the protocol ... */
38
39 ////////////////////////////////////////////////
40 //* This Code is the same for client and server *//
41 ////////////////////////////////////////////////
42 SSL_shutdown(ssl);
43 SSL_free(ssl);
44 SSL_CTX_free(ctx);
```