Cryptography with OpenSSL– Basic operations

Laboratory for the class "Information Systems Security" (01TYMOV, 02KRQ)
Politecnico di Torino – AA 2023/24
Prof. Diana Berbecaru

prepared by:
Diana Berbecaru (diana.berbecaru@polito.it)
Enrico Bravi (enrico.bravi@polito.it)

v. 1.0 (21/10/2023)

Contents

Purpose of the laboratory

The goal of this laboratory is to allow you experiment the procedures and the problems associated to the use of different basic cryptographic techniques. The laboratory is based on the use of OpenSSL (`http://www.openssl.org/`), a cryptographic suite released as open-source software with Apache license and available for various platforms, including Linux and Windows.

All the exercises proposed use OpenSSL command line programs that provide various cryptographic functions via a specific shell, which you can start with the following command:

```
openssl command [ command_opts ] [ command_args ]
```

Specifically, to run the exercises proposed in this text, you will use the following OpenSSL commands:

```
enc genrsa rsa ecparam ec pkeyutl dgst rand speed
```

which will be presented further below. For a complete list of options, as well as for a detailed description of the OpenSSL commands, you can consult the corresponding `man` pages.

openssl enc

The OpenSSL `enc` command allows the encryption and decryption of data with several symmetric cipher routines. For instance, you can execute the following command to view the list of parameters for the command `enc`:

```
openssl enc -help
```

For example, to view the list of algorithms supported by the command `enc`, you can execute the command:

```
openssl enc -ciphers
```

or

```
openssl list -cipher-algorithms
```

We provide a short description of the main commands used throughout this laboratory (we remind you that the parameters enclosed by square brackets are optional):

```
openssl enc [-encryption_algorithm] [-e] [-d] [-K key] [-iv vector]
   [-in file_input] [-out file_output] [-nopad] [-p]
```

where:

- *-encryption_algorithm*, is the symmetric encryption algorithm used to encrypt and/or decrypt data (e.g. `-aes128`, `-aes-cbc-256`, `-rc4`, the complete list can be found with the command `openssl enc -ciphers`);
- `-e` indicates that the operation to be performed on data is encryption;
- `-d` indicates that the operation to be performed on data is decryption;
- `-K` *key*, indicates the key to use for the symmetric cryptographic operations;

  ATTENTION

  OpenSSL parameters are case sensitive, thus pay attention to use the uppercase letter 'K' and not the lowercase 'k' when the exercise will require it.

- `-iv` *vector*, indicates the initialization vector to use;
- `-in` *file_input* indicates the file containing the data to be encrypted or decrypted;
- `-out` *file_output* indicates the file where to save the result of an encryption or decryption operation;
- `-nopad`, indicates that the padding must not be applied

  ATTENTION

  If you use `-nopad` with a symmetric block algorithm, the length of the plaintext must necessarily be a multiple of the algorithm block otherwise the operation will fail. This option does not work instead with stream algorithms and OpenSSL ignores it.

- `-p`, is used to print on standard output the key and the initialization vector (and also the *salt*, if used).

openssl dgst

The OpenSSL command `dgst` allows to calculate the digest of data using different algorithms. To view the list of supported algorithms by the command `dgst`, execute the command:

```
openssl list -digest-commands
```

For a detailed description of the `dgst` command, you can use the command:

```
man dgst
```

The syntax of the `dgst` command is given below together with the main parameters:

```
openssl dgst [-digest_algorithm] [-out output_file] input_file(s)
```

where:

- *-digest_algorithm*, is the digest algorithm to use (e.g. `-sha256` to use the SHA256 algorithm (SHA2 family with a 256 bits digest), or `-sha512` to use the SHA512 algorithm (SHA2 family with a 512 bits digest); the complete list of digest algorithms supported can be found with the command `openssl list -digest-commands`);
- `-out` *output_file*, indicates the (name of the) file where the digest will be saved;
- *input_file(s)*, is the file containing the data on which the digest will be calculated (if absent, the digest will be calculated on data provided via standard input)

> **ATTENTION**
>
> `dgst` uses this form and does not use the option `-in` to specify the file containing the data on which the digest needs to be calculated.

openssl genrsa

To perform simple asymmetric operations with the RSA algorithm, you will have first to generate a pair of RSA keys with the OpenSSL command `genrsa`, whose syntax is:

```
openssl genrsa [-out filename] [numbits]
```

where:

- `-out` *filename* indicates that the generated (public and private) keys will be saved in the file named *filename*;
- *numbits* specifies the length (in bits) of the RSA modulus.

> **NOTE**
>
> The man pages (`man`) of OpenSSL refers to the private key, they actually mean both the private and public keys, since both are contained in the same data structure. The RSA public key can be separated from the private one by using a dedicated OpenSSL option. In general, however, the keys (public and private) are kept together since we refer to them as “a key pair”.

openssl rsa

To manage and use the RSA keys in cryptographic operations, you can use the OpenSSL `rsa` command, whose syntax is given below:

```
openssl rsa [-in file_input] [-out file_out] [-text] [-pubin] [-pubout] [-noout]
```

where:

- `-in` *file_input*, specifies the input file (containing an RSA public key or an RSA private key);
- `-out` *file_out*, saves the RSA keys (public or private)in the file *file_out* after executing the operation requested;
- `-text`, prints the keys in text format. In addition, the keys are also shown encoded in Base64 format unless you use also `-noout`;
- `-pubin`, is a parameter indicating that the key passed in input (via the `-in` option) is a RSA public key. Pay attention, if this parameter is not specified, the `rsa` command assumes that the input key is a (RSA) private key;
- `-pubout`, is the parameter used to generate as output only the RSA public key. Pay attention, if this parameter is not specified, the `rsa` command returns also the RSA private key;
- `-noout`, indicates that the keys in Base64 format do not have to be shown.

openssl ecparam

To manage and maipulate the EC algorithm parameters you can use the `ecparam` command, whose syntax is given below:

```
openssl ecparam [-list_curves] [-name curve] [-genkey] [-out -file_out]
```

in cui:

- `-list_curves`, lists the available curves;
- `-name` *curve* specifies a curve by its name;
- `-genkey` generates the private/public key pair;
- `-out` *file_out*, saves in the file *file_out* the private/public key pair.

openssl ec

To manage and maipulate the EC algorithm keys you can use the `ec` command, whose syntax is given below:

```
openssl ec [-in file_in] [-out file_out] [-pubout] [-pubin] [-text]
```

in cui:

- `-in` *file_input*, specifies the input files that must contain the private/public key pair to read;
- `-out` *file_out*, specifies the file where the extract key will be saved;
- `-pubout`, indicates to produce in output the public key (if not specified, the private key is instead

produced);

- `-pubin`, indicates to read in input the public key (if not specified, the private key is instead read);
- `-text`, prints the keys in text format.

openssl pkeyutl

The command `pkeyutl` performs asymmetric encryption/decryption, signature/verification, and key exchange, by using various asymmetric algorithms. Currently, the asymmetric algorithms supported are:

- RSA, to encrypt, decrypt, sign and verify data;
- DSA, to sign and verify data;
- Diffie-Hellman (DH), for symmetric key exchange;
- Elliptic Curve (EC) algorithms to sign and verify with ECDSA or to establish a symmetric key with ECDH.

```
openssl pkeyutl [-encrypt] [-decrypt] [-sign] [-verify] [-verifyrecover]
     [-in file_input] [-out file_output]
     [-pubin] [-inkey file_key] [-sigfile signature]
```

where:

- `-encrypt/-decrypt`, encrypts with the public key or decrypts with the private key (the content of) the input file whose name is passed with the parameter `-in`;
- `-sign`, generates the signature applied on the input file (passed with `-in`) by using the key passed in with the option `-inkey`. A private key is required in this case. More precisely, if an RSA key is used, the file passed in input is encrypted with the private key, otherwise the operation fail;
- `-verify`, computes the signature on the input file (passed in with `-in`) by using the public key passed in with the option `-inkey` (a public key is required) and compares this signature with another signature passed in with the option `-sigfile`. If the file contains a pair of public and private keys, it will be used only the public key. Returns a boolean value (*Signature Verified Successfully/Signature Verification Failure*);
- `-verifyrecover`, verifies the signature passed in as the input file (that is the file passed in with the option `-in`) by using the key passed in with the option `-inkey` and shows the decrypted data. Pay attention that an RSA public key is required (the command does not function with DSA, DH or EC). This option is available only if an RSA key is used and, in practice, the input file is decrypted with the public key;
- `-in` *file_input*, specifies the input file (containing the message to encrypt, decrypt, sign or verify);
- `-out` *file_out*, saves the output of `pkeyutl` in *file_out*;
- `-inkey` *file_key*, specifies that the file *file_key* contains the public key or the private key;
- `-pubin`, parameter indicating that the key passed in input (with the option `-inkey`) is a public key. Pay attention, if this parameter is not specified `pkeyutl` assumes that a private key is passed as input;
- `-sigfile` *signature*, specifies that the signature to be used for comparison when using the option `-verify` is memorized in the file *signature*.

Note: In `pkeyutl` the order in which the parameters are passed in is important, while in the other OpenSSL commands typically the order of parameters is not important. We advise you to follow the order of parameters presented above, otherwise the execution of the `pkeyutl` command will fail.

openssl speed

The OpenSSL command `speed` can be used to measure the performance of the various algorithms implemented by OpenSSL. To measure the performance of a specific algorithm, you can use the following command:

```
openssl speed [name_algorithm1 name_algorithm2 ...]
```

If you do not specify any algorithm name, it will be evaluated the speed of all the algorithms supported by OpenSSL (this process may be long, it depends on the performance of the CPU on which you are performing this operation).

openssl rand

With the command `rand` you can generate `numbyte` pseudo-random data and save them in the file `file_name`:

```
openssl rand -out file_name numbyte
```

Other commands

When executing the exercises, you may need to exchange some data among two computers: for this purpose, you can use the `scp` utility (acting as client) and the `ssh` server. Consequently, on one of the two computers (let's say Alice) you will have to start the `ssh` server:

```
systemctl {status start restart stop enable } ssh
```

or

```
service ssh start
```

or

```
/etc/init.d/ssh start
```

If you want to connect remotely as the `kali` user, use the password "kali". For example, Bob can transfer a file named `prova` to Alice's host with the command (the file will be copied to her home directory):

```
scp prova kali@IPaddress_Alice:
```

1 Symmetric cryptography

1.1 Symmetric cryptography exercises

In this exercise, you'll use the OpenSSL command `enc` to encrypt a plaintext.

Create a text file named `ptext` containing the message you want to encrypt, such as:

```
This message is my great secret
```

After creating `ptext`, check that its size is 32 bytes (for example, with the command `ls -l`).

Let's now encrypt with the AES algorithm in CBC mode by using a 128-bit key the file `ptext`, where the symmetric key will be derived from a password (inserted any password you want when asked):

```
openssl enc -in ptext -e -out ctext.aes128 -aes-128-cbc -nosalt
```

The encrypted message has been saved in the `ctext.aes128` file. How long is the file `ctext.aes128` with respect to the original plaintext file?

→ 48

Now find out and write in the box the optional parameter that allows you to also show the used IV and the key.

→ -p

Run several times the encryption operation (by using the same password), with the command:

```
openssl enc -in ptext -e -out ctext.aes128 -aes-128-cbc -p
```

Observe the key generated and the length of the ciphertext file `ctext.aes128`. Has the key changed (even if you have used the same password)? If yes, why? How long is the ciphertext file `ctext.aes128` now? If the length is different than in the previous encryption, explain what is the difference.

→ The key changes even if you use the same password due to the useof an initialization

The salt itself needs to be included in the ciphertext to ensure that it can be used during decryption.This means that

Now execute the following command:

```
openssl enc -e -aes-128-cbc -md sha512 -pbkdf2 -iter 100000 -nosalt -in ptext
-out ctext.aes128.pbkdf2 -p
```

Which is the difference with respect to the previous command ?

→ - md sha512: specifies the message digest algorithm to use for key derivation. In this case, it's set to SHA-512.-pbkdf2: specifies the key der

Find out the corresponding command to use to decrypt the file `ctext.aes128.pbkdf2`.

Now also find out the commands to encrypt the same data (`ptext`) with another symmetric algorithm (e.g. 3DES with 2 and 3 keys, in CBC mode). The encrypted message must be saved in a file named `ctext.`*algorithm*.

→ openssl enc -d -aes-128-cbc -md sha512 -pbkdf2 -iter 100000 -nosalt -in ctext.aes128.pbkdf2 -out decrypted_output.txt

openssl enc -e -des-ede3-cbc -in ptext -out ctext.des-ede3-cbc

Find out the command to encrypt the plaintext with the AES algorithm in CTR mode.

→ openssl enc -e -aes-128-ctr -in plaintext.txt -out ciphertext.enc

Compare now the length of the generated `ctext.*` file against the length of the original `ptext`. Is the size of the ciphertext file different from the size of the plaintext?

→ 00000000 53 61 6c 74 65 64 5f 5f 59 b3 41 c2 8f 61 28 29 |Salted__Y.A..a()

In conclusion: what happens when you use the following command to encrypt the `ptext` file?

```
openssl enc -e -in ptext -out ctext.aes128.nopad -aes-128-cbc -nopad -p
```

Check the size of the file `ctext.aes128.nopad` and confront it with the size of the file `ptext`.

→ The size of ctext.aes128.nopad is 48 due to the salt

Find out the commands to encrypt the same data (`ptext`) with the ChaCha20 algorithm and a key of your choice. Explain the difference between the encryption with the AES algorithm in the CBC and CTR modes and the encryption with the ChaCha20 algorithm.

→ AES in CBC (Cipher Block Chaining) mode operates as a block cipher. It divides the data into fixed-size blocks and encrypts each block separately.
AES in CTR (Counter) mode and ChaCha20 operate as stream ciphers. They convert the data into a stream of key-based pseudorandom bitsand XOR it with the plaintext.
AES-CBC and AES-CTR require an IV. ChaCha20 uses a nonce instead of an IV

Now find out the commands to decrypt the files `ctext.`*algorithm*. Find out the OpenSSL commands used to decrypt the cyphertext files and write them down:

→ openssl enc -d -chacha20 -k mykey -in ctext.chacha20 -out decrypted_output.txt

Verify whether the decryption has been performed correctly (by controlling every time the content of the file resulted from the decryption operation).

1.2 Brute force attack

In this exercise you'll perform a brute force attack against a symmetric encryption algorithm.

1. Alice prepares a new plaintext message and saves it in the file `ptext`. Next Alice chooses a key, which is **4 bits long** (that is one hexadecimal character) and encrypts the message with the following command:

   ```
   openssl enc -e -in ptext -out ctext_alice -K short_key -chacha20
   ```

2. Alice makes available the encrypted message `ctext_alice` to Bob (e.g. by transferring the file with the `scp` tool to Bob, after having started the `ssh` server on Bob).

3. Bob knows the original message was a text message. He wants to find out the key used by Alice to encrypt the message, by using `ctext_alice`. After how many tries (at most) can he discover the Alice's secret key? Try out.

NOTE

To perform the operation 3 you don't need to write a script, given the limited number of trials that have to be done. Since this is a laboratory exercise, you can simply change the value of the key used in the command line.

→

1.3 Performance evaluation

In this exercise, you will evaluate the time required to perform the cryptographic operations and the additional data overhead.

Create files of different sizes (e.g. 100 B, 10 kB, 1 MB and 100 MB) by using the following command:

```
openssl rand -out r.txt size_in_bytes
```

OpenSSL contains a command used to measure the performance of various cryptographic algorithms. For example, execute the following command to measure the performance of AES-128-CBC:

```
openssl speed aes-128-cbc
```

Now complete the Table 1 with the times required to encrypt the files with different algorithms and different key lengths.

Will the decryption time (using the algorithms in the table) be significantly different? (try out).

	100 B	10 kB	1 MB	100 MB
DES-EDE3				
AES-128-CBC				
AES-192-CBC				
AES-256-CBC				
ChaCha20				

cmd time for the evaluetion

Table 1: *Performance of some symmetric encryption algorithms.*

How much time is required for one single encryption operation (for each algorithm)?

→

divide the time for the number of operation with speed cmd

2 Asymmetric cryptography

2.1 RSA Key generation

Starting from the OpenSSL command `genrsa`, run the following OpenSSL command used to generate a 2048 bit RSA key pair, and save it in the `rsa.key.`*name*, where *name* is the name of the person creating the key (e.g. Alice or Bob):

```
openssl genrsa -out rsa.key.name 2048
```

Once you have generated the RSA key pair, check out the content of the file `rsa.key.`*name* with the following command:

```
openssl rsa -in rsa.key.name -text
```

NOTE

Additionally, the standard PKCS#1 defines also the primitive operations for encryption and signatures and secure cryptographic schemes.

Which of the parameters can be made public (are part of the public key) and which ones instead must be kept secret (are part of the RSA private key)?

→

Suppose you want to distribute your new RSA public key to your colleagues: write down the OpenSSL commands used to extract the RSA public key from the file `rsa.key.`*name* to the file `rsa.pubkey.`*name*, and to view its content:

→

2.2 RSA encryption and decryption

In this exercise, you will encrypt/decrypt a block of data by using the RSA algorithm and the RSA key that you have generated in the previous exercise. Create a text message, like for example "This is a confidential message", and save it in a file named `plain`.

How can you use your RSA key pair to ensure confidentiality of the file `plain`? Which (RSA) key do you have to use?

→

Write down the OpenSSL command to encrypt the file `plain`, and save the result in the file `encRSA` (suggestion: use the command `pkeyutl`).

→

Which operation performs the following OpenSSL command and which key is used in this command?

```
openssl pkeyutl -encrypt -in plain -inkey rsa.key -out plain.enc.RSA.for.name
```

→

Write down the command used to decrypt the message encrypted above:

```
→
```

Try to download from Internet the following file [1]

```
wget https://cacr.uwaterloo.ca/hac/about/chap8.pdf
```

and try to encrypt it. Do you face any problem? Why (see the note below)?

```
→
```

NOTE

RSA allows in theory to encrypt any message, which interpreted as a binary value is smaller than the value of the modulus (that is a string of 2048 bits for a 2048-bit RSA key. Nevertheless, the PKCS#1 format imposes additional limitations that are due to the encapsulation of the message to be encrypted in a PKCS#1 envelope, and in particular due to the padding required. In practice, if you have an N-bytes RSA key, you can perform successfully encryption/decryption operations with OpenSSL only if the (plaintext) data is at most $N - 11$ bytes long.

If you downloaded the file `chap8.pdf`, look at its content it by using the command:

```
atril chap8.pdf &
```

(note: `atril` is a document viewer in Linux).

2.3 RSA signature generation and verification

The command `pkeyutl` allows not only to encrypt/decrypt blocks of data, but also to sign/verify them. Which is the difference in terms of the RSA operations to be performed?

```
→
```

Write down the OpenSSL `pkeyutl` command used to sign the file `plain`, and save the signature in the file `sig.`*nome*. Next, find out and write down the OpenSSL command used to verify the signature contained in the file `sig`, by using again the `pkeyutl` command. Which keys have you used for each of the above operations?

```
→
```

2.4 EC Key generation

Starting from the OpenSSL command `ecparam`, run the following OpenSSL command used to generate a SECG curve over a 192 bit prime field, and save it in the `ec.key.`*name*, where *name* is the name of the person creating the key (e.g. Alice or Bob):

```
openssl ecparam -name secp192k1 -genkey -out ec.key.name
```

[1] If you have problems to download, for example due to network problems, you could use instead the file `/etc/apache2/apache2.conf` in this exercise.

Suppose you want to distribute your new EC public key to your colleagues: starting from the OpenSSL command `ec`, write down the command used to extract the EC public key from the file `ec.key.`*name* to the file `ec.pubkey.`*name*, and to view its content:

→

2.5 EC signature generation and verification

The command `pkeyutl` allows also to sign/verify data with ECDSA algorithm.

Run the next OpenSSL `pkeyutl` command used to sign the file `plain` with ECDSA, and save the signature in the file `ecsig`.

```
openssl pkeyutl -sign -in plain -inkey ec.key.alice -out ecsig
```

Next, find out and write down the OpenSSL command used to verify the signature contained in the file `ecsig`, by using again the `pkeyutl` command. Which keys have you used for each of the above operations?

→

2.6 Performance evaluation

Use the command `openssl speed` to perform performance comparisons among:

- RSA keys of 2048, 3072, 4096, 7680, 15360 bits.

  How does the performance changes with respect to the key length?

  What is the difference (in terms of performance) between operations requiring the use of the public key and the ones requiring the use of the private key?

  →

- digital signature algorithms (low security): 1024 bit RSA (`rsa1024`), 1024 bit DSA (`dsa1024`), 160 bit ECDSA (`ecdsap160`).

  What differences do you note? Are the ECC operations more efficient than the RSA/DSA ones? Is it more efficient to sign or to verify with DSA/ECDSA? Is it more efficient to sign or to verify with RSA?

  →

- digital signature algorithms (high security): 4096 bit RSA (`rsa4096`), 256 bit ECDSA (`ecdsap256`).

  NOTE

  In practice, 256 bit ECC guarantees security strength equivalent to 3072 bit RSA/DSA.

  What differences do you note between the results obtained at this step and the ones obtained at the previous step? How does the performance decreases for RSA/DSA and for ECDSA with the increase of the key length (and thus of the security strength)?

→

3 Digest algorithms

3.1 Computation and verification of message digests

Create a text message like "This is a trial message to test digest functions!", save it in a file named `msg`, and calculate its digest by using different digest algorithms, such as SHA-256, SHA-512, SHA3-256, SHA3-384, and SHA3-512. Save the results (i.e. the digests) in the files `SHA256dgst`, and `SHA3-256dgst` respectively. Find out the correct OpenSSL commands to calculate the digests and write them down:

→

Now try to modify the message (e.g. delete the "!" at the end of the message) and recalculate the digest (with one algorithm at your choice). What do you note when you compare the (above) two digests (are they similar, the same or different)?

→

Assume you have a digest composed of only **the first** hexadecimal digit in the file `SHA256dgst`. Are you able to find a collision? After how many tries?

→

3.2 Performance evaluation

Evaluate the cost associated to the digest algorithms implemented by OpenSSL, by using the method explained/used in Exercise 1.3. Use the same files of 100 B, 10 kB, 1 MB e 100 MB created previously. Evaluate the performance with the specific OpenSSL commands:

```
openssl speed sha256
```

```
openssl speed sha512
```

Fill in the results in the Table 2.

	100 B	10 kB	1 MB	100 MB
SHA256				
SHA512				

Table 2: *Costs associated with some digest algorithms.*

Compare the results obtained in this exercise with the ones obtained for the symmetric encryption algorithms. How fast are the hash algorithms with respect to the symmetric encryption algorithms?

→

3.3 Application of digest algorithms: file integrity

The tools `sha1sum` and `hashdeep` allow to compute easily the hash of one or more files (the second one processes recursively the files contained in a directory with a chosen algorithm).

Create your own directory named `tree`, together with a subtree of directories and files

Write down the command used to calculate the digest of all files contained in the directory `tree` with SHA256. Save the digest in a file named `hash_list`.

→

Next change the content of a file (e.g. by adding a blank space at the end) and verify what happens with the following command:

```
hashdeep -c sha256 -r -x -k hash_list tree
```

How can an attacker can change the content of some files so that its modification remains undetected (hint: the hash_list is saved in a public, unprotected location.) ?

→

What kind of protections can you adopt to defend from such attack? Enumerate at least two methods:

→