Cryptography with OpenSSL– Basic operations

Laboratory for the class "Information Systems Security" (01TYM,02KRQ)
Politecnico di Torino – AA 2020/21
Prof. Antonio Lioy

prepared by:
Diana Berbecaru (diana.berbecaru@polito.it)
Ignazio Pedone (ignazio.pedone@polito.it)

v. 3.4 (21/10/2020)

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.

The additional files and programs needed in some of the proposed exercises are available in the archive

`lab02_support.zip`

To uncompress it, you can use the command:

```
7z x lab02_support.zip
```

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. `-md5` to use the MD5 algorithm, or `-sha1` to use the SHA1 algorithm; 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 root, use the password "toor". Alternatively, you could create the `kali` user on the host running the `ssh` server (with the command `adduser kali`) and connect remotely from another host with the `scp` tool to the `ssh` server. 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 root@`*IPaddress_Alice*`:`

1 Symmetric cryptography

1.1 Symmetric cryptography exercises

In this exercise, you'll use the OpenSSL command `enc` to encrypt a block of data. First, create a 32 hexadecimal digits long key (corresponding to 128 bits) and annotate it here: (16 Bytes=16*8 bits=128 bits=128/4 hex=32 hex)

→ NO! AGGIUNGE UN \n AL FONDO! <- openssl rand -out my128key -hex 16
openssl rand -hex 16 | tr -d '\n' > my128key
my128key = 8721a5c06b60eb796314773b7d2a2a39

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

echo "`This message is my great secret` " | tr -d '\n' > ptext

`This message is my great secret`

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

(It shows 33 B, you can't create a file of 1 B, for some reason? od -c nomefile shows that a \n was added)

Let's now encrypt with AES (with 128 bits key) the file `ptext` with the key you have previously annotated. Save the encrypted message in the `ctext.aes128` file. Find out and write in the box below the OpenSSL commands to perform this operation. Add the optional parameter that allows you to also show the use[illegible] and key.

openssl rand -hex 16 (aes block is 128 bits) -> iv = 9c0942a8836c16c35b0ba25752edaf69

→ openssl enc -aes128 -e -K 8721a5c06b60eb796314773b7d2a2a39 -iv 9c0942a8836c16c35b0ba25752edaf69 -in ptext -out ctext.aes128 -p
salt=3A9104D6797F0000
key=8721A5C06B60EB796314773B7D2A2A39
iv =9C0942A8836C16C35B0BA25752EDAF69
[AES128=AES128-CBC]

Now also find out the commands to encrypt the same data (`ptext`) with other symmetric algorithms (e.g. DES, 3DES, RC4). The encrypted messages must be saved in files named `ctext.`*algorithm*.

openssl rand -hex 8 | tr -d '\n' > my64key (des key is 64 bits) -> my64bitskey = e29daff3ffeb6beb
→ openssl rand -hex 8 (des block is 64 bits) -> iv = 97ab913fbfa3dd18
openssl enc -des -e -K e29daff3ffeb6beb -iv 97ab913fbfa3dd18 -in ptext -out ctext.des -p
salt=3A210ABFF07F0000
key=E29DAFF3FFEB6BEB
iv =97AB913FBFA3DD18
[DES=DES-CBC]

What else do you need to specify when you choose to use DES, 3DES and AES as encryption algorithms? Write down the possible choices (and commands).

→
-des -des-cbc
-des-cfb -des-cfb1 -des-cfb8
-des-ecb -des-ede -des-ede-cbc
-des-ede-cfb -des-ede-ecb -des-ede-ofb
-des-ede3 -des-ede3-cbc -des-ede3-cfb
-des-ede3-cfb1 -des-ede3-cfb8 -des-ede3-ecb
-des-ede3-ofb -des-ofb -des3 -des3-wrap

-aes-128-cbc -aes-128-cfb -aes-128-cfb1
-aes-128-cfb8 -aes-128-ctr -aes-128-ecb
-aes-128-ofb -aes-192-cbc -aes-192-cfb
-aes-192-cfb1 -aes-192-cfb8 -aes-192-ctr
-aes-192-ecb -aes-192-ofb -aes-256-cbc
-aes-256-cfb -aes-256-cfb1 -aes-256-cfb8
-aes-256-ctr -aes-256-ecb -aes-256-ofb
-aes128 -aes128-wrap -aes192
-aes192-wrap -aes256 -aes256-wrap

Compare the length of the generated `ctext.*` files against the length of the original `ptext`. Guess what happened.

→

-rw-r--r-- 1 root root 32 Nov 5 11:12 ptext
-rw-r--r-- 1 root root 48 Nov 5 11:15 ctext.aes128
-rw-r--r-- 1 root root 48 Nov 5 11:46 ctext.aes128-cbc
-rw-r--r-- 1 root root 48 Nov 5 11:21 ctext.aes256-cbc
-rw-r--r-- 1 root root 40 Nov 5 11:17 ctext.des
-rw-r--r-- 1 root root 40 Nov 5 11:19 ctext.des3
-rw-r--r-- 1 root root 32 Nov 5 11:23 ctext.rc4

Some added padding

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

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

If you tried to encipher a different plaintext message, use now the `ptext` file created before.

→ -rw-r--r-- 1 root root 32 Nov 5 11:47 ctext.aes128-cbc-nopad

Explain the difference between the encryption with the above command and the encryption with the RC4 algorithm.

→ 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 like RC4 and OpenSSL ignores it.

Find out the commands to encrypt the same data (`ptext`) with the algorithm ChaCha20 and a key of your choice. Explain the difference between the encryption with the above command and the encryption with the ChaCha20 algorithm.

It's the only stream cipher that supports a custom iv. 256 bits keys and 128 bits iv
→ openssl enc -chacha20 -e -K 20d5eae01ebaf30a4b7a78a81192523d54cf38708bf17f49333513e89977caa4 -iv 1efb56276d5b34755716624781039544 -in ptext -out ctext.chacha20 -p
salt=3A217094837F0000
key=20D5EAE01EBAF30A4B7A78A81192523D54CF38708BF17F49333513E89977CAA4
iv =1EFB56276D5B34755716624781039544

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

→ openssl enc -chacha20 -d -K 20d5eae01ebaf30a4b7a78a81192523d54cf38708bf17f49333513e89977caa4 -iv 1efb56276d5b34755716624781039544 -in ctext.chacha20 -out dtext.chacha20 -p
diff ptext dtext.chacha20
openssl enc -d -aes-128-cbc -K 8721a5c06b60eb796314773b7d2a2a39 -iv 8e0ede46dd6b672403b2efdbba29377a -in ctext.aes128-cbc -out dtext.aes128-cbc -p
diff ptext dtext.aes128-cbc

Verify whether the decryption has been performed correctly (check out the content of the files `dtext.rc4` and `dtext.aes128cbc`).

How does OpenSSL generate the key and the IV in the following command?

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

No salt:
key=[sha256(password)] first 128 bits truncated
iv=[sha256(password)] last 128 bits
https://linux.die.net/man/3/evp_bytestokey

Can you guess how the key and the IV have been derived from the passphrase?

→ openssl enc -e -in ptext -out ctext.aes128-cbc-passphrase -aes-128-cbc -p
enter aes-128-cbc encryption password: Alice
Verifying - enter aes-128-cbc encryption password: Alice
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
salt=3E7B7F77EC072394   (64 bit)
key=D1C89A31308AD5F7E08D5EDBCA32C4D0
iv =9DD425C58586652698A885F4C625123B

openssl enc -e -in ptext -out ctext.aes128-cbc-passphrase-nosalt -aes-128-cbc -nosalt -p
enter aes-128-cbc encryption password: Alice
Verifying - enter aes-128-cbc encryption password: Alice
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
key=3BC51062973C458D5A6F2D8D64A02324
iv =6354AD7E064B1E4E009EC8A0699A3043
openssl dgst -sha256 pass

1.2 Brute force attack

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

-rw-r--r-- 1 root root 48 Nov 5 11:15 ctext.aes128
-rw-r--r-- 1 root root 48 Nov 5 12:12 ctext.aes128-cbc
-rw-r--r-- 1 root root 32 Nov 5 11:47 ctext.aes128-cbc-nopad
-rw-r--r-- 1 root root 64 Nov 5 12:57 ctext.aes128-cbc-passphrase
-rw-r--r-- 1 root root 48 Nov 5 12:57 ctext.aes128-cbc-passphrase-nosalt
-rw-r--r-- 1 root root 48 Nov 5 11:21 ctext.aes256-cbc

ctext.aes128-cbc-passphrase is 16 hex longer (16*4=64bits=8Bytes)
because the first 8 bytes are used to store the hashed salt

Create two users, Alice and Bob, and a directory in which both of them can write using the following commands (you can find the equivalent script `create_users.sh` in `lab02_support.zip`):

```
adduser alice
passwd alice

adduser bob
passwd bob

mkdir /shared
groupadd aliceandbob
usermod -a -G aliceandbob alice
usermod -a -G aliceandbob bob
chgrp -R aliceandbob /shared
chmod -R 770 /shared
```

-a, **--append**
Add the user to the supplementary **group**(s). Use only with the **-G** option.
-G, --groups *GROUP1*[,*GROUP2*,...[,*GROUPN*]]]
A list of supplementary groups which the user is also a member of. Each group is separated from the next by a comma, with no intervening whitespace. The groups are subject to the same restrictions as the group given with the **-g** option.
If the user is currently a member of a group which is not listed, the user will be removed from the group. This behaviour can be changed via the -a option, which appends the user to the current supplementary group list.

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:

b

```
openssl enc -e -in ptext -out ctext_alice -K short_key -rc4
```

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

3. Bob tries to find out the key used by Alice to encrypt the message

> 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.

→
```
>openssl enc -d -in ctext_alice -out dtext_bob -K {0,1,..,f} -rc4
hex string is too short, padding with zero bytes to length
>cat dtext_bob
Ehi Bobbone, che fai stasera?
```
(With key=0 -> dtext empty, with other keys -> dtext gibberish)

Now take the file named `bruteforce_enc`, available in `lab02_support.zip`. It was encrypted by using the OpenSSL command executed above by Alice, that is with a hexadecimal character and with the OpenSSL command shown above. Have you encountered (any) difficulties in discovering the correct (encryption) key used? Why?

→ Yes, because the data that was encrypted was not an ASCII text or a common type of file, but it was just pure data, and we can't distinguish which one is the true one, because we do not know the context.
(It may be another binary key)

1.3 Performance

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
```

To evaluate the time required by the encryption operations, you can use the `time` command:

```
time openssl_encryption_command
```

Which is the most significant value among the times shown by the command `time`?

→ Bisogna valutare il campo user, perche` e il tempo passato in user mode, ` sys e il tempo passato in kernel ` mode e real e il wall-clock. The highest value is real time (for single CPU execution) while the most useful time to compare is user+sys which refers to the real cpu time used by the program

Now fill in 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	DECRYPT 100MB
DES-CBC	0.004	0.006	0.023	1.562	
DES-EDE3	0.004	0.006	0.052	3.931	
RC4	0.008	0.006	0.008	0.359	0.289
AES-128-CBC	0.004	0.004	0.010	0.347	
AES-192-CBC	0.005	0.005	0.009	0.318	
AES-256-CBC	0.003	0.003	0.007	0.396	0.245
ChaCha20	0.003	0.005	0.009	0.259	0.182

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

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

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

Which is the difference between using `time` and OpenSSL `speed` command to calculate the performance of an algorithm?

Which one (`time` vs. `speed`) should be preferred to measure the performance of a cryptographic algorithm?

→ Both are highly variable!

Doing aes-128 cbc for 3s on 16 size blocks: 13282475 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 64 size blocks: 3347798 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 256 size blocks: 909852 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 1024 size blocks: 256229 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 8192 size blocks: 26772 aes-128 cbc's in 2.98s
Doing aes-128 cbc for 3s on 16384 size blocks: 12923 aes-128 cbc's in 2.99s

block size	16 bytes	64 bytes	256 bytes	1024 bytes	8192 bytes	16384 bytes
des-cbc	69421.40k	70661.10k	72319.23k	72187.92k	72466.43k	72280.75k
aes-128-cbc	174794.75k	202956.89k	74436.88k	73045.33k	72376.32k	197985.13k

k=kBps encrypted ex. I can encrypt 72'376'320 Bytes of plaintext in 1s using aes-128-cbc with blocks of 8192B each.

2 Asymmetric cryptography

2.1 RSA Key generation

Starting from the OpenSSL command `genrsa`, discover and write down the 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):

→

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
```

What is the purpose of the parameters `modulus`, `publicExponent`, `privateExponent`, and `prime*`?

→

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)?

→

Compare the parameters with the ones generated by your colleague: do you notice any common parameters? If yes, which is their purpose?

→

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 Encryption and decryption operations with RSA asymmetric algorithm

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 http://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`, try to peek it by using the command

```
atril chap8.pdf &
```

2.3 Signing and verifying operations with RSA asymmetric algorithm

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`, discover and write down the 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):

[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 Signing and verifying operations with EC

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

Write down the OpenSSL `pkeyutl` command used to sign the file `plain` with ECDSA, and save the signature in the file `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

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

- RSA keys of 512, 1024, 2048 and 4096 bits in length.

  How does the performance decreases in terms of 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, 1024 bit DSA, 160 bit ECDSA (`ecdsap160`).

  What differences do you note? Are the elliptic curve cryptography operation more efficient than RSA/DSA operations? 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): 2048 bit RSA, 2048 bit DSA, 256 bit ECDSA (`ecdsap256`).

  NOTE

  In practice, 256 bit elliptic curve cryptography 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)?

→

- symmetric, asymmetric cryptography and digest: 2048 bit RSA, 2048 bit DSA, 128 bit AES and SHA-256.

NOTE

as above, 128 bit AES guarantees a security level equivalent to 3072-bit RSA/DSA.

How can you compare the results obtained?

How much faster are symmetric cryptography operations (aes-128-cbc, sha256) when compared to asymmetric operations (rsa2048, dsa2048)?

→

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 using MD5, SHA1, SHA-256, SHA3-256. Choose the output in binary format and save the results (i.e. the digests) in the files `MD5dgst`, `SHA1dgst`, `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)?

→

Suppose now that you want to calculate the hash of `msg` by using the the AES (symmetric) algorithm: how can you do this operation (think at the AES mode to use, the key, the IV if necessary)?

→

3.2 Collisions

In this exercise, you’ll have the possibility to test the robustness of a digest algorithm based on digest length. To this purpose you will use an “insecure” hash algorithm that we have created on purpose.

The file `IHA.sh`, present in the material provided for this lab, is an example of a digest algorithm than has been rendered insecure because the digest length is very short. More precisely, IHA produces a digest that is 4 bits long.

The file `IHA.sh` takes as the first parameter from the command line the name of the file containing the data on which the "insecure hash" will be calculated:

```
bash IHA.sh file_input
```

Calculate its digest with IHA.

Now you can try to find a collision: can you create another message that, if IHA is used, returns the same digest (as the one calculated above)?

How many IHA operations are required (on average) to find the collision, that is to perform the above operation (remember that the digest is 4-bits long)?

→

Suppose you want to find out an arbitrary collision, that is, to find any two messages whose digests are the same (in this case, the digest value to be matched is not fixed as above). How long should be the digest so that the trials required to find the arbitrary collision is equal to the time required to find the collision above in which the digest value was fixed. Why it is dangerous to have the same hash for two different data ?

→

3.3 Performance

Evaluate the cost associated to the digest algorithms implemented by OpenSSL, by using the method explained/used in Exercise 1.3. Fill in the results in the Table 2.

	100 B	10 kB	1 MB	100 MB
MD5	0.00s	0.00s	0.00s	0.17s
SHA1	0.00s	0.00s	0.00s	0.13s
SHA256	0.00s	0.00s	0.02s	0.28s
SHA512	0.00s	0.00s	0.01s	0.21s
SHA3-256	0.00s	0.00s	0.00s	0.30s
SHA3-512	0.00s	0.00s	0.01s	0.31s

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

Compare the results obtained in this exercise with the ones obtained for the symmetric encryption algorithms.

→

type	16 bytes	64 bytes	256 bytes	1024 bytes	8192 bytes	16384 bytes
md5	125630.59k	277246.31k	499334.55k	306179.77k	343587.98k	669574.08k
sha1	125306.52k	302186.07k	502604.54k	617593.51k	266234.56k	267621.72k
sha256	23731.82k	51548.20k	90198.70k	110408.36k	120984.92k	120324.10k
sha512	15383.87k	61897.83k	98182.51k	143789.40k	165923.50k	166401.37k

4 Additional exercises (optional)

Additional exercise with symmetric cryptography

4.1 Modifying a message encrypted with a symmetric cipher algorithm

In this exercise we will try to modify a message encrypted with symmetric cryptography. For this purpose, we will assume that two persons Alice and Bob communicate by exchanging encrypted messages as described below. Their communication is intercepted by Carlo, an active attacker who will also modify the encrypted (sniffed) messages. Immagine for example that Alice has to send to Bob a number, which is written in a text file. The number represents the amount of money that Alice asks to Bob to be sent to Carlo. Carlo wants to increase this amount, by performing the attack below.
Let's suppose Alice and Bob have a shared secret, like a symmetric key (that has been exchanged in a secure way). Alice creates the text file `numbers.txt` in which she saves only decimal numbers and encrypts it with the RC4 algorithm and the shared secret (the one shared with Bob) obtaining this the encrypted file `numbers.enc`.

```
echo 456789 > numbers.txt
openssl enc -rc4 -in numbers.txt -out numbers.enc -K shared_secret
```

The scenario we try to simulate is the following one. Alice sends `numbers.enc` to Bob. Carlo intercepts it, but before sending it to Bob he tries to modify it (Carlo does not know the shared secret). Bob received `numbers.enc` and tries to understand whether the message that he received from Carlo was modified or not.
In pratice, to simulate the attack:

1. open the file `numbers.enc` by using a hexadecimal editor, for example `hexedit`:

   ```
   hexedit numbers.enc
   ```

2. change some bits in `numbers.enc`, save it and exit from `hexedit` (press `Ctrl+X` to save and exit)

3. decrypt the file `numbers.enc` and check out its content. Is it still a decimal number ?

What can do Carlo to modify Alice's message, so that the modification goes unnoticed by Bob? In other words, what can do Carlo so that Bob (still) receives a decimal number (with high probability) even after the modification of Carlo?

→

What happens when you use AES-128-CBC instead of RC4? (try it).

→

4.2 Operation modes of symmetric block cipher algorithms

In this exercise we'll use the three files `ecb*` available in the archive. The first two files `ecb_plain_1` and `ecb_plain_2` contain two (possible) plaintext messages. The third file `ecb_enc` contains the ciphertext message obtained by encrypting one of the first two files with the following command:

```
openssl enc -e -in ecb_plain_X -out ecb_enc -K key -iv 0 -des-ecb
```

Can you find out which of the two plaintext messages has been encrypted by comparing the ciphertext message with the two plaintext messages?

→

Additional exercises with digest algorithms

4.3 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; for simplicity you could use the test script named `gen_tree.sh` in the support material.
Write down the command used to calculate the digest of all files contained in the directory `tree`. 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 sha1 -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:

→