Cryptography with OpenSSL – Applications

Laboratory for the class "Information Systems Security" (02TYMUV)
Politecnico di Torino – AA 2024/25
Prof. Antonio Lioy

prepared by:
Flavio Ciravegna (flavio.ciravegna@polito.it)
Andrea Atzeni (andrea.atzeni@polito.it)

v. 1.1 (15/11/2024)

Contents

Purpose of the laboratory

The goal of this laboratory is to experiment with the application of the main cryptographic primitives presented in the previous laboratory. The laboratory uses the OpenSSL (http://www.openssl.org/) open-source library and tools, available for various platforms, including Linux and Windows.

All the proposed exercises basically use the OpenSSL command line program, which allows the use of several cryptographic functions through the OpenSSL shell that can be started with the following command:

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

To perform the exercises of this lab, you will use the following OpenSSL commands already used in the laboratory 2:

```
dgst rand genrsa rsa pkeyutl enc
```

In addition, also the following new commands will be used:

```
crl x509 ocsp dhparam genpkey pkey
```

These commands are briefly described below. For the complete list of options and for a detailed description of the commands, please consult the corresponding `man` pages.

Additional files and programs required to run some of the exercises proposed below are available at:

`ISS_lab03_support.zip`.

To unzip this file, you can use the command:

```
7z x ISS_lab03_support.zip
```

openssl crl

The OpenSSL `crl` command allows to manipulate a Certificate Revocation List (CRL).
Its simplified syntax is:

```
openssl crl [-in file] [-CAfile file] [-text] [-noout]
```

where:

`-in` *file*, specifies that the file *file* contains the CRL to use;

`-CAfile` *file* specifies that the file *file* contains the certificates used to verify the CRL;

`-text` prints out the CRL in text form;

`-noout` indicates not to produce in output the Base64 representation of the CRL.

openssl x509

To sign and view an X.509 certificate, you can use the OpenSSL command `x509`. Actually, the `x509` command is a multi-purpose certificate utility: it can be used to display certificate information, convert certificates to various forms, sign certificate requests (behaving thus as a "mini CA"), or edit certificate trust settings. The simplified syntax of this command for the purpose of exercises proposed is:

```
openssl x509 [-inform DER|PEM] [-outform DER|PEM] [-in file] [-out file]
      [-noout] [-req] [-text]
```

where the main options have the following meanings:

`-inform DER|PEM` specifies the input format; normally the command will expect an X.509 certificate but this can change if other options (such as `-req`) are present. The DER value indicates that the input certificate is encoded with the DER encoding, PEM that the certificate is encoded in PEM, which is the base64 encoding of the DER encoding with header and footer lines added.

`-outform DER|PEM` specifies the output format (same possible values as with the `-inform` option).

`-in` *filename* specifies the input file to read a certificate from (standard input is used otherwise).

`-out` *filename* specifies the output file to write to (standard output is used otherwise).

`-noout` indicates not to produce in output the Base64 representation of the certificate.

`-text` prints out the certificate in text form. Full details are shown including the public key, signature algorithms, issuer and subject names, serial number any extensions present and any trust settings.

openssl ocsp

The validity of the certificate can also be verified with OCSP by using the OpenSSL `ocsp` command. The simplified syntax is:

```
openssl ocsp [-issuer file] [-cert file] [-url url] [-resp_text]
```

where:

`-issuer` *file*, specifies that the file *file* contains the current issuer certificate;

`-cert` *file* specifies that the file *file* contains the certificate that is added in the OCSP request;

`-url` *url*, specifies the URL of the responder host;

`-resp_text` indicates to produce in output the text form of the OCSP response.

openssl dhparam

The OpenSSL `dhparam` command allows the manipulation and generation of DH parameters. Its simplified syntax is:

```
openssl dhparam [-in filename] [-out filename] [-2] [-text] [numbits]
```

where:

- `-in` *filename* specifies the input file to read a certificate from (standard input is used otherwise);
- `-out` *filename* specifies the output file to write to (standard output is used otherwise);
- `-2` specifies to use 2 as generator;
- `-text` indicates to produce in output the text form of the DH parameters in a human-readable format;
- *numbits* specifies that a parameter set should be generated of size *numbits*. It must be the last option.

1 Cryptography applications: integrity

1.1 Message Authentication Codes

1.1.1 Manual generation of a keyed-digest

Let's suppose Alice wants to generate a keyed-digest of a message she wants to send to Bob. Alice creates a file named `msg` with a text message as below:

```
This is a text message, create a keyed-digest!
```

and creates a file `key` in which she saves the symmetric key that will be used to calculate the keyed-digest. Assume that Alice combines manually the key and the text to calculate the keyed-digest.

```
echo 012345 >key
cat msg >>data
cat key >>data
```

Write down the OpenSSL `dgst` that Alice could use to compute a keyed-digest on the file `msg`, using SHA-256 as base function:

→

What information needs to know Bob to verify this manually constructed keyed-digest? Which steps must be performed by Bob to verify the keyed-digest?

→

Why this way of constructing a keyed-digest is not secure? Explain what an attacker could do. Then, indicate a more secure way of constructing a keyed-digest

→

1.1.2 Computing an HMAC with OpenSSL

Run the following OpenSSL command to compute an HMAC (saved in `msg.hmac`) by using:

- the hash function SHA-256
- the key stored in the file `key`
- the message stored in the file `msg`

```
openssl dgst -sha256 -binary -hmac key -out msg.hmac msg
```

Now, assume that you are the verifier. You need to verify a message `msg` that has been protected with the `msg.hmac` HMAC calculated above. Write the OpenSSL command used to verify the keyed-digest generated above (hint: to compare two files, you can use the command `cmp`).

→

Next, let's check the protection that HMAC provides to the message(s).

Change one character in the content of the message `msg` and recompute the HMAC with the OpenSSL command. Is this HMAC different from the one you computed previously on the original message?

→

At this point, assume that an attacker has access to the original data stored in `msg`, either because it was intercepted while in transit in the communication between Alice and Bob or when it was stored on disk (e.g. in a local directory at Bob).

Assuming you use a secure hash function resistant to collisions, could an attacker modify the message `msg` and then recompute correctly an HMAC so that its modification cannot be detected by Bob? (look at the the optional exercise proposed in the previous laboratory where you computed the digest of folder trees with `hashdeep`.)

→

1.2 Digital signature

Suppose Alice wants to protect the authenticity and integrity of the messages sent to Bob with asymmetric cryptography. Alice owns a pair of RSA keys. You may use the RSA keypairs generated in the laboratory 2 or create a new pair of RSA keys with the following commands:

```
openssl genrsa -out rsa.key.alice 2048
openssl rsa -in rsa.key.alice -pubout -out rsa.pubkey.alice
```

Now, let's suppose Alice wants to apply a digital signature to a file. For instance, you can download a file from the Internet using the following command[1]:

```
wget https://cacr.uwaterloo.ca/hac/about/chap11.pdf
```

Run the following solution that allows Alice to sign the file `chap11.pdf` and Bob to verify the signature applied to it. This process leverages the commands `dgst` and `pkeyutl`:

- Alice→Bob: *rsa.pubkey.alice* (securely)
- Alice runs
  ```
  openssl dgst -sha256 -binary chap11.pdf > chap11.hash
  ```
- Alice runs
  ```
  openssl pkeyutl -sign -in chap11.hash -inkey rsa.key.alice -out chap11.sig
  ```
- Alice → Bob: `chap11.pdf`, `chap11.sig`
- Bob runs
  ```
  openssl dgst -sha256 -binary chap11.pdf > chap11.hashbob
  ```

[1]Alternatively, if you don't manage to download this file you can use the local file `/etc/apache2/apache2.conf`

- Bob runs

  ```
  openssl pkeyutl -verify -in chap11.hashbob -sigfile chap11.sig -inkey
  rsa.pubkey.alice -pubin
  ```

Note: Besides the command `pkeyutl`, you can directly use the command `dgst` for digital signature creation and verification (in practice, this is the easiest way to create/verify signatures with OpenSSL). For example, you can construct a digital signature of the file `chap11.pdf` with the following command:

```
openssl dgst -sha256 -sign rsa.key.alice -out chap11.sig chap11.pdf
```

Write down the `dgst` OpenSSL command to verify the signature:

→

Do you notice any difference when you use the `pkeyutl` command instead of the `dgst` command?

→

What happens if the attacker Chuck modifies one of the data blocks Alice sent to Bob? In practice, if Chuck modifies the original message (but not the signature), what will be the result of signature verification?

→

If Chuck modifies the signature (but not the data), what will be the result of signature verification?

→

What happens if Chuck replaces Alice's public key with his own public key and sends it to Bob claiming that it's Alice's public key? Can he replace the original data Alice created and the signature (created by Alice)?

What will be the result of the signature verification in this case?

→

2 Applications of asymmetric cryptography: key exchange/agreement

Imagine Alice and Bob want to exchange data encrypted with AES, but they do not share a secret key.

Using the OpenSSL commands you have learned for both symmetric and asymmetric operations, can you design a solution that would allow Alice to send an encrypted file to Bob without needing a pre-shared key? (pay attention: Bob must be able to correctly decrypt the ciphertext received from Alice).

Suppose Alice proceeds in the following way:

1. creates[2] (or downloads) a file of at least 1 MB, such as

```
wget https://cacr.uwaterloo.ca/hac/about/chap12.pdf
```

2. executes:

```
echo key > aeskey
openssl enc -aes-128-cbc -in chap12.pdf -out chap12.pdf.enc -kfile aeskey -iv 0
openssl pkeyutl -encrypt -in aeskey -inkey rsa.key.alice -out aeskey.encrsa
```

3. sends to Bob the following files:

   (a) *chap12.pdf.enc*

   (b) *aeskey.encrsa*

   (c) *rsa.key.alice*

Which serious errors did (inexperienced) Alice? (There are at least two!)

→

Now run the following protocol exploiting the OpenSSL commands `enc` and `pkeyutl`:

1. Bob (B) sends → Alice (A): *rsa.pubkey*
2. Alice: `echo "chiave" > aeskey`
3. Alice: `openssl enc -aes-128-cbc -in chap12.pdf -out chap12.pdf.enc -kfile aeskey`
4. Alice: `openssl pkeyutl -encrypt -in aeskey -pubin -inkey rsa.pubkey -out aeskey.enc`
5. Alice → Bob: *chap12.pdf.enc*, *aeskey.enc*
6. Bob: `openssl pkeyutl -decrypt -inkey rsa.key -in aeskey.enc -out aeskey`
7. Bob: `openssl enc -d -aes-128-cbc -in chap12.pdf.enc -kfile aeskey`

Has Bob managed to successfully recover the original plaintext (*chap12.pdf*)?

→

What kind of security attack could Chuck perform to intercept the data messages exchanged between Alice and Bob? How can you avoid this attack? (Hint: think about the security of the first step in the workflow above.)

→

Which other asymmetric algorithm could Alice and Bob use instead of RSA to perform the key exchange/agreement?

[2] You can use the `rand` command to generate a file big enough.

→

Considering that you identified the alternative asymmetric algorithm (instead of RSA) for key exchange/agreement, what kind of security attack could (still) Chuck do to intercept the data messages exchanged between Alice and Bob? How can you avoid this attack?

→

3 Digital certificates and public key infrastructures

3.1 X.509v3 certificates

The purpose of this part of the laboratory is to experiment with the format of an X.509 digital certificate, as well as with the most common methods used to check the revocation status, namely the CRL and the OCSP protocol.

3.1.1 Analyzing an X.509v3 certificate and the certificate chain

First, let's analyse an X.509 certificate. You could start from the file `myexample_cert.pem` provided with the lab material. Use the `x509` OpenSSL command to view its content:

```
openssl x509 -in myexample_cert.pem -text -noout
```

Take a look at its main fields and extensions.

Is this certificate valid?

→

Next, use your browser to navigate to `https://www.polito.it/`. Click the "lock" icon. Select "Connection secure" and then click on "More information". In the new window opened, click on "View Certificate". In this way, it is possible to inspect the certificate chain.

How many certificates are included in the chain, excluding the one for `www.polito.it`?

→

Who is the issuer of the certificate for `www.polito.it`? Write down the entire Distinguished Name (which is in the form CN= ..., O = ..., ST = ..., C = ...).

→

Which is the subject of the certificate for `www.polito.it`? Write down the entire Distinguished Name.

And what is the time validity for the certificate of `www.polito.it`?

→

Now, identify the issuer, the subject, and the time validity of the Root CA certificate in the certificate chain of www.polito.it.

NOTE

The Root CA certificate is the last in the chain, starting from the one for www.polito.it

→

Where can be found the Certificate Revocation List (CRL) for the certificate www.polito.it? Write down the entire URL from where the CRL for the certificate www.polito.it can be downloaded.

→

Next, download the CRL (save it in a file `downloadedCRL` and view its content with the `crl` command:

```
openssl crl -inform DER -in downloadedCRL -text
```

Where can be found the OCSP responder that may provide the status of the certificate for www.polito.it at the current time? Write down the URL of the OCSP responder providing the status for the certificate of www.polito.it.

→

3.1.2 Checking the certificate status with OCSP

In order to check a certificate status with OCSP, we need to download the current certificate for www.polito.it. To download this certificate, it is sufficient to open the browser, go to `https://www.polito.it`, and perform the steps explained before to find the certificate details. Once you reach the “View Certificate” window, go to the *Miscellaneous* section, search for the “Download” field, and click on “PEM (cert)”. It will start downloading the file containing the certificate. To check the status of the certificate in the file `www-polito-it.pem`, you need to first save the certificate of the issuer of www.polito.it (save the certificate of the issuer in a file named `www-polito-it-GEANT.pem`). Then, you can use the following command to check the status of the certificate of www.polito.it with OCSP:

```
openssl ocsp -issuer www-polito-it-GEANT.pem -cert www-polito-it.pem -url
http://GEANT.ocsp.sectigo.com/ -resp_text
```

4 Key exchange with DH

Now, let’s assume that Alice and Bob want to use the Diffie-Hellman algorithm to agree on a common secret key. You will use OpenSSL commands for this purpose.

The first step consists in generating public parameters for DH, by using the command:

```
openssl dhparam -out dhparams.pem -2 1024
```

By using the following command, you can see the generated parameters:

```
openssl dhparam -in dhparams.pem -text
```

Next, Alice and Bob will use the parameters to generate their own DH pair of public and private keys. Let's assume that this file is called `Alicedhkey.pem` (for Alice) and `Bobdhkey.pem` (for Bob). For this purpose, Alice and Bob run the following commands.

Alice:

```
openssl genpkey -paramfile dhparams.pem -out Alicedhkey.pem
```

Bob:

```
openssl genpkey -paramfile dhparams.pem -out Bobdhkey.pem
```

To view the keys just generated (e.g. for Alice), you can use the command:

```
openssl pkey -in Alicedhkey.pem -text
```

Next, Alice and Bob have to exchange their own public keys. Thus, each of them will have to extract the DH public key and save it in a file, named `Alicedhpub.pem` (for Alice) and `Bobdhpub.pem` (for Bob).

For this purpose, Alice and Bob run the following commands:

(Alice)

```
openssl pkey -in Alicedhkey.pem -pubout -out Alicedhpubkey.pem
```

(Bob)

```
openssl pkey -in Bobdhkey.pem -pubout -out Bobdhpubkey.pem
```

To view the content of the above files, you can use the command:

```
openssl pkey -pubin -in Alicedhpubkey.pem -text
```

Now, Alice and Bob can generate their shared key, by using these commands:

- (Alice)

  ```
  openssl pkeyutl -derive -inkey Alicedhkey.pem -peerkey Bobdhpubkey.pem -out
  Alice_secret.bin
  ```

- (Bob)

  ```
  openssl pkeyutl -derive -inkey Bobdhkey -peerkey Alicedhpubkey.pem -out
  Bob_secret.bin
  ```

Now compare the two generated keys with:

```
cmp -b Alice_secret.bin Bob_secret.bin
```

You shouldn't receive any output: this indicates that the two keys are identical. You can use the following commands to see that the two keys are identical:

```
xxd Alice_secret.bin
```

```
xxd Bob_secret.bin
```

5 Asymmetric challenge response authentication

In this exercise, you'll use the RSA functions (and the `pkeyutl` command) to implement a challenge-response protocol.

Form two groups (Alice and Bob) and proceed in the following way:

- Alice sends Bob her own public key `rsa.pubkey.alice` (if Alice and Bob are on the same PC it is not necessary to transfer any data, Alice just tells Bob the correct file on the file system to be used);
- Bob generates a random string in the file `random` and encrypts it with Alice's public key: the result obtained is the file `challenge`. Run the following OpenSSL commands required to perform this step, by using `rand` and `pkeyutl`:

```
openssl rand -out random 20
openssl pkeyutl -encrypt -pubin -inkey rsa.pubkey.alice -in random -out
challenge
```

- Bob sends the file `challenge` to Alice.
- Alice decrypts the challenge in the file `response` and sends it (or makes it available) to Bob. Use the following `pkeyutl` command to complete this step (the response to the challenge is saved in the file `response`):

```
openssl pkeyutl -decrypt -inkey rsa.key.alice -in challenge -out response
```

- Bob verifies that the response to the `challenge` is correct:

```
cmp response random
```

Is the `response` to the `challenge` correct?

→

Is this method subject to replay attack (explain why)?

→

Is this method subject to a sniffing attack (explain why)?

→

What if Bob is an attacker? What kind of attack could he do with the `challenge` and the `response` received from Alice (hint: we assume that Alice has not authenticated Bob)?

→

6 Authentication with passwords

6.1 Password hashing and dictionary attack

Storing passwords in clear within a database is a bad practice that leads to serious security concerns. This practice is equivalent to writing them down on a piece of digital paper. If an attacker breaks into the database, or rogue system administrators improperly access it, all credentials would be directly available. This will grant him access to all the users' accounts. This section aims at showing, at first, how to mitigate this issue, and then how to perform a dictionary attack over the password hashes.

6.2 Password hashing

Computing and storing a hash value from the password instead of saving it in cleartext is strongly suggested as a security best practice. For instance, we could compute the hash of a given password and store this value in the database along with other information such as the username. With OpenSSL, we could compute a password hash (in this case with SHA-256) using the following command:

```
echo -n "mypassword" | openssl dgst -sha256
```

When a user needs to be authenticated, the hash could be measured again (from the password provided by the user) and compared against the (hashed password) values stored in the database.

Why is this solution safer than the previous one?

→

Can you guess an attack which could still be effective in this case?

→

6.3 Dictionary attack

In this section, we show how to perform a dictionary attack. A dictionary attack uses a predefined collection of hashes and related passwords, precisely a dictionary, to quickly compare a specific password hash (or hashes) against those values. This hash could be obtained from a security breach or even stolen from a database. If the cracking software matches the input hash with one of the hashes in the dictionary, the attacker can successfully identify the original password.

Starting from a password hash that we may have stolen from a database, we will use the `hash_dict.py` script to search our dictionary and see if there is a match with a known password.

Launch the Python script `hash_dict.py` from the lab material to start the attack:

```
python3 hash_dict.py
```

the script will ask you to insert the password hash that you want to crack and the dictionary you want to rely on. In this case we leveraged SQLite for creating a database where we stored all the hashes calculated in advance and the related passwords. This database contains the table `dict_attack` where those information are retained. In the lab material, you can find an example of the dictionary (`dictionary.db`) that you could use for this exercise.

Use the hash `9fd13a8c6c17da6b9ed242f788efd8fe9fd5143e3444091b8cc5aa6a9c263114` and the provided dictionary and run the script. Did you manage to crack the password?

→

If you want to explore the dictionary content, you can use the `sqlite3` console as in the following:

```
sudo apt update && sudo apt install sqlite3 -y
sqlite3 dictionary.db -header -column "SELECT * FROM dict_attack LIMIT 10;"
```

you could change the LIMIT parameter in the query to see more or less rows.

You could try to search for other password hashes generating them with the command:

```
echo -n mypassword | openssl dgst -sha256
```

Did you find any match? Try to explain what is happening.

→

6.4 Password salting

In this exercise, we will explore the salting mechanism used commonly to protect from dictionary attacks. Form a group of two hosts, Alice and Bob.

On Bob (acting as a server) open a terminal and run the following command:

```
sudo adduser test1 --disabled-password
```

What have you done in the previous step (check `/etc/shadow`)?

→

Next, on Bob, run the command:

```
sudo mkpasswd --method=sha512crypt --salt=coolsalt 1234
```

You should see the following output in the terminal:

```
$6$coolsalt$XMeYB41McYDfApgGicyIRK7JC4I.wThWxLwwOSbW7HMFXJZJFxMdsShTIsxoiy/yG2BKqqD
IRH2Aasf/XDWks/
```

Open the /etc/shadow file:

```
sudo vi /etc/shadow
```

In the entry corresponding to the user 'test1', modify the row in the following way (copy and paste into the file the output of the previous command in the entry corresponding to the user test1):

```
test1:$6$coolsalt$XMeYB41McYDfApgGicyIRK7JC4I.wThWxLwwOSbW7HMFXJZJFxMdsShTIsxoiy/y
G2BKqqDIRH2Aasf/XDWks/:19674:0:99999:7:::
```

What have you done?

→

On Bob, start the ssh server with the command:

```
sudo systemctl start ssh
```

Now connect from Alice to Bob, by using ssh. On Alice, run the command:

```
ssh test1@IP_address_Bob
```

When asked, use the password '1234'.

Have you managed to connect via ssh to Bob's host?

→

On Bob, run next the following command:

```
sudo adduser test2
```

Insert the password '1234' when asked. Next, open `/etc/shadow` and check out the entry created for user test2. Which algorithm has been used for salting the password?

→

7 Additional exercises (optional)

Additional exercise with Authenticated Encryption

7.1 Authenticated Encryption with Associated Data (AEAD)

Now let's try to use the authenticated encryption (AEAD) method. Take a look at the cryptographic algorithms proposed by OpenSSL `enc` by using the following command:

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

Do you manage to identify any algorithm that allows you to use AEAD?

→

Do you manage to use the algorithms (allowing you to use AEAD) from the command line?

→

Unfortunately, as you have noticed, OpenSSL does not allow you to invoke the algorithms from the command line (our guess is that the API needs to be adapted to support AEAD). However, the AEAD primitives are available through the OpenSSL EVP programming interface.
To allow you to experiment with AEAD, we have written a Python script that implements AEAD with AES128 in GCM and whose syntax is:

```
./aes-gcm.py [-d|-e] plaintext associated_data ciphertext tag -K key -iv initialization_vector
```

This script always accepts four file names as input parameters. If the `-e` option is used, the script performs encryption and integrity tag computation, that is, it reads the `plaintext` file and saves the encrypted content in the the `ciphertext`, and reads the associated data from the `associated_data` file and saves in the `tag` file the authentication tag (computed over both `plaintext` and `associated_data`). If the `-d` option is used, the script performs decryption and tag verification. Specifically, it reads the `ciphertext` file and deciphers it into the `plaintext` file, and verifies the tag in the `tag` file against the tag which is computed by using the `associated_data` and the deciphered data.
More precisely, the meaning of all the script options is presented below:

- `[-e|-d]` allows you to choose the operation, `-e` to encrypt, `-d` to decrypt;
- *plaintext* is the name of the file containing the plaintext data to be encrypted (if `-e` is used), or the file where the decrypted data will be stored (if `-d` is used);
- *associated_data* is the name of the file containing the "associated data" to be used in the computation of the integrity tag together with the plain text (if `-e` is used) or to check the tag (if `-d` is used);

- *ciphertext* is the name of the file where the encrypted content will be saved (if `-e` is used) or from which it will be read (if `-d` is used);
- *tag* is the name of the file where the tag will be saved (if `-e` is used) or from which it will be retrieved to perform the verification (if `-d` is used);
- `-K` *key* where `key` is a string of 32 hexadecimal digits to be used for the encryption and for the MAC calculation;

  BEWARE

  The `-K` (uppercase) option does not accept a filename. Furthermore, the key must be 128 bits long (32 hex digits). You will receive a 'weird' error message (from the function `fromhex()`) if you write a wrong number of hex digits.

- `-iv` *initialization_vector* is the initialization vector.

`aes-gcm.py` is available in the support material of this laboratory. After you have unzipped it, you have to make it executable with the command `chmod +x aes-gcm.py`.
Now let's try to use it. Alice creates two files:

- `plain` where Alice saves the message to keep confidential, e.g. "*Bob, this message has to be kept secret*";
- `aad` where Alice saves the data to be only used for tag calculation, e.g. "*Secret message from Alice to Bob*";

Afterwards, Alice computes the tag and the ciphertext with the `aes-gcm` command and saves it in the files `tag` and `cipher`, respectively. Write the command to do it.

→

Now, let's try to verify how AEAD can be used to provide authenticity. Try to modify the plaintext and compute, with `aes-gcm.py`, both the `cipher2` and `tag2` files. Do you notice any difference in the tag and ciphertext files? (hint: open it with `hexeditor`).

→

Now, try to use the verification functions of the `aes-gcm.py` script. Use the original `aad` and `cipher` files together with `tag2`. Do you receive an error message?

→

Now let's try to change the file containing the associated data and save it into `add2`. Recompute the tag and the encrypted message (with the script above) into different files (e.g., `ciphertext3` and `tag3`) and indicate which file contains any difference (with respect to the ones you have initially computed). Then use the script to verify `aad2` with the originally generated `cipher` and `tag`. Does the script report modifications to the tag?

→

Finally, let's evaluate the performance of the AEAD mode and compare it with the performance of the encryption and digest algorithms evaluated in the laboratory 2 (sections 1.3 and 3.2).
Use the same files of size 100 B, 10 kB, 1 MB, and 100 MB that you created in the previous laboratory, or recreate them with the command:

```
openssl rand -out file num_bytes
```

Evaluate the *user time* with the following command:

```
time ./aes-gcm.py -e file associated cipher tag -K key -iv iv
```

In your opinion, is it efficient to use AEAD (instead of encrypting and calculating HMAC)? (OK, you are comparing OpenSSL performance with python libraries, nonetheless you can draw some conclusion . . .)

→