Lab Work 4

Data Protection
Master in Cyber Security (UPC) 2025
Jorge L. Villar

Last updated: Sep 1 13:22:12 2025

Contents

1 Key Generation with openssl

As described in previous lab work documents, the command line tool `openssl` can be used to generate keypairs (secret and public keys) for some public key cryptosystems like RSA or Diffie-Hellman. For the particular case of digital signatures, keys for RSA and DSA and Elliptic-curve based DSA (ECDSA) can be generated the same way as for RSA encryption or Diffie Hellman key agreement.

For instance, the command line

```
$ openssl genpkey -algorithm rsa > myRSApkey.pem
```

outputs a text encoding of an RSA keypair suitable both for RSA encryption and digital signature, and stores it into a PEM format file (see previous lab work documents for more details about command syntax and file formats).

Generating keypairs for Diffie Hellman key agreement or DSA signatures (either the plain version or the elliptic-curve based one) is a bit more involved, as the common public parameters have to be generated before the actual key generation process starts.

The following command example

```
$ openssl genpkey -genparam -algorithm ec -pkeyopt ec_paramgen_curve:P-256 > myECppar.pem
```

produces a file containing the public parameters corresponding to one NIST standard elliptic curve called "prime256v1" or simply "P-256". Next, a keypair for key agreement or digital signature can be generated from the public parameters with

(Generated by lineprocx v2.97)

```
$ openssl genpkey -paramfile myECppar.pem > myECpkey.pem
```

Public keys need to be detached from the keypair files with

```
$ openssl pkey -pubout -in myECpkey.pem > myECpubkey.pem
```

2 Signature Generation and Verification

Given a keypair `myECpkey.pem` and a document `mydoc.doc`, a digital signature can be generated by `openssl` (up to version 1.1.1) with the following command

```
$ openssl pkeyutl -inkey myECpkey.pem -sign -in digest.bin > sig.bin
```

where `digest.bin` is a binary file containing the hash (SHA-256) of the file to be signed. Thus, the complete signature generation procedure requires a previous command like

```
$ openssl dgst -sha256 -binary mydoc.doc > digest.bin
```

From version 3.0 on, a single command can perform the two actions by adding the option `-rawin` that tells `openssl` that the input file needs to be hashed before signing. Thus, for newer versions the command is

```
$ openssl pkeyutl -inkey myECpkey.pem -sign -rawin -in mydoc.doc > sig.bin
```

The contents and structure of the resulting binary file `sig.bin` depend on the particular digital signature algorithm used. In some cases (like in DSA or ECDSA) the signature itself is the aggregation of some mathematical objects. The way the different objects are encoded and aggregated into a single file is application dependent.

In the previous example the signature file (inspected with `xxd -p`) looks like

```
30450220491c9eb622fe80685ca5c8fdd90e98f15f6620c4f473494aef6b
3730b95a207b022100fb90133c6294d3356cf28c8e9f821c0f2c6503e15a
1d395cbcfdde555a0045a8
```

It is actually the ASN1 format aggregation of two integers:

```
30:45 (0x45 bytes = 552 bit sequence)
  02:20 (0x20 bytes = 256 bit integer)
    49:1c:9e:b6:22:fe:80:68:    5c:a5:c8:fd:d9:0e:98:f1:
    5f:66:20:c4:f4:73:49:4a:    ef:6b:37:30:b9:5a:20:7b
  02:21 (0x21 bytes = 264 bit integer)
    00:fb:90:13:3c:62:94:d3:    35:6c:f2:8c:8e:9f:82:1c:
    0f:2c:65:03:e1:5a:1d:39:    5c:bc:fd:de:55:5a:00:45:
    a8
```

The previous signature can be verified from the corresponding public key in a similar way, with

```
$ openssl pkeyutl -pubin -inkey myECpubkey.pem -verify -in digest.bin -sigfile sig.bin
```

where `digest.bin` contains the hash of the signed document, or for newer versions of OpenSSL, you can simply write

```
$ openssl pkeyutl -pubin -inkey myECpubkey.pem -verify -rawin -in mydoc.doc -sigfile sig.bin
```

The previous command outputs one of the two following lines, depending on the result of the verification:

```
Signature Verified Successfully
Signature Verification Failure
```

Since the keypair file also contains the public key, verification can alternatively be done directly from it, but then the option `-pubin` must be removed:

```
$ openssl pkeyutl -inkey myECpkey.pem -verify -in digest.bin -sigfile sig.bin
```

or

```
$ openssl pkeyutl -inkey myECpkey.pem -verify -rawin -in mydoc.doc -sigfile sig.bin
```

3 Certificate Management

OpenSSL provides a wide support for public key certificate management, including creation, certificate chain verification as well as certification authority (CA) and certification revocation list (CRL) functionalities.

A X.509 public key certificate is basically a document digitally signed by a certification authority (CA), that includes information about the certificate issuer (the CA), the subject identity and the subject's public key to be certified, among other things.

X.509 certificates are issued by a CA on request. Indeed, a subject must create certificate signature request (CSR) and send it to the CA. Then, the CA transforms the CSR inyto a certificate. Both the creation of the CSR (in PKCS#10 format) by the subject and the generation of an X.509 certificate from the CSR by the CA can be done using OpenSSL.

There are two types of public key certificates: end users' and CAs' certificates. They mainly differ in some properties like the purpose statement about the public key contained in the certificate. A CA's public key certificate can be used to sign other public key certificates (e.g. end users' ones) while end users' certificates cannot.

3.1 Obtaining Certificates from a CA

When an end user wants to obtain a public key certificate, first a CSR file must be created and submitted to the CA. The following command

```
$ openssl req -new -key myECpkey.pem -out cert_req.pem
```

asks the user in an interactive way to provide all the necessary identity information (including name, company, country, e-mail among other fields) and produces a CSR file that contains all the identity data and the public key to be certified. The CSR is also signed by its creator (and this is the reason why the command uses the keypair and not only the public key). This signature added tothe CSR, also known as a "self signature", is used to prove to the CA the knowledge of the secret key associated to the public key to be certified. This mechanism prevents the CA from signing bogus keys submitted by a malicious end user.

The contents of the generated CSR can be inspected with the command

```
$ openssl req -in cert_req.pem -text
```

The output produced is like

```
Certificate Request:
    Data:
        Version:  1 (0x0)
        Subject:  C=ES, ...  CN=Student001/emailAddress=student001@upc.edu
        Subject Public Key Info:
            Public Key Algorithm:  id-ecPublicKey
                Public-Key:  (256 bit)
                pub:
                    04:3a:ea:cf:93:42:f3:bd:70:c5:2f:48:49:13:03:
                    c8:b9:b3:9d:e3:ea:74:3c:bc:d4:6a:6d:e2:b4:14:
                    92:92:a9:8a:45:b0:6d:19:61:b5:86:3e:03:06:90:
                    54:68:4b:92:93:62:ce:c7:89:44:06:ff:13:05:a6:
                    43:89:00:0d:4e
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        Attributes:
            (none)
            Requested Extensions:
    Signature Algorithm:  ecdsa-with-SHA256
    Signature Value:
         30:44:02:20:61:14:d1:e8:c7:c7:ac:4b:aa:3f:12:92:ae:b6:
         60:c4:b4:88:4f:59:c5:ac:1f:24:0b:ad:00:b7:4e:f3:69:a7:
         02:20:54:b9:15:2b:fd:13:04:59:7a:61:54:58:6e:3a:7c:ca:
         4b:04:eb:85:7d:2d:98:13:87:25:4d:f7:c1:66:bf:e3
```

The CA (that is often split into two different entities: one for registration and another for certification) receives the CSR file, checks the identity information provided by the subject, and then produces the certificate by adding the issuer identification information, some extra fields, and a signature on all the data.

The following command does the conversion from the CSR to a certificate:

```
$ openssl x509 -in cert_req.pem -req -CA CAcert.pem -CAkey CApkey.pem -out mycert.pem
```

where `CApkey.pem` and `CAcert.pem` are the CA's keypair (necessary for signing the certificate) and the CA's public key certificate. CA's certificate is issued by a higher level CA in the trust hierarchy, or it is a self signed certificate if the CA is a root CA that everybody trusts.

The certificate contents can be inspected with

```
$ openssl x509 -in mycert.pem -text
```

producing, for example,

```
Certificate:
    Data:
        Version:  1 (0x0)
        Serial Number:  16803433828964055675 (0xe931ca09fe2f3a7b)
    Signature Algorithm:  ecdsa-with-SHA256
        Issuer:  C=ES, ...  CN=Jorge/emailAddress=jorge.villar@upc.edu
        Validity
            Not Before:  Nov 16 14:15:02 2023 GMT
            Not After :  Dec 16 14:15:02 2023 GMT
        Subject:  C=ES, ...  CN=Student001/emailAddress=student001@upc.edu
        Subject Public Key Info:
            Public Key Algorithm:  id-ecPublicKey
                Public-Key:  (256 bit)
                pub:
                    04:3a:ea:cf:93:42:f3:bd:70:c5:2f:48:49:13:03:
                    c8:b9:b3:9d:e3:ea:74:3c:bc:d4:6a:6d:e2:b4:14:
                    92:92:a9:8a:45:b0:6d:19:61:b5:86:3e:03:06:90:
                    54:68:4b:92:93:62:ce:c7:89:44:06:ff:13:05:a6:
                    43:89:00:0d:4e
                ASN1 OID: prime256v1
                NIST CURVE: P-256
    Signature Algorithm:  ecdsa-with-SHA256
         30:45:02:20:25:73:83:22:e1:de:6f:7a:73:e8:8a:25:8a:56:
         66:04:97:d9:73:15:50:b4:1d:54:d6:13:86:25:3c:44:9c:b7:
         02:21:00:ac:16:e1:80:e9:26:42:f2:b9:67:a2:37:5c:0e:82:
         cb:ca:0f:ce:17:0c:2a:2d:0e:33:af:e9:e9:f5:37:ae:f1
```

3.2 Certificate Validation

A public key certificate can be validated by verifying all the signatures contained in all certificates in the trust chain (from the end user's certificate to the self signed root CA certificate). The following command checks the validity of a peer's certificate (for simplicity, directly issued by the root CA, without any intermediate CAs):

```
$ openssl verify -CAfile CAcert.pem -check_ss_sig peercert.pem
```

For longer certificate chains (i.e. when there are some intermediate CAs) the intermediate CA certificates must be aggregated into a single file and provided to the `openssl` call with a special option (see the detailed documentation of the `openssl verify` command).

For testing purposes, OpenSSL allows the generation of self signed CA certificates with the command

```
$ openssl req -x509 -new -key myECpkey.pem -out myselfCAcert.pem
```

that also asks to interactively provide all the necessary identity information.

Since the certified public key is part of any certificate, it can be extracted with

```
$ openssl x509 -in mycert.pem -pubkey -noout > extractedpubkey.pem
```

then recovering the same information as in file `myECpubkey.pem`.

For the same reason, the certificate file can replace the public key file in the signature verification command (the option `-pubin` is now replaced by `-certin`), as in

```
$ openssl pkeyutl -certin -inkey mycert.pem -verify -in digest.bin -sigfile sig.bin
```

or for newer versions of OpenSSL, simply

```
$ openssl pkeyutl -certin -inkey mycert.pem -verify -rawin -in mydoc.doc -sigfile sig.bin
```

For security reasons, before any use of a received peer's public key, the corresponding certificate chain must be verified as explained above.

3.3 Certificate Revocation

Certificates can be revoked by many reasons before their expiration date. For this reason, the public key certificate verification process must ensure that none of the certificates in the chain has been revoked. AC usually publishes Certificate Revocation Lists (CRL) in a periodic basis (there are other mechanisms like the Online Certificate Status Protocol (OCSP) that will probably supersede CRLs).

A CRL is mainly a list of revoked certificates signed by the issuing CA. OpenSSL provides mechanisms to create, maintain and check CRLs, mainly intended to be run by a CA. From the point of view of an end user, a CRL must be verified (i.e., the user must verify the CA's signature on the CRL information), and the user must always use the CRL to check the revocation status of all certificates in a certificate chain before using a peer's public key.

The following command can be used to inspect a CRL file:

```
$ openssl crl -in crl001.pem -text
```

A toy example of the output (a CRL with just one revoked certificate) is:

```
Certificate Revocation List (CRL):
        Version 2 (0x1)
    Signature Algorithm:  ecdsa-with-SHA256
        Issuer:  /C=ES .../CN=Jorge/emailAddress=jorge.villar@upc.edu
        Last Update:  Nov 16 16:39:04 2023 GMT
        Next Update:  Dec 16 16:39:04 2023 GMT
        CRL extensions:
            X509v3 CRL Number:
                2
Revoked Certificates:
    Serial Number:  E931CA09FE2F3A7B
        Revocation Date:  Nov 16 16:37:13 2023 GMT
```

```
        CRL entry extensions:
            X509v3 CRL Reason Code:
                Key Compromise
    Signature Algorithm:  ecdsa-with-SHA256
         30:46:02:21:00:fe:e8:b4:ef:6b:66:5e:4b:c5:e4:27:44:fe:
         40:70:c0:79:9a:de:ae:df:18:3b:7a:22:23:8a:e4:d9:bd:32:
         8c:02:21:00:df:83:f7:6d:61:40:48:30:11:ef:64:d2:a4:03:
         ce:36:fd:5a:11:fe:17:5d:f4:10:f1:29:89:88:54:41:1e:78
```

The CRL file itself can be verified with:

```
$ openssl crl -in crl001.pem -CAfile CAcert.pem -noout
```

where `CAcert.pem` is the issuing CA's certificate.

The complete verification of a peer's certificate taking into account the CRL is

```
$ openssl verify -CAfile CAcert.pem -check_ss_sig -CRLfile crl001.pem -crl_check peercert.pem
```

If there are some intermediate CAs involved in the verification, their certificates must also be checked for possible revocations, and then the option `-crl_check` in the previous command must be replaced by `-crl_check_all`.

4 Practical Work: ECDSA Signatures and Certificates

In this lab I will play the role of a (manual) root CA. Hence, you will generate keypairs and ask me to certify them. I will also provide some other peers' certificates and signatures, and publish a CRL (but some of the information could have been corrupted!).

You must analize the certificates and signatures and write a report with the results describing the corruptions found. You must also generate a digital signature of your report with the above certified key. Namely, you have to follow the steps in this list:

1. Create a public parameters file for NIST elliptic curve P-256, as given in the example above, and generate a keypair to be used for ECDSA signatures.
2. Create a CSR for the generated keypair, providing some identification information (country, province, city, name and email of one of the team's member). Because of the policy implemented in the CA it is mandatory the use of the fields Country=ES, Province=Barcelona, City=Barcelona in order to be able to issue a valid certificate. You can set the organization field to "UPC" and the entity to "DPROT".
3. Send me the CSR by email with the subject "DPROT:CSR" (one per lab team). If the CSR is correct, then I will replay the message with the corresponding public key certificate.
4. Download my public key root CA certificate (it is self-signed), the CRL lists and all the other certificate, document and signature files from the auxiliary files section of the lab work.
5. Verify the validity of all this material (CRL files, certificates and signatures) and write a report with the details of all verifications done. Each of the 3 sets of files contains the corresponding README file containing some extra information about the file set.
6. Digitally sign the lab report and send me by email both the report file and the generated signature. I will accept or reject the files based on the validity of the digital signature.