Public key cryptography in OpenSSL

HW6/7 - CNS Sapienza

Valerio Coretti 1635747

2019-12-19

1 Introduction

When we talk about cryptography we have to distinguish two types: *symmetric cryptography* and *asymmetric cryptography*. In this paper, we will discuss the latter. This is also named *Public-Key cryptography*, because it uses a public key and a private key. Such keys are generated by cryptographic algorithms that produce one-way functions. In the systems that implement this type of cryptography, we can encrypt a message using the receiver's public key, but that encrypted message can only be decrypted with the receiver's private key.

With asymmetric cryptography, a robust system of *authentication* is also possible. In this case, the sender can combine the message with a private key to create a *digital signature* on the message.

One issue could be the authenticity of a public key, i.e. that it is correct and belongs to the person or entity claimed, and has not been replaced by a malicious third party. To solve this problem was created a public key infrastructure (PKI), in which one or more third parties, known as *certificate authorities* (CA), certify ownership of key pairs.

In modern cryptosystems, the best algorithm to do both confidentiality, with asymmetric encryption/decryption, and authenticity, with the digital signature, is *RSA*.

In the following sections, we will present to the reader a guide on how to use the main functions of asymmetric cryptography in OpenSSL. Finally, we will show an example where, with two virtual machines, we will emulate the exchange of messages between an entity and a CA.

2 Public-key cryptography in OpenSSL

In this section we will show the main shell command to apply the functions of the asymmetric cryptography in OpenSSL. In particoular we will pay attention in four main functions: *keys generation*, *certificate generation* and *verification*, *digital signature*.

2.1 Keys generation

The first step is the generation of the public and the private keys. To do this we use the command:

```
openssl genpkey -algorithm rsa -pkeyopt
rsa_keygen_bits:2048 -out keys.pem
```

this command means that we want to generate the two keys, in particular, we want a private key of 2048 bits, with RSA algorithm (we can drop the -algorithm rsa flag in this example because genpkey defaults to the type RSA) and we save the output in the file *keys.pem*. Privacy Enhanced Mail (PEM) extension is customary for the default PEM format (OpenSSL has commands to convert among formats if needed). Now to see the result of the output:

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

this command shows us not only the generated key but also the main parameters used by the algorithm, i.e. the two prime numbers with relative exponent, the modulus, etc...

Now we have saved our keys in *keys.pem*, but to start the communication with another entity we want our public key in a different file because we have to share it:

```
openssl pkey -in keys.pem -pubout -out public.pem
```

with these three simple steps, we have created our keys. One issue in this commands is that we save our keys in clear text, for this reason, a better way is to encrypt the keys:

```
openssl pkey -in keys.pem -des3 -out private.pem
```

This command asks us a passphrase to do the encryption. Every time in the future that we have to use the *private.pem* encrypted keys, OpenSSL will ask us the passphrase.

2.2 Digital signature

With asymmetric cryptography, we can sign and verify a file. Let's think the sender entity sign a file then the receivers have to be able to check that the sender was the one who has signed the file. Now that we have the key pairs, OpenSSL allows us to sign a file with the following commands:

```
openssl dgst -sha256 -sign keys.pem -out sign.sha256 msg
```

With this command, we have signed a message *msg* with our keys and we write the output in a file called *sign.sha256*. To get a readable version of this file, the follow-up command is:

```
openssl enc -base64 -in sign.sha256 -out sign.sha256.base64
```

Now we will see how a receiver verify the signed file. There are two OpenSSL commands used for this purpose. The first decodes the base64 signature:

```
openssl enc -base64 -d -in sign.sha256.base64
-out sign.sha256
```

The second verifies the signature:

```
openssl dgst -sha256 -verify keys.pem -signature sign.sha256
```

If everything went well the output from this second command is: *Verified OK*. Notice that we verify the signed file with the public key.

2.3 Digital Certificates

Digital certificates bring together all the pieces: key pairs, digital signatures, and encryption/decryption. First of all, to generate a certificate, we have to create a certificate signing request (CSR), which is then sent to a certification authority (CA). To do this in OpenSSL we run this command:

```
openssl req -out req.csr -new -newkey rsa:4096
-nodes -keyout keys.pem
```

This example generates a CSR document and stores it in req.csr (base64 text). The CSR document requires the CA to guarantee the identity associated with the specified domain name, also called the common name (CN). A new key pair also is generated by this command, although an existing pair could be used. Furthermore to show and verify req.csr run the command:

```
openssl req -in req.csr -noout -text -verify
```

2.3.1 Self–signed certificate

If we are developing an HTTPS web site, it is convenient to have a digital certificate without going through the CA process. This type of certificate is also called *self-signed certificate*. OpenSSL allow us to create it with a very simple command:

```
openssl req -x509 -sha256 -nodes -days 365 -newkey
rsa:4096 -keyout keys.pem -out certificate.crt
```

Now we have our self–signed certificate valid for 365 days and with an RSA public key. The OpenSSL command below presents a readable version of the generated certificate:

```
openssl x509 -in certificate.crt -text -noout
```

The digital certificate contains the exponent and modulus values that make up the public key. These values are part of the key pair in the originally generated PEM file (keys.pem). The modulus from the key pair should match the modulus from the digital certificate. OpenSSL provides two commands that check for the same modulus, thereby confirming that the digital certificate is based upon the key pair in the PEM file:

```
openssl x509 -noout -modulus -in certificate.crt | openssl
sha1
openssl rsa -noout -modulus -in keys.pem | openssl sha1
```

If the resulting hash values match, we are sure that the digital certificate is based upon the specified key pair.

2.3.2 Convert Certificate Formats

Until now, all the certificates we have seen, are x509 which is *ASCII PEM* encoded. But there are different types of formats and for this reason, in this section, we will see how to convert a certificate in another type. This is important because, for example, some applications prefer certain formats over others.

The main formats are: *pem*, *der*, *pkcs7*, *pkcs12*.

- **DER** format is typically used with Java, we do the convertion with:

  - *PEM to DER*

```
openssl x509 -in domain.crt -outform der -out domain.der
```

- *DER to PEM*

```
openssl x509 -inform der -in domain.der -out domain.crt
```

- **PKCS7** files, also known as P7B, are typically used in Java Keystores and Microsoft IIS (Windows). They are ASCII files that can contain certificates and CA certificates.

  - *PEM to PKCS7*

  ```
  openssl crl2pkcs7 -nocrl -certfile domain.crt -certfile
  ca-chain.crt -out domain.p7b
  ```

  Note that you can use one or more −certfile options to specify which certificates to add to the PKCS7 file.

  - *PKCS7 to PEM*

  ```
  openssl pkcs7 -in domain.p7b -print_certs -out domain.crt
  ```

  Note that if your PKCS7 file has multiple items in it (e.g. a certificate and a CA intermediate certificate), the PEM file will contain all of the items in it.

- **PKCS12** files, also known as PFX files, are typically used for importing and exporting certificate chains in Microsoft IIS (Windows).

  - *PEM to PKCS12*

  ```
  openssl pkcs12 -inkey domain.key -in domain.crt -export
  -out domain.pfx
  ```

  You will be prompted for export passwords, which you may leave blank. Note that you may add a chain of certificates to the PKCS12 file by concatenating the certificates together in a single PEM file (domain.crt) in this case.

  - *PKCS12 to PEM*

  ```
  openssl pkcs12 -in domain.pfx -nodes -out domain.combined.crt
  ```

  Note that if your PKCS12 file has multiple items in it (e.g. a certificate and private key), the PEM file that is created will contain all of the items in it.

3 Practice with PKey cryptography in OpenSSL

In this section, we will simulate an exchange of information between a user (e.g. Alice) and a Certification Authority. For our purpose, we create two virtual machines with *Docker* and connect them with *Netcat*. With this configuration, Alice will be able to send requests to CA to sign or revoke certificates, and CA can send back the responses.

3.1 Preprocessing

In this part, we see the commands to generate the virtual machine and how to configure it for our purpose. Then, in the subsequent sections, we will see how to configure the CA machine and the main netcat scripts used for the communication.

3.1.1 Docker Machines creation

Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. In this section we will show the main steps to create a simple VMs. We create two *Ubuntu VMs* in this way:

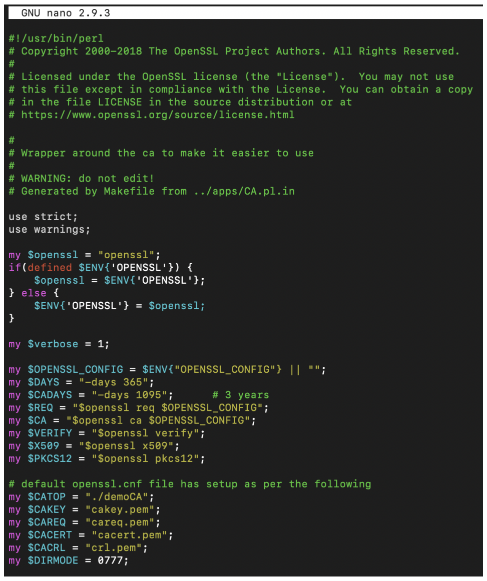
```
docker pull ubuntu
docker run -it ubuntu
apt-get update                /* Inside the VMs shell */
apt-get install openssl netcat /* Inside the VMs shell */
```

These run our VMs and install the tools needed.

3.1.2 Configuration of a CA

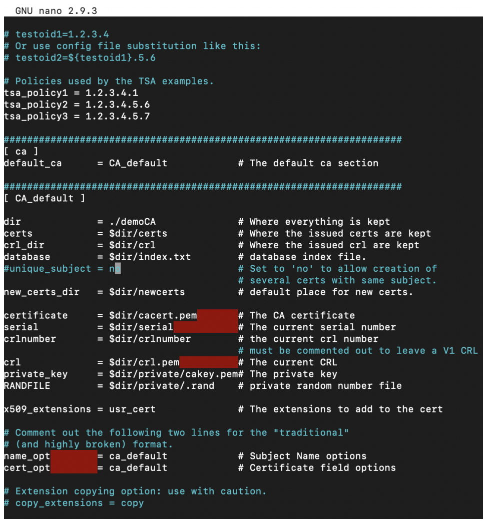
Here we will set up the configuration files in the machine corresponding to the CA. We can manage easily the CA using two files: *CA.pl* and *openssl.cnf*. In this files, we have some configurations parameters of the CA but what we need is only the *CATOP* that is the relative path where the certification authority will be, and we will change it with an absolute path: `./demoCA => /root/demoCA`.

We can find the first file in: `/usr/lib/ssl/misc/CA.pl`.



```
GNU nano 2.9.3

#!/usr/bin/perl
# Copyright 2000-2018 The OpenSSL Project Authors. All Rights Reserved.
#
# Licensed under the OpenSSL license (the "License").  You may not use
# this file except in compliance with the License.  You can obtain a copy
# in the file LICENSE in the source distribution or at
# https://www.openssl.org/source/license.html

#
# Wrapper around the ca to make it easier to use
#
# WARNING: do not edit!
# Generated by Makefile from ../apps/CA.pl.in

use strict;
use warnings;

my $openssl = "openssl";
if(defined $ENV{'OPENSSL'}) {
    $openssl = $ENV{'OPENSSL'};
} else {
    $ENV{'OPENSSL'} = $openssl;
}

my $verbose = 1;

my $OPENSSL_CONFIG = $ENV{"OPENSSL_CONFIG"} || "";
my $DAYS = "-days 365";
my $CADAYS = "-days 1095";      # 3 years
my $REQ = "$openssl req $OPENSSL_CONFIG";
my $CA = "$openssl ca $OPENSSL_CONFIG";
my $VERIFY = "$openssl verify";
my $X509 = "$openssl x509";
my $PKCS12 = "$openssl pkcs12";

# default openssl.cnf file has setup as per the following
my $CATOP = "./demoCA";
my $CAKEY = "cakey.pem";
my $CAREQ = "careq.pem";
my $CACERT = "cacert.pem";
my $CACRL = "crl.pem";
my $DIRMODE = 0777;
```

Now we have to make the same change also in the openssl configuration file, in `/usr/lib/ssl/openssl.cnf`



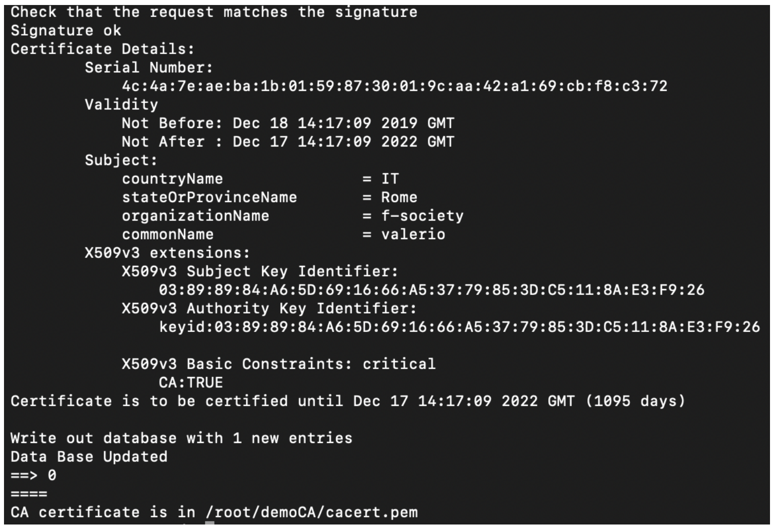
```
GNU nano 2.9.3
# testoid1=1.2.3.4
# Or use config file substitution like this:
# testoid2=${testoid1}.5.6

# Policies used by the TSA examples.
tsa_policy1 = 1.2.3.4.1
tsa_policy2 = 1.2.3.4.5.6
tsa_policy3 = 1.2.3.4.5.7

####################################################################
[ ca ]
default_ca      = CA_default            # The default ca section

####################################################################
[ CA_default ]

dir             = ./demoCA              # Where everything is kept
certs           = $dir/certs            # Where the issued certs are kept
crl_dir         = $dir/crl              # Where the issued crl are kept
database        = $dir/index.txt        # database index file.
#unique_subject = no                    # Set to 'no' to allow creation of
                                        # several certs with same subject.
new_certs_dir   = $dir/newcerts         # default place for new certs.

certificate     = $dir/cacert.pem       # The CA certificate
serial          = $dir/serial           # The current serial number
crlnumber       = $dir/crlnumber        # the current crl number
                                        # must be commented out to leave a V1 CRL
crl             = $dir/crl.pem          # The current CRL
private_key     = $dir/private/cakey.pem# The private key
RANDFILE        = $dir/private/.rand    # private random number file

x509_extensions = usr_cert              # The extensions to add to the cert

# Comment out the following two lines for the "traditional"
# (and highly broken) format.
name_opt        = ca_default            # Subject Name options
cert_opt        = ca_default            # Certificate field options

# Extension copying option: use with caution.
# copy_extensions = copy
```

In these files, we find also the main pieces of information about the CA like the duration of the CA and the generated certificates.

Now we are ready to create our CA with the following command:

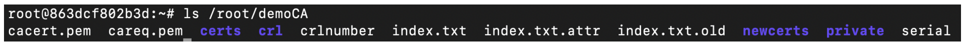
```
/usr/lib/ssl/misc/CA.pl -newca
```

Following all the steps we have this results:



```
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number:
            4c:4a:7e:ae:ba:1b:01:59:87:30:01:9c:aa:42:a1:69:cb:f8:c3:72
        Validity
            Not Before: Dec 18 14:17:09 2019 GMT
            Not After : Dec 17 14:17:09 2022 GMT
        Subject:
            countryName               = IT
            stateOrProvinceName       = Rome
            organizationName          = f-society
            commonName                = valerio
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                03:89:89:84:A6:5D:69:16:66:A5:37:79:85:3D:C5:11:8A:E3:F9:26
            X509v3 Authority Key Identifier:
                keyid:03:89:89:84:A6:5D:69:16:66:A5:37:79:85:3D:C5:11:8A:E3:F9:26

            X509v3 Basic Constraints: critical
                CA:TRUE
Certificate is to be certified until Dec 17 14:17:09 2022 GMT (1095 days)

Write out database with 1 new entries
Data Base Updated
==> 0
====
CA certificate is in /root/demoCA/cacert.pem
```

This commands generate the CA certificate in the path `/root/demoCA/cacert.pem`. Meanwhile the encrypted private key is saved in the path: `/root/demoCA/private/cakey.pem`. Ok now we have a certification authority that is able to validate and revoke certificates. But after to start the communication analyze a bit the folder *demoCA*:



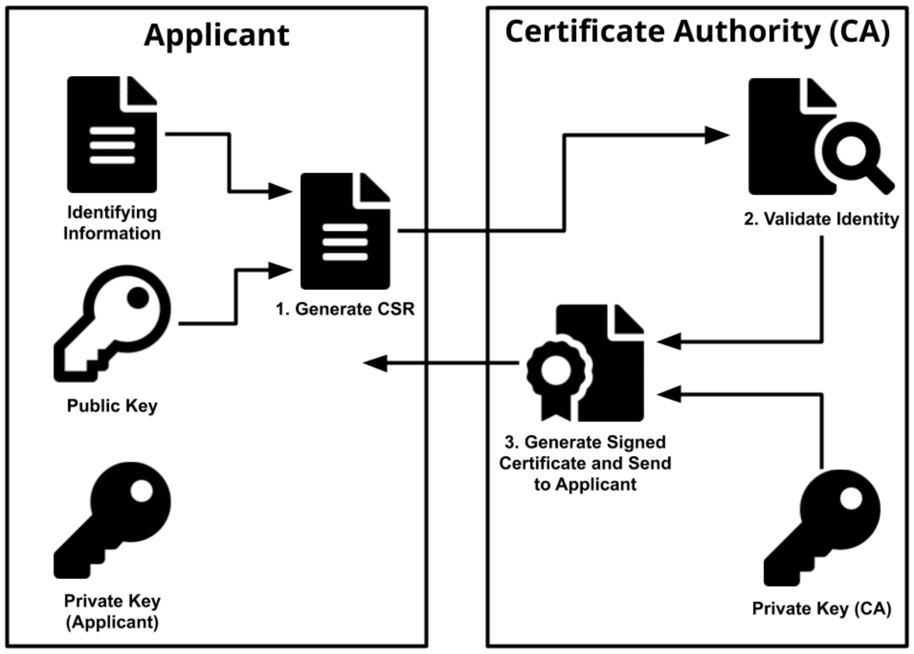
```
root@863dcf802b3d:~# ls /root/demoCA
cacert.pem  careq.pem  certs  crl  crlnumber  index.txt  index.txt.attr  index.txt.old  newcerts  private  serial
```

this is the structure and this is important to know it because all the certificates will be saved in this folder: *careq.pem* is the request for itself to subsign the certificate, *certs* directory has the trusted certificate, *crl* and *crl-number* are the certification revocation list folder and number, *index.txt.att* is the database of certificates, in *newcerts* will be starting the new certificates that we validate from this CA, *private* folder has the key pair, and finally *serial* is the serial number of the CA.

3.2 Experimentation

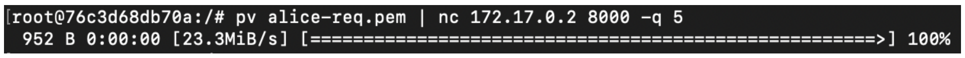
In the next steps, we simulate the request to sign a certificate and the request to validate a certificate.

3.2.1 Certificate signing request (CSR)



What we do in this section is represented in the figure above. We have our VM with a CA configured and on the other side have Alice's VM. The following are the main steps:

1. CA wait for a request:
   `$ nc -l -p [CA port] > alice-req.pem`
2. Alice generate a Certificate signing request (CSR) and send it to CA:
   ```
   $ openssl req -new -keyout alice-keys.pem -out alice-req.pem
   $ pv alice-req.pem | nc [CA ip] [CA port] -q 5
   ```



```
root@76c3d68db70a:/# pv alice-req.pem | nc 172.17.0.2 8000 -q 5
 952 B 0:00:00 [23.3MiB/s] [================================================>] 100%
```

3. CA check that the request matches the signature, sign the certificate and finally add it to the database:
   ```
   $ openssl ca -in alice-req.pem
   ```
   now this generates a new certificate in the folder `/demoCA/newcerts` where the certificates are named automatically by the CA with a hexadecimal number. The last generated certificate is the one with the biggest number.

4. Alice wait for the signed certificate:
   ```
   $ nc -l -p [Alice port] > alice-cert.pem
   ```

5. CA send back the signed certificate:
   ```
   $ pv alice-cert.pem | nc [Alice ip] [Alice port] -q 5
   ```

3.2.2 Certificate revocation

For any reason, a CA can decide to revoke a certificate before its expiry. In OpenSSL this procedure is very simple, in fact in the CA VM just run the commands:

1. CA revokes the certificate:
   ```
   $ openssl ca -revoke demoCA/newcerts/number_of_cert.pem
   ```

2. CA updates the certificates revocation list:
   ```
   $ openssl ca -gencrl -out demoCA/crl/crl.pem
   ```

3.2.3 Certificate validation

The last step we will face is the verification of the certificate. This procedure is very important for a secure communication. The simple way to do this is:

1. CA wait for a request:
   ```
   $ nc -l -p [CA port] > alice-req.pem
   ```

2. Alice send her own certificate to the CA:
   ```
   $ pv alice-cert.pem | nc [CA ip] [CA port] -q 5
   ```

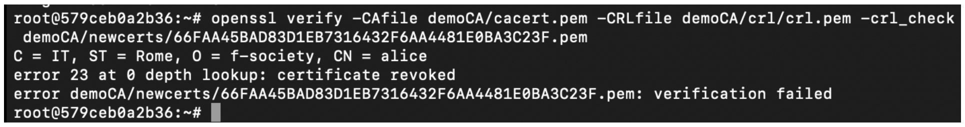
3. CA verifies the certificate trough its own certificate:
   ```
   $ openssl verify -CAfile demoCA/cacert.pem
   demoCA/newcerts/alice-cert.pem
   ```

If the verification successed then we have the output:
`demoCA/newcerts/alice-cert.pem:  OK`.

But this verification is not complete because it verifies only the validation of the signature and the deadline of the certificate. The issue is that even if this last two are correct the certificate could be not valid because it could be revocated in the past. For this reason, we change this last command to verify also the revocation:

```
$ openssl verify -CAfile demoCA/cacert.pem -CRLfile demoCA/crl/crl.pem
-crl_check demoCA/newcerts/alice-cert.pem
```

If this verification succeeds the certificate is valid and not revocated, instead if the certificate has been revocated the output will be like that:



```
root@579ceb0a2b36:~# openssl verify -CAfile demoCA/cacert.pem -CRLfile demoCA/crl/crl.pem -crl_check
 demoCA/newcerts/66FAA45BAD83D1EB7316432F6AA4481E0BA3C23F.pem
C = IT, ST = Rome, O = f-society, CN = alice
error 23 at 0 depth lookup: certificate revoked
error demoCA/newcerts/66FAA45BAD83D1EB7316432F6AA4481E0BA3C23F.pem: verification failed
root@579ceb0a2b36:~#
```

We end here our adventure with public key cryptography with OpenSSL.

References

[1] *OpenSSL Documentation.*
`https://www.openssl.org/docs/manmaster/man3/`

[2] *Docker Documentation*
`https://docs.docker.com`

[3] *Cryptography basics in OpenSSL.*
`https://opensource.com/article/19/6/cryptography-basics-openssl-part-2`

[4] *Using Netcat for File Transfers*
`https://nakkaya.com/2009/04/15/using-netcat-for-file-transfers/`

[5] *Create and manage certificates and certification authorities*
`https://www.youtube.com/watch?v=uo_xz6sA4dY&t=11s`