Public key cryptography in OpenSSL

HW6-CNS Sapienza

Manuel Ivagnes 1698903

12/12/19

1 Introduction

Until now we have considered only the symmetric encryption for our tests, which assume that both parties have already stored the same private key. This assumption can not be used for the beginning of a conversation in real scenario, moreover, considering a situation where more people need to talk each other in a secure way, the number of keys needed becomes n^2 (where n is the number of people).

Given these facts, we need the introduction of the public-key infrastructure, based on the "Diffie and Hellman" idea. Here the original key is divided in 2 parts:

- k_E = public key, used for the encryption
- k_D = private key, used for the decryption

The idea behind this method is to provide the public key to everyone and use it only to perform the encryption, the private key is known only by the owner and used to perform the decryption of the messages previously encrypted with the other key.

We can also derive a similar approach for the digital signature, which uses the private key for the encryption and the public key for the decryption, indeed only the owner of the key can perform the opposite operation.

Certificates, using a trusted Certification Authority, can ensure that the public key is provided by the real owner and not by an attacker. The trusted CAs are already known by the operating system.

2 Local attempt

In this section I will analyze the command line interfaces provided by OpenSSL to ensure the communications security over a computer network.

2.1 Generating keys

The NIST standard for the public-key Cryptosystems is RSA which is based on the Diffie-Hellman idea in the introduction. The first 2 methods that I found to be used to generate private keys with RSA and DSA are *genrsa* and *gendsa*, actually superceded by *genpkey.* Therefore, to generate a private key using RSA with the default options, I have used the following command:
openssl genpkey -algorithm RSA **-out** private-key**.pem**

This will generate a 2048 bits key, which for the current standard is not considered much secure anymore. To increase the security I decided to add the option:
-pkeyopt rsa_keygen_bits:4096

The same approach can be used to generate a DSA key, noticing that the options have a different notation.

For real scenario, it is never a good idea to store the key as plaintext, indeed, by adding -aes256 before the algorithm option, it is possible to save the private key encrypted with the AES chiper.
To see the private key details is possible to use the command:
openssl pkey -in private-key**.pem -text**

This will show the private key and all the components used to derive the public key. Therefore, to generate the public-key, the following command can be used:
openssl pkey -in private-key**.pem -out** public-key**.pem -pubout**

Then, to see the public key details is possible to use the command:
openssl pkey -in public-key**.pem -pubin -text**

Note: I decided to use the extension *.pem* because I found it to be RFC standard, but only for the key, also the *.key* extension is commonly used.

2.2 Generating certificates

In real scenario, an attacker could provide his/her own public key, pretending to be someone else. To avoid these situations, digital certificates are used to ensure the authenticity of the keys.

The first step to generate the certificate is to produce the CSR (Certificate Signing Request), using:

```
openssl req -new -sha256 -key private-key.pem -text -out request.csr
```

This command uses the private-key previously generated, but it is also possible to directly generate everything using more options. Considering that, also in this case there are too many options to list all of them in this paper, I want only to highlight the *-sha256*. This option must be used, otherwise openssl will use as default the SHA1, which is known to be not secure anymore.

To show the CSR informations the following command can be used:

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

Now, generated the CSR and inserted all the parameters, there are 2 options:

- A trusted certificate authority sign the certificate (better)
- self-sign the document

For this section I will use the self-sign option. To perform this task we need to generate a pseudo CA with the corresponding key. The key can be generated as explained in the previous subsection, for the CA-cert:

```
openssl req -new -x509 -days 365 -key ca-key.pem -sha256
 -extensions v3_ca -out ca.crt
```

Now we can sign the CSR with the CA-cert:

```
openssl x509 -sha256 -req -in request.csr -CA ca.crt
 -CAkey ca-key.pem -CAcreateserial -out self-signed.pem
 -days 365
```

This generates an X.509 self-signed certificate using the key and CSR previously generated, the *-days* option declare after how many days the certificate will expire. At this point it is possible to show all the informations about the certificate with the command:

```
openssl x509 -text -noout -in self-signed.pem
```

To verify the certificate the following command can be used:

```
openssl verify -CAfile ca.crt self-signed.pem
```

2.3 Digital signing

A digital signature can be attached to a document or some other electronic artifact to vouch for its authenticity. This is performed using a combination of hash (to avoid forgery attacks) and encryption with the private key, as explained before. The openssl command for the signature is:

```
openssl dgst -sha256 -sign private-key.pem -out sign.sha256 file
```

Now, the following command can be used by the other people to verify the signature by the public key:

```
openssl dgst -sha256 -verify public-key.pem
 -signature sign.sha256 file
```

Note: the current standard for the digital signature is DSS which is implemented by the DSA algorithm

2.4 Converting certificates

Certificates and keys can be saved in a few different formats. For my tests I have used the *.pem* extension, because I found it to be governed by RFCs and used preferentially by open-source software. To convert PEM and DER certificates, the following commands can be used:

```
openssl x509 -in cert.pem -outform der -out cert.der

openssl x509 -in cert.der -inform der -outform pem -out cert.pem
```

In case of PKCS#12 and PFX, the commands change and become:

```
openssl pkcs12 -in keyStore.pfx -out keyStore.pem -nodes

openssl pkcs12 -export -out certificate.pfx -inkey privateKey.key
-in certificate.crt -certfile CACert.crt
```

Here is a list fo the most common formats:

- PEM: Extension .pem, .crt, .cer
- DER: Extension .der
- PKCS#7: Extension .p7b, .p7c

- PKCS#12: Extension .p12
- PFX: Extension .pfx

References

[1] https://wiki.openssl.org/index.php/Command_Line_Utilities

[2] https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175B.pdf

[3] https://www.openssl.org/docs/man1.0.2/man1/pkey.html

[4] https://www.openssl.org/docs/man1.1.0/man1/genpkey.html#KEY-GENERATION-OPTIONS

[5] https://www.openssl.org/docs/man1.0.2/man1/openssl-req.html

[6] https://www.openssl.org/docs/man1.0.2/man1/x509.html

[7] https://www.openssl.org/docs/man1.0.2/man1/openssl-verify.html

[8] https://support.ssl.com/Knowledgebase/Article/View/19/0/der-vs-crt-vs-cer-vs-pem-certificates-and-how-to-convert-them