Public and private keys management

HW5 - CNS Sapienza

Luigi Russo 1699981

30/11/2018

1 Introduction

Most of cryptographic algorithms and protocols need the usage of a key, both private or public. The key management is quite complex and involves [1] a lot of operations and tasks: generation, exchange, storage, use, destruction, replacement, etc. The goal of this homework is to provide basic infos about the storage of cryptographic keys. Since a key can be seen as an object whose properties can vary a lot according to the specific usage (e.g. the length), the first thing to do in order to guarantee a correct storage is to serialize the key object, i.e. represent the key as a stream of bytes, ready to be "packed" as a file.

1.1 Key serialization

There are many common schemes that provide a way to serialize both private and public keys to bytes. These methods generally support encryption of private keys and additional key metadata. Also public keys can be encrypted, despite their "intrinsic nature" would suggest to not do that. Some of the choices we could face when storing a key are:

- Type of file:
  - binary file
  - ASCII (text) file
- Use of encryption:
  - NO $\rightarrow$ plaintext
  - YES $\rightarrow$ ciphertext

- The encoding / encapsulation type:
  - PEM
  - DER
  - XML
  - PKCS #12
  - RFC 4253
  - ...
- The format:
  - PKCS #1
  - PKCS #3
  - PKCS #8
  - ...

2 Popular encodings

2.1 PEM

PEM - *Privacy-Enhanced Mail* - is a popular file format for storing keys and certificates. The keys stored in PEM can be also of different types, since PEM uses the so called self-identification, i.e. each key is associated with some overhead useful for the identification.

Type of file It uses the ASCII **base64** encoding and all the keys are encapsulated in a block delimited by an header

```
  -----BEGIN {format}-----
```

and its corresponding footer

```
-----END {format}-----
```

Encryption It is possible to encrypt the private key, using a proper encryption algorithm (e.g AES, Camellia, etc).

Extension The extension of the file is **.pem**, however there are some extensions, not always standardized, like **.key**.

2.2 DER

DER is an ASN.1 encoding type, quite popular on Windows systems.

Encoding There are no encapsulation boundaries and the data is stored in binary. It is the parent format of PEM that, except for the delimiters, can be seen as the base64 equivalent of DER.

Encryption As PEM, also DER supports the encryption of both private and public keys.

Extension The extension of the file is **.der**.

2.3 PKCS #12

Originally defined by RSA as the standard 12, and later defined as standard RFC 7292, it is used generally by Windows systems, although it can be freely converted to PEM format.

Type of file This is a binary file.

Encryption It is encrypted by design.

Extension The extensions of the file are **.pkcs12**, **.pfx** and **.p12**.

2.4 RFC 4253

This is the format used by OpenSSH to store public keys.

Type of file It's a base64 file. There are no header nor footer, no internal line-breaks, and the labels are space-separated.

Encryption It is possible to encrypt the key with the common algorithms.

Extension The extension of the file is **.pub**.

2.5 XML

There is also a W3C standard (**XKMS 2.0**) that supports the storing of both private and public keys.

3 Serialization scripts

I have written a simple interface to handle the most common formats and encodings, both for public and private keys. I have implemented it in Python 3.7, the source code is attached. Currently it is possible to handle these formats and encodings:

- PEM
- DER
- RFC 4253 (*)
- PCKS #1
- PCKS #8
- RFC 5280 (*)

(*) *available only for public keys*

3.1 Example

```
//load key from file PEM
pd = PrivateDumping()
file_pem = 'pkey.pem'
pd.load_key(file_pem)

//converto to DER
pd.set_encoding('der')
file_der = 'pkey.der'
pd.store_key(file_der)
```

References

[1] *Wikipedia Cryptography Portal*
https://en.wikipedia.org/wiki/Portal:Cryptography