SIC Lab
Digital signatures
version 1.0

Authors: André Zúquete, Vitor Cunha

Version log:

- 1.0: Initial version

1 Introduction

In this guide we will develop programs that use cryptographic methods, relying in the **Python3 Cryptography** module. The module can be installed using the typical package management methods (e.g, `apt install python3-cryptography`), or using the `pip3` tool (e.g. `pip3 install cryptography`).

This lab makes use of the commands for creating key pairs developed in the previous lab.

2 RSA

2.1 RSA key pair generation

Create 4 RSA key pairs with the following key length: 1024, 2048, 3072 or 4096.

2.2 RSA signature creation

Create a small program to sign a file using the RSA algorithm. You can use as a starting point the `rsa_sign.py` file. The user must indicate the following parameters:

1. Name of the file with the private key to produce the signature;
2. A padding scheme (either PKCS1 or PSS);
3. A digest function (SHA-256, SHA-348, SHA-512, etc.);
4. Name of the file to sign (or stdin if missing);
5. Name of the signature file (or stdout if missing).

For the same file, use different combinations of private keys, padding algorithms and hashing functions.

PKCS1, or more precisely PKCS1_v1.5, is the oldest padding scheme, which is used very often, but has been progressively abandoned in favor of PSS (Probabilistic Signature Scheme). PSS is a randomized computation, which means that its computation uses a random value. Consequently, it never produces the same output for the same inputs (excluding that random value).

You can save the signature data both in a binary format (the one produced by RSA, and the one being used in the example provided) or in a textual encoding. Base64 is a good choice.

```
import base64

signature = ...
textual_signature = base64.standard_b64encode( signature )
```

For validating a signature encoding in Base64, you need to do the opposite oporation prior to the verification:

```
import base64

signature = base64.standard_b64decode( textual_signature )
# verify the signature validity
```

Questions:

- What are the parameters that have impact on the dimension of the signature?
- Sign the same file two times with the same combination of parameters and save the two signatures in two files. Check if they are equal (use the 'cmp' command). Change the padding scheme and check again. In which case does the signature change, and why?

2.3 RSA signature verification

Create a small program to verify a signature of a file using the RSA algorithm. You can use as a starting point the `rsa_wverify.py` file. The user must indicate the following parameters:

1. Name of the file with the public key to verify the signature;
2. The padding scheme that was used;
3. The digest function that was used (SHA-256, SHA-348, SHA-512, etc.);
4. Name of the signed file;
5. Name of the signature file (or stdin if missing).

Questions:

- What happens, when you do not use the correct parameters to validate a signature?
- What happens, when you update the original final and validate the signature afterwords?

3 Elliptic curves

3.1 ECC key pair generation

In the last lab we created an application to produce a password-protected P-251 key pair. Add the curve selection to the program, in order to be able to use different curves.

Create 3 EC key pairs with the following curves: P-521, B-409, K-163.

In Python, you can manage the implemented curves by their name using the '_CURVES_TYPES' dictionary:

```
from cryptography.hazmat.primitives.asymmetric import ec

def list_curves():
    print( "Lists of available curves:" )
    for curve in ec._CURVE_TYPES:
        print( "\t%s" % (curve) )
```

3.2 EC signature creation

Create a small program to sign a file using an EC key pair. You can use as a starting point the `ec_sign.py` file. The user must indicate the following parameters:

1. Name of the file with the private key to produce the signature;
2. The password used to protect the private key;
3. A digest function (SHA-256, SHA-348, SHA-512, etc.);
4. Name of the file to sign (or stdin if missing);
5. Name of the signature file (or stdout if missing).

For the same file, use different combinations of private keys and hashing functions.

EC signatures do not use a padding scheme (because they are not based on cipher/decipher operations). They use an algorithm known as ECDSA (Elliptic Curve Digital Signature Algorithm). This produces two values that form the signature.

ECDSA is a randomized computation, which means that its computation uses a random value. Consequently, it never produces the same output for the same inputs (excluding that random value).

Questions:

- What are the parameters that have impact on the dimension of the signature?
- Sign the same file two times with the same combination of parameters and save the two signatures in two files. Check if they are equal (use the 'cmp' command). Change the padding scheme and check again. In which case does the signature change, and why?

3.3 EC signature verification

Create a small program to verify a signature of a file using the EC algorithm. You can use as a starting point the `ec_verify.py` file. The user must indicate the following parameters:

1. Name of the file with the public key to verify the signature;
2. The digest function that was used (SHA-256, SHA-348, SHA-512, etc.);
3. Name of the signed file;
4. Name of the signature file (or stdin if missing).

Questions:

- What happens, when you do not use the correct parameters to validate a signature?
- What happens, when you update the original final and validate the signature afterwords?

4 Further Reading

1. Python3 Cryptography: RSA
2. Python3 Cryptography: Elliptic curve cryptography
3. SafeCurves