SIO Lab
Asymmetric cryptography

version 1.0

Authors: João Paulo Barraca, Vitor Cunha, Paulo C. Bartolomeu, André Zúquete

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`).

2 RSA

2.1 RSA key pair generation

Create a small program to generate an RSA key pair, with a key length specified by the user (1024, 2048, 3072 or 4096). The program must save the key pair in two files, one for the private key and the other one for the public key, whose names should also be specified by the user. The keys can be saved as binary blobs but, probably, the PEM format will be more appropriate.

Run the program, several times, varying the key length and register the elapsed time using the `time` program in the shell (use `man time` for more information), for example:

```
time python keygen.py ./pub.txt ./priv.txt 4096
```

Questions:

- What do you think of using 4096 bit keys by default in relation to speed?
- How the actual key size varies with the number of bits?

2.2 RSA Encryption

Create a small program to encrypt a file using the RSA algorithm.

The user must indicate the following data:

1. Name of the original file to encrypt
2. Name of the file with the public key
3. Name for the encrypted file.

For the encryption select the PKCS #1 padding scheme. Note that this scheme is not recommended for new applications. Also, *pay attention to the size of the original file*, i.e., the file to encrypt. Using the PKCS #1 default configuration, the block size is equal to the key size minus eleven bytes (eleven bytes for padding). So, for example using a 1024 bits RSA key (128 bytes), the block size is 117 bytes (128 - 11).

2.3 RSA Decryption

Create a small program to decrypt the contents of a file, whose name is provided by the user, using the RSA algorithm. The user must also indicate:

1. Name of the encrypted file;
2. Name of the file to save the decrypted content;
3. Name of the file containing the private key to use.

2.4 Large file encryption

As should be known by now, RSA encryption is not efficient and is not recommended to encrypt data bigger than its block size. Let's conduct a benchmark.

2.4.1 Symmetric vs. Asymetric cryptograpy benchmark

Generate two files with 100 kB and 10 MB of size. Using these files and the time application, register the time it takes to encrypt and decrypt these files with RSA (key = 1024 bits) and AES128. Employ the applications developed in this guide and in the previous one.

The following bash command can be used to generate a 100 kB file:

```
dd if=/dev/zero of=file.txt bs=1024 count=100
```

2.4.2 Symmetric and Asymetric cryptograpy combination

Consider a large file (500 MB, for example) that you want to send to some person, with the guarantee that only that person can decrypt the file. More, you have the public key of that person, and you are not able to contact the person before sending the file. So, you have a big file to transmit to a person, from which you know the public key, but the process will be very slow if you encrypt the file using RSA . However, you can use any other encryption algorithm to encrypt the file, but remember you want that only the receiver must be able to decrypt the file.

Questions: What combination of encryption technologies allow to efficiently send the file to the recipient, with the guarantee that only that person can decrypt the file?

A typical solution is called *Hybrid Encryption*, which combines two ciphers, a symmetric to encrypt the file with a random key, and a asymmetric to encrypt the key used in the previous step.

Implement a program that is able to encrypt/decrypt a large file using the AES128 algorithm, employing a randomly generated secret. The secret should be supplied to the intended receiver ciphered with RSA using its own public key. The receiver should be able to reverse the process and obtain the original version of the large file. The program should accept the following arguments:

1. Operation to perform (encrypt/decrypt);
2. Name of the (large) file to encrypt/decrypt;
3. Name of the file with the key to be used (public/private).

Questions:

- With this method, what is sent to the destination?
- Should we always send the pubic key?

3 Elliptic curves

Elliptic curve cryptography (ECC) is a lot different from RSA.

First, whilst in RSA one chooses the length of the RSA modulus (in bits), in ECC one chooses a curve, and the curve defines the key length.

Second, the private and public keys that belong to the same key pair are of a different nature: the *private key is an integer* (a scalar), whilst the *public key is a two-coordinate point*. The public component is a point given by the multiplication of the private key with a so-called curve generator (G), which is a point of the curve. Since the sum of two points defines a new point (elliptic curves have a special algebra. . .), and the multiplication of G by the private key (a scalar) is nothing more than a sum of G values, it is simple to conclude that the public key is a point.

Third, there is no such thing as an encryption with a public key and a posterior decryption with the corresponding private key. This is not used at all. Instead, a method similar to a Diffie-Hellman key agreement is used. For encrypting a message to a receiver B, the sender A uses B's public and a new private key to compute a point in the curve, and from that point derives a secret, symmetric key. The message is then encrypted with that symmetric key and sent to B, together with the public component of the random private key created by A. The receiver B multiplies the received public key with its own private key and reaches the same point in the curve, which it can use to derive the exact same secret, symmetric key and use it to decipher the message.

There are many elliptic curves, both proposed in standards and other publications. NIST proposed 15 curves, known as P, B and K curve (5 of each). The name stems from the mathematical elements used in the curves – P from prime, B from binary, K from Koblitz, a variant a B curves. Other curves are recommended by *de facto* standards, such as RFC 7748, which describes Curve25519 and Curve448). The SafeCurves publication provides some guidelines for selecting a god curve.

3.1 ECC key pair generation

ECC key pair generation is fairly simple. First, one chooses a curve; we will use P-521.

Then, we generate a random private key for that curve. This is a 521-bit random integer. Then, from that private key, we generate the public key.

Create a program the generates a key pair and saves it in a file. The private component should be protected by a password, provided as a program parameter. You can use the PEM format for encapsulating each of the keys. We will assume this program will be called 'p256_gen'.

```
p256_gen one_password > KPair
```

Note: you can generate two PEM contents, one for the private component, another for the public component, and store them in the same PEM file. We will consider this approach below.

3.2 Secure messaging with public encryption and private decryption

Create a program 'ecc_encrypt' that encrypts the contents received from the standard input to the standard output. The program should have a parameter a file containing the public of the receiver.

The contents produced must contain both the encrypted input and the originator's public key. You can use any message format for that, for instance, JSON (do not forget to use Base64 for encoding binary contents into strings for JSON). For symmetric encryption, you can use AES in CTR mode and an IV derived from the secret key (e.g., by hashing it).

```
ecc_encrypt KPair < plaintext > ciphertext
```

Create a program 'ecc_decrypt' that decrypts the contents received from the standard input to the standard output. The program should have as parameters a file containing the private key of the receiver and a password for having access to that private key. The format of the input is the one produced by 'ecc_encrypt'.

```
ecc_decrypt KPair one_password < ciphertext > recovered_plaintex
```

After these two command, the contents of 'plaintext' should be equal to the contents of 'recovered_plaintext'.

4 Further Reading

1. Python3 Cryptography: RSA
2. SafeCurves