Crypto APIs

HW3 - CNS Sapienza

Pietro Spadaccino 1706250

13 November 2018

1 Introduction

We present several libraries that offer useful cryptographical functions for both Java and Python languages. The presentation is divided into topics, hashing and symmetric/asymmetric encryption, followed by a brief summary of the cited libraries, listing some of their characteristics.

2 Hashing

2.1 Java

2.1.1 Apache Commons Codec

The library provides the implementation of the most commond hashing functions through the class `DigestUtils`. As an example let's consider the method `static byte[] HNAME(byte[] data)`, where `HNAME` could be one of the following: `sha1`, `sha256`, `sha512` and others. The method takes as input an array of bytes and calculates its hash and can also be overloaded by passing a `String` as argument. The method returns an array of bytes containing the raw binary content of hash. If a hex representation of the has is needed, it can be used the method `static String HNAMEHex(byte[] data)`. It works as the previous method but it returns a `String` instead of an array of bytes.

2.1.2 Guava

Guava offers hashing methods through the class `Hashing`. Instead of directly calculating a hash value, the class has static factory methods that return a `HashFunction` instance. These method includes: `sha1()`, `sha256()`,

`sha512()` and more, all with the signature prefix of `static HashFunction`. Note that `sha1()` is deprecated. The obtained `HashFunction` can be later used with methods like `HashCode hashBytes(BytesBuffer input)` to calculate the hash, and then methods can be called on the returned value to obtain various representation of the hash, like `byte[] asBytes()`, returning an array of bytes or `toString()`, returning the hex representation.

2.1.3 Bouncy Castle

The library provides the interface `Digest` to be used by hashing methods. Implementation of the interface includes `SHA256Digest`, `SHA512Digest` and even `SHA3Digest`. The main methods made available by the interface are: `void update( ... )` that updates the string to be hashed, and `int doFinal(byte[] out, int outOff)`, that closes the digest and produces the final value.

2.2 Python

2.2.1 hashlib

This module provides an interface to different hashing methods. Calculate an hash can be as straightforward as `hashlib.sha256(b"my string")`, which returns the hexadecimal the hash as a hexadecimal string. The string to be hashed can also be constructed via multiple passes by first constructing the hasher `h = hashlib.sha256()` and then updating the string with `h.update(b"my ")` and `h.update(b"string")`. At this point `m.digest()` can be called to return the raw hash, or `m.hexdigest()` to obtain the hexadecimal representation. Note that `sha256` is just an example and these methods can be used with SHA-1 and all families of SHA-2.

2.2.2 pycrypto

The module `Crypto.Hash` provides the same interface as hashlib discussed before.

3 Symmetric encryption

3.1 Java

3.1.1 javax.crypto

This java package implements different cryptographic ciphers through the class `Cipher`. A cipher object can be created with
`Cipher c = Cipher.getInstance("TYPE/MODE/PADDING").` `TYPE` can be for example `AES` or `DES`, `MODE` is the mode of operation like `CBC` or `ECB`, and `PADDING` is the dsired type of padding. All the valid combinations are listed in the official documentation.

3.1.2 Bouncy Castle

The library provides an interface named `BlockCipher` for encryption and decryption of data blocks. If you want to use AES, the class `AESEngine` imlpements the method of the `BlockCipher` interface. To initialize a cipher, it must be called the method `init` on the object. It receives as argument an object implementing the `CipherParameter` interface, which in the case of AES it must include the IV, the key, the block size, the mode of operation and the eventual padding.

3.2 Python

3.2.1 pycripto

The module `Crypto.Cipher` provide some neat APIs for encrypting and decrypting. Let's take AES an an example: first a cipher object must be initialized with `cipher = Cipher.AES.new(key, AES.MODE_CBC, IV)`, where `key` and `IV` are plain python strings and the mode of operation is a constant inside the `AES` class, and could be one of the following: `MODE_ECB`, `MODE_CBC`, `MODE_CFB`, `MODE_PGP`, `MODE_OFB`, `MODE_CTR`, `MODE_OPENPGP`. Then, to encrypt a message it suffices to call `ct = cipher.encrypt(msg)`, returning the raw binary ciphertext, while to decrypt it is necessary to create another cipher object and then to call `cipher2.decrypt(ct)`. Note that by doing so, no padding is added automatically to the key, IV or the message, thus yielding an exception if the key or the IV is not 128, 192 or 256 bit long, or if the message has a length which is not a multiple of the length of the key.

4 Asymmetric encryption

4.1 Java

4.1.1 javax.crypto

The same module used for symmetric encryption, offers also methods for asymmetric encryption. It does so by offering the same interface `Crypto`. Before starting encrypting a message, the key pair must be generated, with the `keyPairGenerator` class. In the example we generate a key pair of 2048 bits.

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(2048);
KeyPair kp = kpg.generateKeyPair();
PrivateKey priv_key = kp.getPrivate();
PublicKey pub_key = keyPair.getPublic();
```

Once the keys are generated, we can create two `Cipher` objects, one for encryption the other for decryption, with `Cipher c = Cipher.getInstance("RSA")`. Then initialize one cipher with for decryption mode using the public (or private) key with `c.init(Cipher.DECRYPT_MODE, publicKey)`, and the other for encryption using the private (or public) key, `c.init(Cipher.ENCRYPT_MODE, privateKey)`. Now we can call `c.doFinal(bytes[] data)` to obtain the ciphertext or the plaintext, depending on which mode was initialized the cipher.

4.1.2 Bouncy Castle

The library provides a more straightforward approach for asymmetric encryption than the standard javax.crypto. The following is an example for generating a key pair:

```
RSAKeyPairGenerator gen = new RSAKeyPairGenerator();
gen.init(new RSAKeyGenerationParameters (
     new BigInteger("10001", 16),
     SecureRandom.getInstance("SHA1PRNG"),
     2048,
     100
 ));

 AsymmetricCipherKeyPair key_pair = gen.generateKeyPair();
```

As the example reports, first a `RSAKeyPairGenerator` must be created, and then it must be initialized with some parameters: public exponent, a secure random generator, the bits of the key and the certainty. The certainty value is used during the generation of prime numbers: if the primality test tells that a number is really prime, then it is prime with a probability not lower than $1 - 1/2^{certainty}$.

4.2 Python

4.2.1 pycrypto

The RSA functionality is offered through the class `Crypto.PublicKey.RSA`. The following example reports how the encryption works:

```
key_pair = RSA.generate(2048, random_seed)
pub_key = key_pair.publickey()
ct = pub_key.encrypt('My secret recipe')
```

where `random_seed` should be a secure random number. To decrypt the message it suffices to call `key.decrypt(ct)`.

4.2.2 rsa

This library is a pure Python implementation of RSA. While its performances are quite bad if compared to other libraries, it simplifies a lot the code and its readability.

```
import rsa
pub_key, priv_key = rsa.newkeys(2048)   #2048 bits
ct = rsa.encrypt('My secret recipe'.encode('utf-8'), pub_key)
...
pt = rsa.decrypt(ct, priv_key)
```

5 Summary of libraries

In Table 1 and 2 are listed some of the characteristics of the presented libraries.

Table 1: Java libraries

	Guava	Bouncy Castle	crypto	Apache Common Codec
Built-in	No	No	Yes	No
Maintained	Yes	Yes	Yes	Yes
Open source	Yes	Yes	Part	Yes
License	Apache	MIT	CDDL	Apache
Mainteiner	Google	Berkley	Oracle	Apache

Table 2: Python libraries

	PyCrypto	rsa	hashlib
Built-in	No	No	Yes
Maintained	Yes	Yes	Yes
Open source	Yes	Yes	Yes
License	GPL	Apache	GPL
Mainteiner	Third party	Third Party	Python Software Foundation