My Cripto Libraries

HW4 - CNS Sapienza

Giulio Serra 1904089

December 5, 2019

Contents

1 Introduction

The aim of this documents is to describe the functionality, compatibility, type and implementations of 4 different cripto libraries for JAVA and Python .

2 JAVA

Java is a widely popular OOP language, there are several cripto library available including a native support with the javax.cripto library and several third party options such as Bouncy Castle (and several others).
For the purpose of this document we are going to evaluate performances, compatibility and options between two libraries.

2.1 javax.Cripto

Java natively provide API to performe encryption using the most common ciphers and operating modes, using the Java Cryptographic Extension(JCE).
This extension remain separated from the core releases of Java for a long time because of the strict rules of the USA government regulating encryption, now is part of JAVA itself and it can easily used worldwide.

The architecture upon wich JCE is based is called Java Cryptography Architecture: the basic components of it's structures are:

- **Cipher**

  The Cipher class rappresent a cryptographic algorithm: it can be used to both encrypt and decrypt a file.
  In order to start the enc/dec process the class need to be instantiated by specifing the mode of encryption and the cipher, for example:
  `cipher.init("AES/CBC/PKCS5Padding");`

- **KeyGenerator**

  As the name suggest, the key generator class is used to create truly random keys (depending on the specified encryption algorithm), avoiding the patterns that could be created using a generic randomize library(ex. *Math.random()*). To create a SecretKey by using the `generateKey();` method, first as usual, we need to instantiate the class by using the `keyGenerator.init(keyBitSize, secureRandom);` method.

- **MesageDigest**

  The java.Cripto library support all the methods to authenticate a message, using the `Mac.getInstance("HmacSHA256");` it's possible to create a MAC(Message authentication code), this ensure a way to authenticate

the message by calculating the MAC using an hash function, before the encryption, and send it along the data on the communication channel.If the message is modified in some way, the MAC will not correspond to the message anymore. Here is a simple implementation to compute the MAC using a simple(and very unsafe) key:

```
MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");
byte[] data = "password".getBytes("UTF-8");
byte[] digest = messageDigest.digest(data);
```

2.2 Bouncy Castle

Bouncy Castle is a collection of API developed by an Australian team, this is a crucial aspect to understand the origin of this library, since at the time of it's first release, the US law covering the export of cryptography were very strict thus creating the necessity for this library.
The architecture of bouncy castle strongly rely on low-level API that holds all the underlying cryptographic algorithms that communicates again with JCE. The main classes and objects that compose bouncy castle are the follows:

- **org.bouncycastle.jcajce.provider.symmetric**

  The symmetric package contain all the needed classes to utilize symmetric ciphers, differently from JCE, there is no super-class (or similar) from which all the ciphers derived from, in fact each cipher (like AES) contains them all.For example, to enc/dec a file using AES in CBC mode, we need to instanciate a new AES engine and padding type using:

  ```
  BufferedBlockCipher cipher = new
  PaddedBufferedBlockCipher( new CBCBlockCipher(new AESEngine()), padding);
  ```

- **KeyGenerator**

  As stated, the Bouncy Castle library rely on JCE, in fact the set of instructions to generate a key are the same used before, it's also possible to salt the message:

  ```
   PBEKeySpec pbeKeySpec = new PBEKeySpec(password.toCharArray(),t
   oByte(salt), 50, 256);
   SecretKeyFactory keyFactory = SecretKeyFactory
  .getInstance("PBEWithSHA256And256BitAES-CBC-BC");
  SecretKeySpec secretKey = new SecretKeySpec(keyFactory.generateSecret(
  pbeKeySpec).getEncoded(), "AES");
  byte[] key = secretKey.getEncoded();
  ```

- **MesageDigest**

  Bouncy castle API support the different variations of MAC, all using the MAC interface.All the different MAC variations need to be initialized using an encryption engine(ex AES) , here is an example:

```
BlockCipher cipher = new AESEngine();
    CMac cmac = new CMac(cipher);
    KDFCounterBytesGenerator kdf = new KDFCounterBytesGenerator(cmac);
```

3 Python

Python is a high-level, multi-paradigm language (in the sense that support different paradigms like OOP), is widely used, for example in machine learning. The main crypto libraries available for python are: cryptography and pycrypto, again we are going to evaluate the performances and compatibility between the two libraries and Java's libraries.

3.1 Pycripto

As described in the official library page: *"It's a collection of both secure hash functions (such as SHA256 and RIPEMD160), and various encryption algorithms (AES, DES, RSA, ElGamal, etc.)"* , pycripto supports all the most common cipher along with all the operating modes such as CBC or ECB. An ideal case of study is for use to authenticate the communication between client and a server (since it support MAC using the Cripto.Hash package).

- **Cipher Class**

  if we want to use a particular cipher we can do so by instatiating the corresponding class, for example AES is implemented as follow:

```
>>> from Crypto.Cipher import AES
>>> obj = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
>>> message = "This is a message"
>>> ciphertext = obj.encrypt(message)

>>> obj2 = AES.new('This is a key123', AES.MODE_CBC, 'This is an IV456')
>>> obj2.decrypt(ciphertext)
The answer is no'
```

- **KeyGenerator**

  For the reason mentioned before, is also possible to generate random keys to use in combination with ciphers:

```
>>> from Crypto import Random
>>> rndfile = Random.new()
>>> rndfile.read(16)
```

- **MesageDigest**

  Pycripto has a strong support form MAC (because python is heavily used server-side), here follows an example of decryption using python(server-side):

```
def decrypt(self, enc):
        enc = base64.b64decode(enc)
        iv = enc[:16]
        hmac = enc[-32:]
        cipher_text = enc[16:-32]

        verified_hmac = self.verify_hmac((iv+cipher_text), hmac)

        if verified_hmac:
            cipher = AES.new(self.key, AES.MODE_CBC, iv, segment_size=128)
            return cipher.decrypt(cipher_text)
        else:
            return 'Bad Verify'
```

3.2 Criptography

As described on the official release page, the aim of this framework is to provide a simple yet powerfull python encryption library: in fact the project is compatible with most Python's versions (ranging from 2.7 to 5) even though the library can be fully customized if more specific needs are required.

- **Cipher Class**

  Again, is possible to use the desired cipher simply creating a new instance of the class rappresenting the desired cipher:

```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
rom cryptography.hazmat.backends import default_backend
backend = default_backend()
key = os.urandom(32)
iv = os.urandom(16)
cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=backend)
encryptor = cipher.encryptor()
ct = encryptor.update(b"a secret message") + encryptor.finalize()
```

```
decryptor = cipher.decryptor()
decryptor.update(ct) + decryptor.finalize()
```

- **KeyGenerator**

  It's possible to auto generate secure keys (a basic requirement for a complete encryption suit by simply using the `os.urandom(NBIT)` instruction)—, for example if we need to create a 16 bit random key.

  ```
  from Crypto import Random
  rndfile = Random.new()
  >rndfile.read(16)
  ```

  In our case a 16 bit key need to be salted to use with our AES-256 cipher.

- **MesageDigest**

  Cryptograpy support HMAC as the it's other counterparts:

  ```
  from cryptography.hazmat.backends import default_backend
  from cryptography.hazmat.primitives import hashes, hmac
  h = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
  h.update(b"message to hash")
  h.finalize()
  ```

4 Compatibility and Conclusions

A good encryption library need to be compatible between different systems : in particular because the heavy use of the Python language in the client-server environment, it's crucial that every library used can decrypt message coming from different library and systems and luckily both Pycrypto and Cryptography ensure a good compatibility with messages encrypted by other systems and programming languages (for example Javascript and in our case JAVA).
Talking about JAVA , the compatibility between Bouncy Castle and javax.Cripto is pretty much implied, because Bouncy Castle is based on JCE, so the only requirement to guarantee a compatibility between framework libraries is to use the correct combination between Cipher/Mode , padding and HMAC(if used).

5 References

Java Cipher:
`https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html`

Legion of Bouncy Castle:

https://www.bouncycastle.org/documentation.html

PyCrypto:
https://pypi.org/project/pycrypto/

Cryptography:
https://cryptography.io/en/latest/hazmat/primitives/cryptographic-hashes/