uc3m | Universidad Carlos III de Madrid

LAB ASSIGNMENT: SYMMETRIC AND ASYMMETRIC CRYPTOGRAPHY

CRYPTOGRAPHY AND COMPUTER SECURITY

Ana I. González-Tablas Ferreres
José María de Fuentes García-Romero de Tejada
Lorena González Manzano
Sergio Pastrana Portillo
UC3M | COMPUTER SECURITY LAB (COSEC) GROUP

TOOLS

The goal of this assignment is to familiarize with one of the most famous encryption software libraries. Bouncy Castle is composed by a set of cryptography classes and APIs for different platforms (.NET, Java, etc.). It has been selected by Google as Android's crypto default implementation.

INTRODUCTION

Sending or receiving confidential information is very frequent in client-server environments. It may be because information is private (e.g. classified information), privileged (i.e. only authorized users may have access) or as a means to defend against security attacks.

For this purpose, almost all programming languages have crypto-related APIs or libraries. It is critical to take into account mixed environments (e.g. cell phones connecting to Linux servers) since their libraries must be interoperable to enable communication. Thus, they need to provide with the same ciphers, with the same configuration parameters, key sizes, etc.). In this assignment, Bouncy Castle will be used on top of Java.

Java's architecture for cryptography (JCA, Java Criptography Architecture) is designed to enable the addition of cryptographic providers. Bouncy Castle is one of them. JCA contains APIs to interact with such providers in an standardized form. In this way, cryptographic providers may be replaced without significant changes in the source code of a given application.

Apart from the use as cryptographic provider, Bouncy Castle may also be integrated into Java applications as external library. Although leveraging on JCA would be more typical for Java, it must be taken into account that not all languages offer such an architecture.

Bouncy Castle structure

Bouncy Castle implements not only encryption algorithms, but also signature, certificate generation, access control, data authentication, etc.

It is organized following the typical package structure of Java, where org.bouncycastle is the root package. In this assignment, we will focus on package org.bouncycastle.crypto, as it contains the implementation for encryption and hash functions that will be considered.

Package org.bouncycastle.crypto.test contains examples of implementation for different ciphers. Its documentation is stored into file docs1.5on of the "doc" folder which comes with this assignment.

Pay attention to the provided code and documentation – it will be helpful as starting point for using BouncyCastle. For example, in order to create and use a symmetric cipher is:

1. Instantiate an Engine object (e.g., AESEEngine)
2. Choose a mode of operation (e.g. CBCBlockCipher) and instantiate it over the Engine.
3. Create a Cipher (e.g. PaddedBufferedBlockCipher) that specifies the particular features (e.g. padding will be added according to PKCS5).
4. Create a key. In this example an initialization vector would be needed, so a KeyParameterWithIV would be created to initialize the cipher.

Software execution

To execute the encryption and hash demos, file Demos.java has to be executed (package p2.demos).

The given code comes in the form of Eclipse project. Thus, it may be executed from Eclipse itself or from a command prompt. In the second case, the code must be executed from the assignment's root directory and library lib/bcprov-ext-jdk15on-151.jar must be included in the Classpath. All files read/write in this code is stored in the "files" directory. The options given to the user are commented below. For each one, several questions are proposed for the student:

EXERCISES

1. Create text file

i) Create text file

It creates a text file within the "files" directory, to be used in the remaining steps of the demo.

2. *DES section*

ii) Generate DES key

It creates a symmetric key for DES and stores it into the “files” directory. It has extension “.deskey” and it is stored in hexadecimal format to increase human readability.

This process uses the SecureRandom class to create a random-as-possible number using a secret seed.

Exercise 1:

a) Read the information provided in the following web page and assess whether it would have been more appropriate to use java.util.Random class.

b) generateKey” method in DES.java multiplies the DES key length by a factor of 8. Why? What happens if the length is changed?

Solution:

a) Using java.util.Random class, a pair of instances using the same seed will generate the same random sequence. Then, if an attacker knows the seed, he/ she could generate the same random sequence. Then, java.util.Random should not be used in applications in which security is a priority. This problem can be solved using java.security.SecureRandom

b) It is multiplied by 8 because keyGenerationParameters constructor needs the length in bits and the constant DESParameters.DES_KEY_LENGTH is in bytes.

If you choose the lengt of the key an exception appears because DES keys are of 64 bits.

iii) DES Encryption/Decryption of a file

It encrypts/decrypts a file using the previously created key. Recall that both the file and the key must be stored within the “files” folder.

As in the example of the previous page, a DES Engine is created, and CBC mode of operation is chosen. This cipher is initialized with the said key and the operation is finally carried out.

Exercise 2:

a) Observe "encrypt" and "decrypt" methods in DES.java. Can they be merged in a single method without impacting to the execution result?

b) Check out Bouncy Castle documentation and change this part to use CFB and OFB modes of operation. Encrypt a given file with both modes. If some byte(s) of the encrypted file are modified, what does it happen when trying to decrypt the file using each of the operations modes?

Solution:

a) DES is symmetric encryption and then, method encrypt could be used for encryption and decryption purposes and it would work completely. The initialization method is the only changed required.

b) Changing operation modes requires updating CBCBlockCipher class by other one, namely CFBBlockCipher, OFBBlockCipher, etc.. Moreover, the initialization should be performed according to the chosen operation mode. You will realize that once encrypting and decrypting with each of the operation modes, ciphertexts are different.

3. AES section

iv) Generate AES Key

Generate a key and an initialization vector for AES algorithm. Both are appended in a file with extension "aeskey", placed in the "files" folder.

Exercise 3:

a) Observe the "generateKey" method in AES.java. Why is blocksize added to the key length when the random number is created? Change the said length, does it have any impact in the execution?

Solution:

a) The length of the encrypted block is added to the length of the AES key because IV should have the same length of the block.

If the generated key length is changed: if the generated key is longer than expected, just the necessary part will be stored; if the generated key is shorter than expected, an exception would appear when storing/ reading AES key to/ from the file.

If you are able to read/ write key+IV of different lengths, exceptions would appear at encryption/ decryption time because IV size will be different from the expected blocksize.

v) AES encryption/ decryption

It allows the encryption and decryption of text data with the previous generated key (both should be located in "files" folder).

Exercise 4:

a) Observe the encrypt/decrypt methods of AES.java. Can they be merged in a single method without impacting to the execution result?

b) Rijndael is a "fast" AES system which allows different key sizes as long as they are multiple of 32 bits. Change the encrypt method to enable that the result of Rijndael is compatible with the existing decrypt function. Rijndael is implemented in the RjindaelEngine class of Bouncy Castle.

Solution:

a) AES is not reversible (in contrast to DES) and then, the same encryption and decryption method cannot be used. However, the difference between both methods is the initialization (init method). If the first parameter is set to "true", encryption will be carried out and if it is set to "false", decryption will take place. Then, from the implementation point of view both methods can be combined (and it is recommendable) but in AES standard this is not possible.

b) The same key and block size (operation mode) for Rjindael should be specified to be compatible with AES standard. Example of Rijndael cipher:

```
BlockCipher engine = new RijndaelEngine(256);
BufferedBlockCipher cipher = new PaddedBufferedBlockCipher(new CBCBlockCipher(engine), new ZeroBytePadding());
byte[] keyBytes = "0123456789abcdef0123456789abcdef".getBytes();
cipher.init(true, new KeyParameter(keyBytes));
byte[] input = "value".getBytes();
byte[] cipherText = new byte[cipher.getOutputSize(input.length)];
int cipherLength = cipher.processBytes(input, 0, input.length, cipherText, 0);
cipher.doFinal(cipherText, cipherLength);
String result = new String(Base64.encode(cipherText));
log.debug("result : " + result);
```

4. Hash functions section

MD5 and SHA1 are implemented in Hash.java. In Bouncy Castle, hash functions (also called "digest") belong to the GeneralDigest class. It contains the basic functions to implement any hash function.

Exercise 5:

a) Refer to the Bouncy Castle documentation to implement a hash method using SHA512.

Solution:

a) The following is an example of the code:

```
public MessageDigest() throws NoSuchAlgorithmException {
   Security.addProvider(new BouncyCastleProvider());
   messageDigest = new SHA512Digest();
}
```

5. *RSA section*

vi) RSA keypair generation

Method "generateKey" of "RSA.java" builds a public-private keypair for RSA. This is tored in a different coding scheme – instead of Hexadecimal, now Base64 is used. For this purpose, Base64Encoder class is adopted.

Base64 is a coding scheme that has two main advantages: the output may be codified with ASCII, and the data representation is independent to how it is stored (Big-Endian / LittleEndian, etc.). This is particularly useful for data exchanges between cell phones and servers. It is also standard for storing personal certificates.

vii) RSA encryption/ decryption

For these operations, standard PKCS1 is applied. It defines how to implement RSA (mathematical properties of the keys, operations to be carried out…).

Exercise 6:

a) Encrypt two text files. The first one must be smaller (in bytes) than the key length whereas the second one must be bigger. What does it happen?
b) Increase the key size to 1548 and 2048 bytes. What does it happen?
c) Observe the encrypt/decrypt methods of RSA. Can they be merged in a single method without impacting to the execution result?

Solution:

a) The general implementation of RSA encryption does not allow encrypting data bigger than the key size. It happens because RSA (and every asymmetric cipher

commonly) is not designed to encrypt “data blocks”, but to encrypt AES session keys which will be used to encrypt data in blocks.

b)When using 1528 bytes it should not work and an exception appears due to the use of an invalid key, while it should work using 2048 bytes

c) RSA is an asymmetric encryption algorithm but its implementation allows “sharing” encryption and decryption code as long as the right key is applied (PrivateKeyFactory or PubliceyFactory). Then, the answer is “yes”, as long as the right keys are applied in the doEncrypt and doDecrypt methods.