Chapter 7. CRYPTOGRAPHY IN JAVA: SYMMETRIC AND ASYMMETRIC ENCRYPTIONS, PASSWORD BASED KEY-DERIVATIONS

The first subject of this section is understanding how encryption functions work in Java. Compared to .NET encryption is done a bit differently but it is not at all hard to do and more, there are external libraries, e.g., Bouncy Castle Crypto APIs, which have extensive support for many encryption functions that are not available in .NET. Moreover, the entire code is open source! For this reason you may prefer to work in Java, while for simplicity you may choose .NET.

Another subject that we reach in this section is how to generate keys. Password based key derivations and randomness are essential tools for generating cryptographic keys. The security of any cryptosystem ultimately depends on the randomness of the key, if the key is easy to guess, then the cryptosystem is trivially broken. Both these primitives are also available in the .NET framework, as well as the encryption primitives that we discussed in .NET have instances in Java.

7.1 SYMMETRIC AND ASYMMETRIC ENCRYPTION: AES, DES AND RSA

According to the Java SE (Standard Edition) documentation, see http://docs.oracle.com/javase/, the following algorithms are supported by the ***Cipher*** class:

- AES/CBC/NoPadding (128)
- AES/CBC/PKCS5Padding (128)
- AES/ECB/NoPadding (128)
- AES/ECB/PKCS5Padding (128)
- DES/CBC/NoPadding (56)
- DES/CBC/PKCS5Padding (56)
- DES/ECB/NoPadding (56)
- DES/ECB/PKCS5Padding (56)
- DESede/CBC/NoPadding (168)
- DESede/CBC/PKCS5Padding (168)
- DESede/ECB/NoPadding (168)
- DESede/ECB/PKCS5Padding (168)

- RSA/ECB/PKCS1Padding (1024, 2048)
- RSA/ECB/OAEPWithSHA-1AndMGF1Padding (1024, 2048)
- RSA/ECB/OAEPWithSHA-256AndMGF1Padding (1024, 2048)

Key sizes are available in parentheses, the mode of operation and padding are also specified. This is not much more support when compared to the .NET framework, we have the same DES, 3DES, AES and RSA suite. Fortunately, there are also many public extensions of Java so there are many others cryptographic functions, modes of operations or paddings supported by external implementations. One of the leading packages is the **Bouncy Castle Crypto package**, see https://bouncycastle.org/specifications.html , which has extensive support for many other constructions, e.g., IDEA, Serpent, RC4, in its ***Cipher*** class. There is clearly much more support for cryptography in Java than .NET, but it is also true that for regular applications it is unlikely that you will need more than the standard AES, 3DES and RSA.

To perform encryption and decryptions you first need to add some imports required for the objects that you are going to use as well as for the exceptions that are going to be thrown. These are summarized in Table 1. Besides the ***Cipher*** class from which all cryptosystems derive, you need to handle keys with ***Key, KeyPair*** and ***KeyPairGenerator*** classes then to randomly generate them with the ***SecureRandom*** class, please refer to the online documentation for more information.

```
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.Security;
import java.security.SecureRandom;
import javax.crypto.Cipher;

import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.ShortBufferException;
```

```
import javax.crypto.spec.SecretKeySpec;
import javax.rmi.CORBA.Util;
```

Table 1. Some imports required in Java for the encryption

To perform encryption the paradigm is a bit different to that of .NET but not necessarily harder while the result is the same. To initialize a cipher object with a particular encryption scheme you will simply call a new instance with:

```
Cipher myAES = Cipher.getInstance("AES/ECB/NoPadding");
```

Then you will have to initialize this to work either in encryption or decryption mode as follows:

```
myAES.init(Cipher.ENCRYPT_MODE, myKey);
myAES.init(Cipher.DECRYPT_MODE, myKey);
```

Encryption and decryption work by simply updating the input with the plaintext (or ciphertext in case of decryption) and then calling the ***doFinal*** method for the remaining blocks (if any). The ***doFinal*** method returns the number of bytes stored in the buffer and the same is done by the ***update*** method. This is all summarized in Table 2.

```
byte[] keyBytes = new byte[16];
// declare secure PRNG
SecureRandom myPRNG = new SecureRandom();
// seed the key
myPRNG.nextBytes(keyBytes);
// build the key from random key bytes
SecretKeySpec myKey = new SecretKeySpec(keyBytes, "AES");
```

```
// instantiate AES object for ECB with no padding
Cipher myAES = Cipher.getInstance("AES/ECB/NoPadding");
// initialize AES objecy to encrypt mode
myAES.init(Cipher.ENCRYPT_MODE, myKey);
// initialize plaintext
byte[] plaintext = new byte[16];
//initialize ciphertext
byte[] ciphertext = new byte[16];
// update cipher with the plaintext
int cLength = myAES.update(plaintext, 0, plaintext.length, ciphertext,
     0);
// process remaining blocks of plaintext
cLength += myAES.doFinal(ciphertext, cLength);
// print plaintext and ciphertext
System.out.println("plaintext:                    "                    +
    javax.xml.bind.DatatypeConverter.printHexBinary(plaintext));
System.out.println("ciphertext:                   "                    +
    javax.xml.bind.DatatypeConverter.printHexBinary(ciphertext));
// initialize AES for decryption
myAES.init(Cipher.DECRYPT_MODE, myKey);
// initialize a new array of bytes to place the decryption
byte[] dec_plaintext = new byte[16];
cLength = myAES.update(ciphertext, 0, ciphertext.length, dec_plaintext,
    0);
// process remaining blocks of ciphertext
cLength += myAES.doFinal(dec_plaintext, cLength);
// print the new plaintext (hopefully identical to the initial one)
System.out.println("decrypted:                    "                    +
    javax.xml.bind.DatatypeConverter.printHexBinary(dec_plaintext));
```

Table 2. Example of AES encryption and decryption in Java

For asymmetric encryption or decryption the procedure is similar. The only distinction is that you now have to deal with two keys: a public and private one. For this

reason you now have a ***KeyPair*** object to store the pair of keys, you ***Key*** objects to store individually the private or public key (when needed for wither decryption or encryption) and of course the ***KeyPairGenerator*** object to generate these keys. Table 3 summarizes the source code for the procedures.

```
// get a Cipher instance for RSA with PKCS1 padding
Cipher myRSA = Cipher.getInstance("RSA/ECB/PKCS1Padding");
// get an instance for the Key Generator
KeyPairGenerator myRSAKeyGen = KeyPairGenerator.getInstance("RSA");
// generate an 1024 bit key
myRSAKeyGen.initialize(1024, myPRNG);
KeyPair myRSAKeyPair= myRSAKeyGen.generateKeyPair();
// store the public and private key individually
Key pbKey = myRSAKeyPair.getPublic();
Key pvKey = myRSAKeyPair.getPrivate();
// init cipher for encryption
myRSA.init(Cipher.ENCRYPT_MODE, pbKey, myPRNG);
// encrypt, as expected we encrypt a symmetric key with RSA rather than
    a file or some longer stream which should be encrypted with AES
ciphertext = myRSA.doFinal(keyBytes);
// init cipher for decryption
myRSA.init(Cipher.DECRYPT_MODE, pvKey);
// decrypt
plaintext = myRSA.doFinal(ciphertext);
System.out.println("plaintext:                    "                    +
    javax.xml.bind.DatatypeConverter.printHexBinary(plaintext));
System.out.println("ciphertext:                   "                    +
    javax.xml.bind.DatatypeConverter.printHexBinary(ciphertext));
System.out.println("keybytes:                     "                    +
    javax.xml.bind.DatatypeConverter.printHexBinary(keyBytes));
```

Table 3. Example of RSA encryption and decryption in Java

7.2 GENERATING KEYS: PASSWORD BASED KEY DERIVATION

Humans are not at all efficient in remembering, e.g., storing in mind, cryptographic random keys. But it happens that humans are better in remembering passwords or even longer sentences if these have some sense, i.e., passphrases. The issue with these is that they are generally incompatible with the format required for a cryptographic key. For example, an AES key has exactly 128 bit, but imagine that password can have 18 characters which require 144 bits for storing. The 18 character passwords have little chances in having 128 bits of entropy, i.e., being more secure than 128 bits picked at random, but is clearly easier to remember. If we truncate the 18 character password to 16 characters it will fit the 128 bits of the ley but it is a pity to lose the additional bits of entropy. Password based key derivation (PBKD) is here to help.

The main idea behind PBKD is to use a hash function in order to get an output of fixed size. But besides this hash function there are two more ingredients which you already met in the section dedicated to the UNIX password authentication system:

i) Salts, which and are used to prevent off-line guessing attacks,
ii) Iterations, which are used to make testing for each password more intensive and to hinder the adversary.

Both the salt and the iterations value are public and they not need to be kept secret.

The example provided in Table 1 shows how to generate a 128 bit key for AES by using a fixed password, a randomly generated salt and a fixed number of iterations. The number of iteration makes the key harder to crack by an adversary. The point is that a user will generate this key rarely, e.g., for each login, so waiting 10.000 iterations can take milliseconds which go unnoticeable. For an adversary however, the same procedure must be repeated for each password it tries, thus hindering the exhaustive search.

```
char[] password = "short_password".toCharArray();
byte[] salt = new byte[16];
int iteration_count = 10000;
```

```
int key_size = 128;

// set salt values to random

myPRNG.nextBytes(salt);


// initialize key factory for HMAC-SHA1 derivation

SecretKeyFactory                        keyFactory                        =
     SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");

// set key specification

PBEKeySpec pbekSpec = new PBEKeySpec(password, salt, iteration_count,
     key_size);

// generate the key

SecretKey         myAESPBKey         =         new         SecretKeySpec(
     keyFactory.generateSecret(pbekSpec).getEncoded(), "AES");

// print the key

System.out.println("AES              key:              "              +
     javax.xml.bind.DatatypeConverter.printHexBinary(myAESPBKey.getEnc
     oded()));
```

Table 3. Example of password based key derivation for 128 bit AES with HMAC-SHA1 from password, random salt and 10000 iterations

7.3 EXERCISES

1. Write a program that performs encryption in CBC mode then in OFB and CFB by using a key that is generated from a user's password. Please remember to correctly set the IVs.

2. Write a program that derives keys from passwords and displays the computational time required for generating the password and the computational time required by an adversary to break the password by considering l iterations for password generation and passwords of k bit entropy.

FURTHER REFERENCES

You may find it useful to consult the following references for cryptography in .NET and Java:

[1] David Hook, ***Beginning Cryptography with Java***, Wiley Publishing, ISBN-13: 978-0764596339, ISBN-10: 0764596330, 2005

[2] Peter Thorsteinson, G. Gnana Arun Ganesh, ***.NET Security and Cryptography***, Prentice Hall, ISBN-13: 007-6092021490, ISBN-10: 013100851X, 2003.

[3] Jason R. Weiss, ***Java Cryptography Extensions: Practical Guide for Programmers***, Morgan Kaufmann, ISBN-13: 978-0127427515, ISBN-10: 0127427511, 2004.

[4] *****, Microsoft Developer Network (MSDN), https://msdn.microsoft.com

[5] *****, ORACLE Java Documentation , http://docs.oracle.com/javase/8/