Public key cryptography in OpenSSL

HW6 - CNS Sapienza

Giulio Serra 1904089

December 12, 2019

Contents

1 Introduction

The aim of this essay is to describe how to Implement RSA and DSA in along with X.509 Standard for certificates using JAVA.

2 RSA

2.1 Overview

RSA(Rivest–Shamir–Adleman) is a public key cryptosystem, where the messages is encrypted using a public key while the decryption can be done using a private key that is kept secret.
RSA is based on choosing two random prime numbers: p and q, with different length, then the public key is computed: it consists of the modulus n and the public (or encryption) exponent e, the private key consists of the private (or decryption) exponent d:

$$d = e^{-1}\lambda(mod(n))$$

2.2 Implementation

The implementation is designed around the java.security package that provides wide support for the RSA encryption system, first we need to implement a class to handle the encryption and decryption along with method to generate a pair of keys:

```
public class RSA extends AbstractAsymetric {

PrivateKey privateKey;

@Override
public KeyPair generateKeyPair() {

try {
  KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA");
  generator.initialize(2048, new java.security.SecureRandom());
  KeyPair pair = generator.generateKeyPair();
  return pair;
}catch(Exception e) {
System.out.println(e);
return null;
}

}
```

To use RSA encryption, we need to initialize the Cipher class (as introduced in HW1), along with the public key:

```
@Override
public String encrypt(String plainText, PublicKey publicKey) {

try {
 Cipher encryptCipher = Cipher.getInstance("RSA");
     encryptCipher.init(Cipher.ENCRYPT_MODE, publicKey);

     byte[] cipherText = encryptCipher.doFinal(plainText.getBytes());

     return Base64.getEncoder().encodeToString(cipherText);
}catch(Exception e) {
System.out.print("Error encrypting using RSA " + e);
return null;
}
```

The decryption method takes for input the private key and return the plain text:

```
@Override
public String decrypt(String cipherText, PrivateKey privateKey) {

try {
  byte[] bytes = Base64.getDecoder().decode(cipherText);

      Cipher decriptCipher = Cipher.getInstance("RSA");
      decriptCipher.init(Cipher.DECRYPT_MODE, privateKey);

      return new String(decriptCipher.doFinal(bytes));
}catch(Exception e) {
System.out.print("Error decrypting using RSA " + e);
return null;
}
```

Putting all togheter:

```
/****
 * Test the RSA asy sign method
 */
public static void testRSASign() {
```

```
RSA rsa = new RSA();

KeyPair keys = rsa.generateKeyPair();

String message = "messageTest";
String encrypted = rsa.encrypt(message, keys.getPublic());

System.out.println("RSA Decryption: " +  rsa.decrypt(encrypted, keys.getPrivate()).equals(me

// Console output:
RSA Decryption: true
}
```

3 DSA

3.1 Overview

The Digital Signature Algorithm is a standard for digital signature based upon modular exponentiation and discrete logarithm (witch has the property of being computational intractable).
DSA works on the same principles of RSA: the algorithm uses a key pair consisting of a public key and a private key that is used to generate a digital signature for a message, and such a signature can be verified by using the signer's corresponding public key.
The main difference between RSA and DSA is that the second one is specifically designed for signature and only assure the the authenticity of the message, since it does not provide any encryption method.

3.2 Implementation

As usual the implementation uses the java.security package, first we define a method inside the DSA class to generate a pair of keys:

```
public KeyPair generateKeyPair() {

try {
  KeyPairGenerator generator = KeyPairGenerator.getInstance("DSA");
  generator.initialize(1024, new java.security.SecureRandom());
  KeyPair pair = generator.generateKeyPair();
  return pair;
}catch(Exception e) {
System.out.print(e);
return null;
}
```

```
}
```

The class also need to implement the signature of data along with the verification of signed informations:

```
/***
 * Sign a message with a private key
 * @param message to sign
 * @param privateKey key to sign the message
 * @return sgined data
 */
public byte[] sign(String message,PrivateKey privateKey) {
try {
Signature dsa = Signature.getInstance("SHA/DSA");
dsa.initSign(privateKey);
dsa.update(message.getBytes());

    return dsa.sign();

}catch(Exception e) {
 e.printStackTrace();
System.out.println("Error signing message" );
return null;
}

}

/**
 * Verify the signature of a message
 * @param data
 * @param publicKey
 * @return
 */
public boolean verifySignature(byte[] data,byte[] signed, KeyPair keys) {

try {

Signature dsa = Signature.getInstance("SHA/DSA");
dsa.initSign(keys.getPrivate());
dsa.update(data);

        /* Initialize the Signature object for verification */
        PublicKey pub = keys.getPublic();
        /* Encode the public key into a byte array */
        byte[] encoded = pub.getEncoded();
```

```
        /* Get the public key from the encoded byte array */
        PublicKey fromEncoded = KeyFactory.getInstance("DSA", "SUN").generatePublic(new X509
        dsa.initVerify(fromEncoded);

        /* Update and verify the data */
        dsa.update(data);

        return dsa.verify(signed);

}catch(Exception e) {
e.printStackTrace();
System.out.println(e);
}
    return false;
}
```

So putting is possible to sign messages with DSA by using:

```
/****
 * Test the DSA asy sign method
 */
public static void testDSASign() {

 DSA dsa = new DSA();
 KeyPair keys = dsa.generateKeyPair();

 String message = "messageTest";

 byte[] signed = dsa.sign(message, keys.getPrivate());
 System.out.println("DSA sign verification: " + dsa.verifySignature(message.getBytes(), sign

}

// Console output:
DSA sign verification: true
```

4 X509

4.1 Overview

X509 is a certificate standard (proposed by the International Union of Telecomunications in 1988) used to define the format of public key certificates (PKC) and attributes certificates, especially used in SSL/TLS.
The certificates are used to validate the identity and propagation of data, only the owner of the certificate is able to read and decrypt, those certificates are provided by a trusted certification authority (CA) that assure the authenticity of the public key used in the certificate.

4.2 Implementation

The implementation in Java of the X509 standard is quite complicated, since the java.security package does not yet support the protocol and the sun.security.x509 framework is not available outside the US (for the reason explained in HW4), so the only way to implement it is by using the legion of bouncy castle framework, here follow an implementation of X509 certificate generation with a pair of RSA keys:

```
import org.bouncycastle.asn1.ASN1ObjectIdentifier;
import org.bouncycastle.asn1.x500.X500Name;
import org.bouncycastle.asn1.x509.BasicConstraints;
import org.bouncycastle.asn1.x509.ExtendedKeyUsage;
import org.bouncycastle.asn1.x509.KeyPurposeId;
import org.bouncycastle.asn1.x509.X509Extensions;
import org.bouncycastle.asn1.x509.X509Name;
import org.bouncycastle.jce.provider.BouncyCastleProvider;
import org.bouncycastle.util.Strings;
import org.bouncycastle.x509.X509V1CertificateGenerator;
import org.bouncycastle.x509.X509V3CertificateGenerator;

/***
 * Generate the X509 certificate
 * @param keyPair to sign the certificate
 * @return an X509 certificate
 */
public X509Certificate generateCert(KeyPair keyPair){

try {

X509V3CertificateGenerator certGen = new X509V3CertificateGenerator();
X500Principal dnName = new X500Principal("cn=example");

// add some options
certGen.setSerialNumber(BigInteger.valueOf(System.currentTimeMillis()));
certGen.setSubjectDN(new X509Name("dc=name"));
certGen.setIssuerDN(dnName); // use the same
// yesterday
certGen.setNotBefore(new Date(System.currentTimeMillis() - 24 * 60 * 60 * 1000));
// in 2 years
certGen.setNotAfter(new Date(System.currentTimeMillis() + 2 * 365 * 24 * 60 * 60 * 1000));
certGen.setPublicKey(keyPair.getPublic());
certGen.setSignatureAlgorithm("SHA256WithRSAEncryption");
certGen.addExtension(X509Extensions.ExtendedKeyUsage, true,
new ExtendedKeyUsage(KeyPurposeId.id_kp_timeStamping));
```

```
// finally, sign the certificate with the private key of the same KeyPair
X509Certificate cert = certGen.generate(keyPair.getPrivate(), "BC");
return cert;
}catch(Exception e) {
System.out.println("error creating X509 cert " + e);
return null;
}
}
```

The X509 certificate can be exported in a variety of extensions (for example .pem and .crt), by using the following methods:

```
/***
 * Convert a certificate from
 * @param cert X509Certificate to convert
 * @return a string to write as pem certificate
 */
private String convertToPem(X509Certificate cert){

try {

final Base64.Encoder encoder = Base64.getMimeEncoder(64, LINE_SEPARATOR.getBytes());

final byte[] rawCrtText = cert.getEncoded();
final String encodedCertText = new String(encoder.encode(rawCrtText));
final String prettified_cert = BEGIN_CERT + LINE_SEPARATOR + encodedCertText + LINE_SEPARATO
return prettified_cert;

}catch(Exception e) {
System.out.println("Error converting X509 to pem:" + e);
return null;
}

}

/**
 * Write a pem certificate in X509 format to directory
 * @param keyPair pair of keys to aut
 */
public void writePemCertificate(KeyPair keyPair) {
try {
String toWrite = convertToPem(generateCert(keyPair));
FileWriter fw = new FileWriter(CERTPATH + ".pem");
fw.write(toWrite);
fw.close();
}catch(Exception e) {
```

```
System.out.println("Cannot write pem certificate " + e);
}

}

/**
 * Write a pem certificate in X509 format to directory
 * @param keyPair pair of keys to aut
 */
public void writeCerCertificate(KeyPair keyPair) {

try {

FileOutputStream os = new FileOutputStream(CERTPATH  + ".cer");
os.write("-----BEGIN CERTIFICATE-----\n".getBytes("US-ASCII"));
os.write(Base64.getEncoder().encode(generateCert(keyPair).getEncoded()));
os.write("-----END CERTIFICATE-----\n".getBytes("US-ASCII"));
os.close();

}catch(Exception e) {
System.out.println("Cannot write cer certificate " + e);
}

}
```