Ejercicios de PKI

Índice

1. Encriptación de Tarjetas de Crédito

La seguridad de una base de datos atañe tanto a la seguridad en la conexión como a la seguridad (encriptación) de los datos propiamente dichos. En este tema nos centraremos en el estudio de ésta última y lo haremos a través de un ejemplo. Supongamos que tenemos un servidor que acepta datos de tarjetas de créditos y los almacena en una BD encriptados con su clave pública. La clave privada solamente es conocida por el departamento de finanzas de la empresa.

Para llevar a cabo esta tarea definiremos e implementaremos 4 clases:

CreditCard	Objeto para incluir datos de la tarjeta: **mAccountID** y **mCreditCardNumber**. Soporta los métodos **CreditCard()**, **getAccountID** y **getCreditCardNumber()**.
CreditCardDBO	Objeto que contiene **mAccountID** y los datos encriptados de la tarjeta **mEncryptedCCNumber** y **mEncryptedSessionKey** (número de tarjeta y clave sesión). Soporta los métodos: **CreditCardDBO()**, **getAccountID()**, **getEncryptedCCNumber()** y **getEncryptedSessionKey()**.
DatabaseOperations	Maneja el acceso a través de JDBC. Están definidas las operaciones: **getAllCreditCardAccountIDs()**, **loadCreditCardDBO()** y **store()**. Maneja objetos **CreditCardDBO** sin realizar tareas de encriptación/desencriptación.
CreditCardFactory	Maneja la encriptación y desencriptación de las tarjetas. Utiliza una misma **mPublicKey** para encriptar. Permite **createCreditCard()**, **findAllCreditCards()** y **findCreditCard()** siempre y cuando la clave privada apropiada se suministre como argumento.

En este tema nos centraremos en la descripción de la clase **CreditCardFactory** cuyo código se encuentra en el fichero CreditCardFactory.java. El código de las clases restantes se encuentra, respectivamente, en los siguientes ficheros: CreditCard.java, CreditCardDBO.java y DatabaseOperations.java.

2. Configuración de la Base de Datos

En primer lugar lanzaremos el servidor de mysql (el servidor está en mySQL y el driver está en driver) y crearemos una BD de datos como:

```
CREATE DATABASE projava;
USE projava;
CREATE TABLE account (
    account_id INT8 PRIMARY KEY,
    customer_name VARCHAR(40),
    balance FLOAT,
    cert_serial_number VARCHAR(255)
);
CREATE TABLE credit_card (
    account_id INT8 PRIMARY KEY,
    session_key VARCHAR(255),
    cc_number VARCHAR(100)
);
```

Para finalizar con la configuración de myslq añadiremos un usuario de nombre *usuario* y de password *clave*:

```
USE projava;
GRANT ALL PRIVILEGES ON * TO usuario@localhost IDENTIFIED BY "clave";
GRANT ALL PRIVILEGES ON * TO usuario@localhost.localdomain IDENTIFIED
BY "clave";
```

Salvo que el ususario sea root.

En segundo lugar, asumimos que la clave pública se obtiene del certificado *publica.cer*. Este certificado se crea y se guarda en el almacén *tarjetas.ks* y se exporta en formato DER:

keytool -genkey -keyalg RSA -keystore tarjetas.ks

keytool -export -file publica.cer -keystore tarjetas.ks

En tercer lugar creamos un fichero de configuración llamado **config.properties** correspondiente a la clase **Java.util.Properties** en donde especificamos: (1) path al certificado que contendrá la clave pública, (2) usuario de la BD, (3) password de dicho usuario, (4) URL de la BD y (5) driver de la BD.

```
PublicKeyFilename:publica.cer
DBUsername:usuario
DBPassword:clave
DBUrl:jdbc:mysql://localhost/projava
```

```
DBDriver:com.mysql.jdbc.Driver
```

3. Clase CreditCardFactory

Como ya vimos, la clase **CreditCardFactory** soporta la creación, encriptación, grabación, acceso y desencriptación de tarjetas de crédito (objetos **CreditCard**). El código completo está en el fichero CreditCardFactory.java del paquete *com.projavasecurity.ecommerce.creditcard*

Supongamos que alguien llama al método **CreditCardFactory.CreateCreditCard()** con el propósito de crear, encriptar y almacenar una nueva tarjeta (le pasaremos el número de cuenta **accountID** y el número de tarjeta de crédito **creditCardNumber**). Se realizan los siguientes pasos:

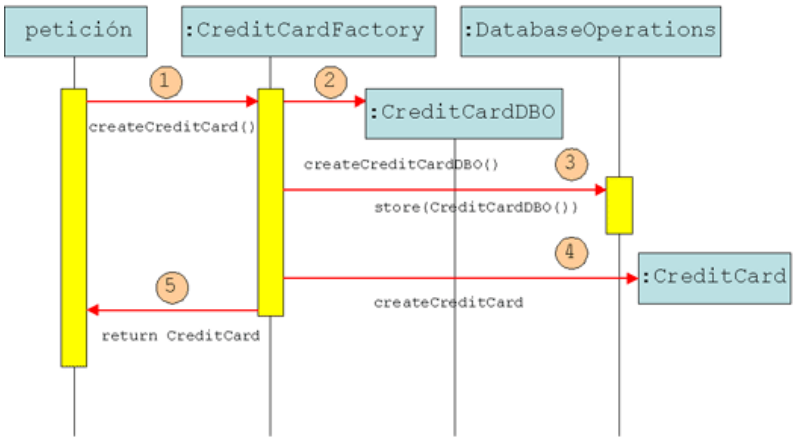
1. Crear una clave de sesión (simétrica) con Blowfish, inicializar un cifrador y aplicarlo sobre **creditCardNumber** con lo que tenemos el número de tarjeta encriptado.

2. Usar la clave pública (**mPublicKey**) del **CreditCardFactory** para inicializar un cifrador asimétrico y encriptar así la clave sesión.

3. Crear un nuevo objeto **CreditCardDBO** con el número de cuenta, la clave sesión encriptada y el número de tarjeta encriptado.

4. Crear un objeto **DatabaseOperations** y llamar al método **store** pasándole como argumento el objeto **CreditCardDBO** que acabamos de crear. Con ello almacenamos los datos encriptados.

5. Devolver el objeto **CreditCard** con los datos sin encriptar.



Creación de una nueva tarjeta

El código se muestra a continuación. Como puede observarse está incluido en un par **try/catch** con múltiples **catch** para capturar diversos tipos de excepción:

```
public CreditCard createCreditCard
   (long accountID, String creditCardNumber)
   throws InvalidKeyException, IOException {

   CreditCardDBO creditCardDBO = null;
   byte[] encryptedSessionKey, encryptedCCNumber;

   try {
    // 1a. Crear clave sesión (simétrica).
    KeyGenerator kg = KeyGenerator.getInstance("Blowfish");
    kg.init(128);
    Key sessionKey = kg.generateKey();

    // 1b. Inicializar cifrador y encriptar número de tarjeta.
    Cipher symmetricCipher = Cipher.getInstance
    ("Blowfish/ECB/PKCS5Padding");
    symmetricCipher.init(Cipher.ENCRYPT_MODE, sessionKey);
    encryptedCCNumber = symmetricCipher.doFinal
    (creditCardNumber.getBytes("UTF8"));

    // 2. Usar clave pública para encriptar clave sesión.
    Cipher asymmetricCipher = Cipher.getInstance
    ("RSA/ECB/PKCS1Padding");
    asymmetricCipher.init(Cipher.ENCRYPT_MODE, mPublicKey);
    encryptedSessionKey = asymmetricCipher.doFinal
    (sessionKey.getEncoded());

    // Need to catch a large number of possible exceptions:
    } catch (NoSuchAlgorithmException nsae) {
    // We're in trouble. Missing RSA or Blowfish.
    nsae.printStackTrace();
    throw new RuntimeException("Missing Crypto algorithm");
    } catch (NoSuchPaddingException nspe) {
    // again, we're in trouble. Missing padding.
    nspe.printStackTrace();
    throw new RuntimeException("Missing Crypto algorithm");
    } catch (BadPaddingException bpe) {
    // Probably a bad key.
    bpe.printStackTrace();
    throw new InvalidKeyException("Missing Crypto algorithm");
    } catch (IllegalBlockSizeException ibse) {
    // Probably a bad key.
    ibse.printStackTrace();
    throw new InvalidKeyException("Could not encrypt");
    }

    // 3. Crear objeto con la información encriptada.
```

```
    creditCardDBO = new
       CreditCardDBO(accountID, encryptedSessionKey,
encryptedCCNumber);

    // 4. Almancenar objeto con la información encriptada.
    mDBOperations.store(creditCardDBO);
    CreditCard creditCard = new CreditCard(accountID,
creditCardNumber);
    // 5. Devolver objeto con la información original.
    return creditCard;
}
```

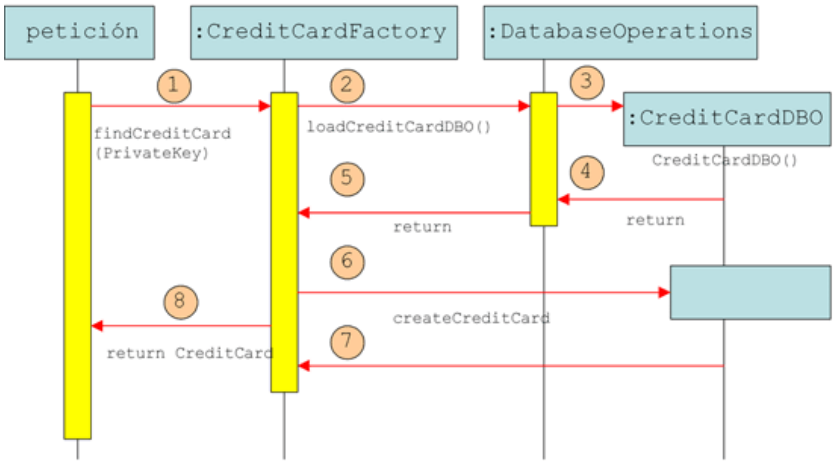
Si ahora, por el contrario pretendemos acceder a los datos encriptados de una tarjeta y luego devolverlos desencriptados accederemos al método **findCreditCard()** pasándole como argumento el número de cuenta y la clave privada (**accountID** y **privateKey**). Se seguirán los siguientes pasos:

1. Crear un objeto **DatabaseOperations** y llamar al método **loadCreditCardDBO** pasándole como argumento el **accountID**. Esta llamada devuelve un objeto **CreditCardDBO** que contiene: número de cuenta, clave sesión encriptada y número de tarjeta encriptado.

2. Utilizar la clave privada para desencriptar la clave sesión.

3. Utilizar la clave sesión para desencriptar el número de tarjeta.

4. Devolver un objeto **CreditCard** con la información desencriptada.



Acceso a una tarjeta

A continuación mostramos el código fuente (de nuevo múltiples **catch**):

```
public CreditCard findCreditCard
```

```
   (long accountID, PrivateKey privateKey)
   throws InvalidKeyException, IOException{

   String creditCardNumber = null;

   // 1. Cargar la información encriptada de la tarjeta desde la BD.
   CreditCardDBO creditCardDBO =
        mDBOperations.loadCreditCardDBO(accountID);
        try {
           // 2. Desencriptar la clave sesión.
           Cipher asymmetricCipher = Cipher.getInstance
                ("RSA/ECB/PKCS1Padding");
           asymmetricCipher.init(Cipher.DECRYPT_MODE, privateKey);
           byte[] sessionKeyBytes = asymmetricCipher.doFinal
           (creditCardDBO.getEncryptedSessionKey());

           // 3. Desencriptar el número de tarjeta con la clave
sesión.
           SecretKey symmetricKey = new SecretKeySpec
                (sessionKeyBytes, "Blowfish");
           Cipher symmetricCipher = Cipher.getInstance
                ("Blowfish/ECB/PKCS5Padding");
           symmetricCipher.init(Cipher.DECRYPT_MODE, symmetricKey);
           byte[] ccNumberBytes = symmetricCipher.doFinal
                  (creditCardDBO.getEncryptedCCNumber());

           creditCardNumber = new String(ccNumberBytes, "UTF8");

       // Need to catch a large number of possible exceptions:
       } catch (NoSuchAlgorithmException nsae) {
         // Missing an algorithm.
           nsae.printStackTrace();
               throw new RuntimeException("Missing crypto algorithm");
       } catch (NoSuchPaddingException nspe) {
         // again, we're in trouble. Missing padding.
           nspe.printStackTrace();
               throw new RuntimeException("Missing Crypto algorithm");
       } catch (BadPaddingException bpe) {
         // This means the data is probably bad.
               bpe.printStackTrace();
               throw new InvalidKeyException("Could not decrypt");
       } catch (IllegalBlockSizeException ibse) {
         // Probably a bad key.
               ibse.printStackTrace();
               throw new InvalidKeyException("Could not decrypt");
       }

   // 4. Crear y devolver objeto CreditCard.
   CreditCard creditCard = new CreditCard(accountID,
creditCardNumber);
   return creditCard;
}
```

En tercer lugar, el método **CreditCardFactory**, es decir el constructor de la clase, se invoca cuando desde cualquier programa java queremos modificar o consultar nuestra base de datos de tarjetas de crédito. Dicho método recibe como argumento un objeto de la clase **Properties** que contiene las propiedades almacenadas en nuestro fichero **config.properties**. Entonces accede al fichero que contiene el certificado digital (en nuestro caso el fichero que figura en el campo **PublicKeyFilename**) y obtiene la clave pública. Finalmente construye un objeto **DatabaseOperations** para implementar las operación de modificación o de consulta:

```
public CreditCardFactory (Properties properties) throws IOException {
   String certFilename = properties.getProperty("PublicKeyFilename");
   try {
       // Acceder a la clave pública
        FileInputStream fis = new FileInputStream(certFilename);
        java.security.cert.CertificateFactory cf =
java.security.cert.CertificateFactory.getInstance("X.509");
        java.security.cert.Certificate cert =
             cf.generateCertificate(fis);
        fis.close();
        mPublicKey = cert.getPublicKey();
   } catch (Exception e) {
        e.printStackTrace();
        throw new IOException(e.getMessage());
   }
   // Crear objeto de operacion de base de datos
   mDBOperations = new DatabaseOperations(properties);
}
```

4. Modificando y Consultando la BD

Finalmente vamos a ver dos programas de aplicación que permiten añadir y consultar tarjetas de crédito. El primero de ellos es CreateTest.java y simplemente lee el fichero de propiedades, crea un **CreditCardFactory** y almancena el número de cuenta y la tarjeta que se le pasan por teclado:

```
...
// Cargar propiedades.
Properties properties = new Properties();
FileInputStream fis = new FileInputStream(PROPERTIES_FILE);
properties.load(fis);
fis.close();

// Crear la tarjeta.
CreditCardFactory factory = new CreditCardFactory(properties);
CreditCard creditCard = factory.createCreditCard(id,ccNumber);
...
```

Para probar esta aplicación nos situaremos en el directorio en donde tenemos los ficheros *config.properties*, *publica.cer* y *tarjetas.ks*. Allí tenemos todos los ficheros *.java y hacemos lo siguiente para construir el package:

*javac -d *.java*

Asumiendo que tenenos el fichero *jar* del JDBC en el *CLASSPATH* o bien accesible en *$JAVA_HOME/jre/lib/ext*, teclearemos el comando

java com.projavasecurity.ecommerce.CreateTest 1 "1234 5678 9012 3456"

Posteriormente haremos desde *mysql* haremos

```
USE projava;
SELECT * FROM credit_card;
```

para ver los campos **session_key** y **cc_number**, que deberán aparecer encriptados.

Para listar la información desencriptada de todas las tarjetas de la BD, recurriremos a la aplicación ViewTest.java. Dicha aplicación consulta, en primer lugar, el *keystore tarjetas.ks*. Para ello hay que suministrar el password correcto para el almacén. Entonces extrae la clave privada para poder desencriptar la clave sesión que a su vez ha permitido encriptar los datos de las tarjetas. Después crea un **CreditCardFactory** con las **Properties** y después llama al método **findAllCreditCards()** pasándole la clave privada. Este método funciona haciendo sucesivas llamadas al método **findCreditCard()** y recogiendo el resultado en un iterador. Este iterador es el que se usa aquí para ir mostrando todas las tarjetas:

```
// Cargar el keystore y obtener la clave privada.
String ksType = KeyStore.getDefaultType();
KeyStore ks = KeyStore.getInstance(ksType);
FileInputStream fis = new FileInputStream(KEYSTORE);
ks.load(fis,PASSWORD);
fis.close();
PrivateKey privateKey = (PrivateKey)ks.getKey("mykey",PASSWORD);

// Cargar las properties
Properties properties = new Properties();
fis = new FileInputStream(PROPERTIES_FILE);
properties.load(fis);
fis.close();

// Crear un CreditCardFactory.
CreditCardFactory factory = new CreditCardFactory(properties);

// Coger todas las tarjetas
Iterator iterator = factory.findAllCreditCards(privateKey);
```

```
// Mostrar las tarjetas
while(iterator.hasNext()) {
   CreditCard creditCard = (CreditCard)iterator.next();
   System.out.println("\nAccount ID: "+creditCard.getAccountID());
   System.out.println("CC Number: "+creditCard.getCreditCardNumber());
}
...
```

Para probar esta aplicación teclearemos

java com.projavasecurity.ecommerce.ViewTest