

PRÁCTICA 1

SEGURIDAD Y AUDITORÍA DE LOS SISTEMAS DE INFORMACIÓN

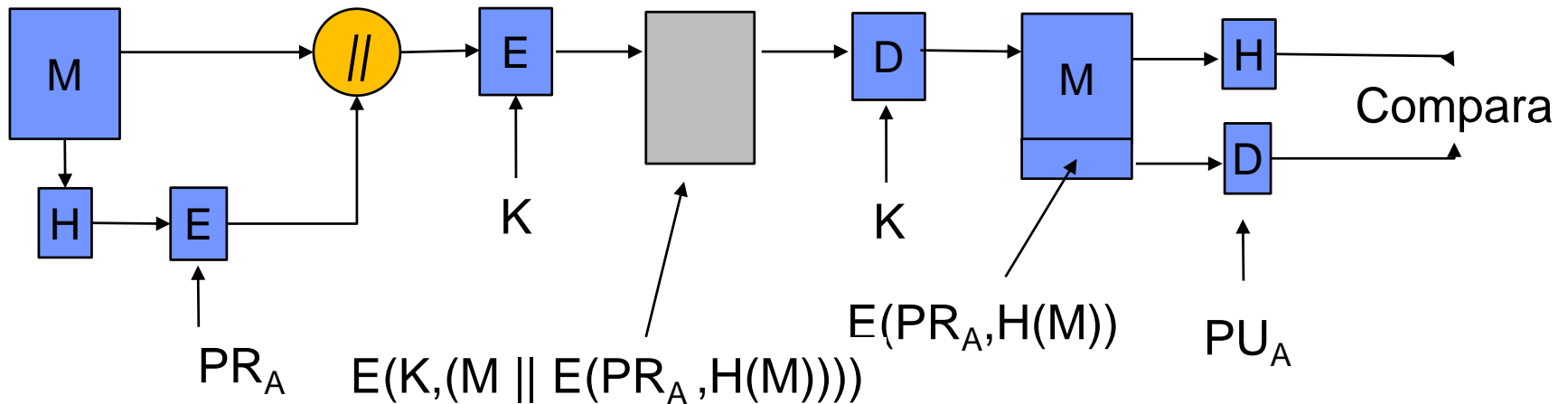
*Metodologías fundamentales para
proporcionar seguridad y privacidad*

Seguridad con Java: Práctica 1

- El objetivo de esta práctica es familiarizarse un poco con la arquitectura de las librerías de seguridad en java y su sistema de proveedores específicos de criptografía y seguridad.
- Para ello como se van a estudiar las diferentes funciones criptográficas básicas, y como referencia se recomienda:
 - Java Cryptography Architecture (JCA) Reference Guide:
 - <https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html>.
 - D. Hook, “Beginning Cryptography with Java”, pero ojo que utiliza **Bouncy Castle** :
 - Los ejemplos del libro los podemos encontrar en:
<http://www.wrox.com/WileyCDA/WroxTitle/Beginning-Cryptography-with-Java.productCd-0764596330.html>
- El objetivo es entender los códigos básicos que os pongo a continuación y extenderlos con otros algoritmos criptográficos básicos.
- Por último hay que implementar los dos protocolos de la siguientes transparencias.
- **Hay que presentar una memoria (en formato pdf), además de los códigos utilizados en la práctica. Así en la memoria se debe detallar el funcionamiento de los códigos java utilizados, junto con las variantes creadas de los mismos para otros algoritmos, y la implementación de los dos protocolos.**
- La entrega se realizará en el moodle de la asignatura.

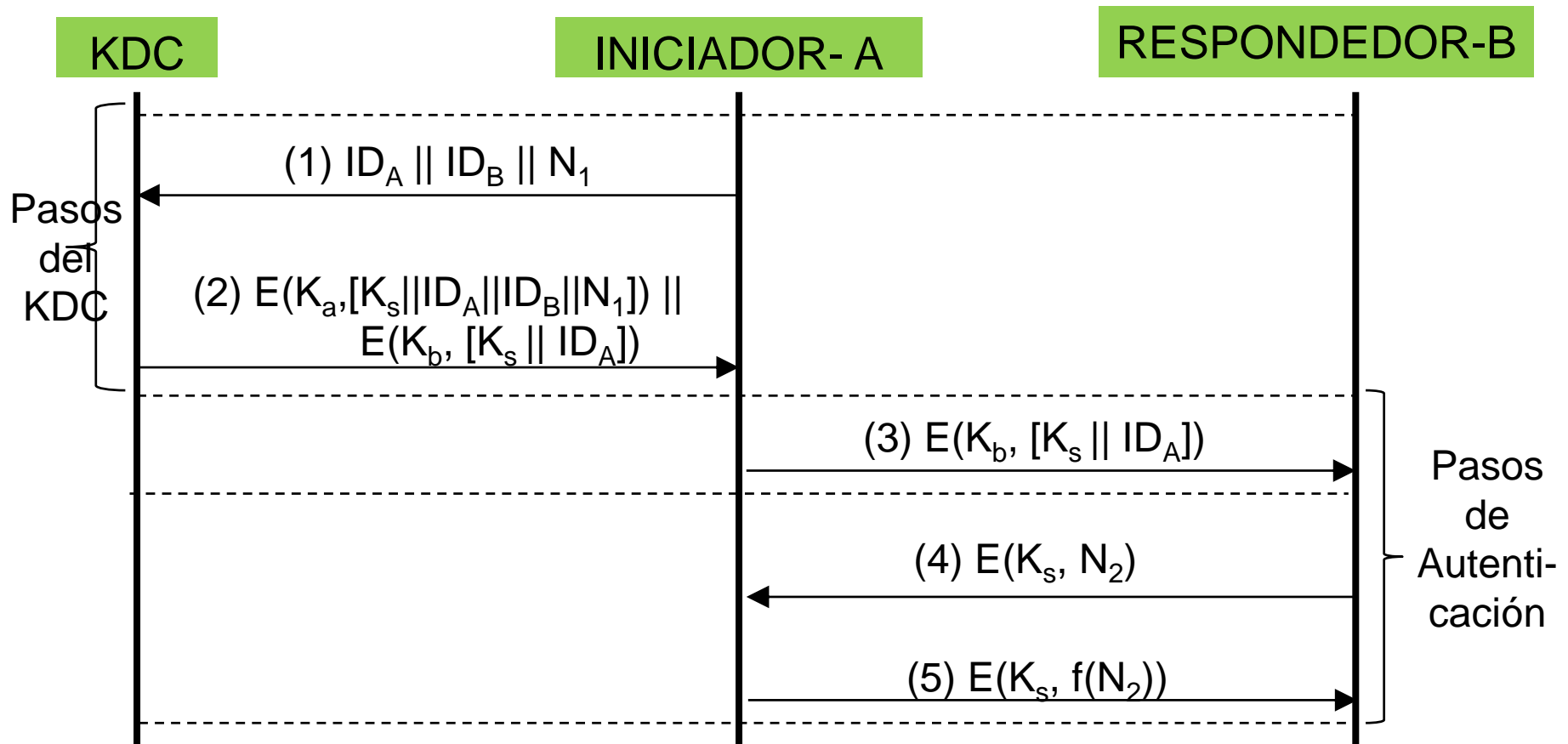
Seguridad con Java: Práctica 1

- **Primer protocolo** para implementar (perturbar el mensaje en el canal inseguro para comprobar el funcionamiento del protocolo implementado):



Seguridad con Java: Práctica 1

- **Segundo protocolo** para implementar de intercambio de claves (analizar la seguridad del protocolo con detalle):



Seguridad con Java: Práctica 1

- Códigos que os pueden ayudar en la práctica:
 - PolicyTest.java
 - ListAlgorithms.java
 - Utils.java
 - SimpleSymmetricExample.java
 - SimpleSymmetricPaddingExample.java
 - EjemploAESGCM.java
 - TamperedExample.java
 - TamperedWithDigestExample.java
 - BaseRSAExample.java

Seguridad con Java

- **FUNDAMENTAL:** Java Cryptography Architecture (JCA) Reference Guide:
 - <https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html>.
- Libro interesante: D. Hook, “Beginning Cryptography with Java”, pero ojo que utiliza **Bouncy Castle** :
 - Los ejemplos del libro los podéis encontrar en:
<http://www.wrox.com/WileyCDA/WroxTitle/Beginning-Cryptography-with-Java.productCd-0764596330.html>
- Antes de nada hay que hacer algunas cosas en la JDK de java.
 - Instalar la política no restrictiva de seguridad (JRE) en versiones antiguas, hoy en día no hace falta con las últimas versiones.
 - Si queréis algún proveedor que no venga lo podéis instalar también.

Proveedores por defecto de java

➤ Por ejemplo en W10 en Program Files\Java\jdk-19\conf\security**java.security**

security.provider.1=SUN

security.provider.2=SunRsaSign

security.provider.3=SunEC

security.provider.4=SunJSSE

security.provider.5=SunJCE

security.provider.6=SunJGSS

security.provider.7=SunSASL

security.provider.8=XMLDSig

security.provider.9=SunPCSC

security.provider.10=JdkLDAP

security.provider.11=JdkSASL

security.provider.12=SunMSCAPI

security.provider.13=SunPKCS11

Seguridad con Java: política no restrictiva de seguridad (JCE)

- Instalar la política no restrictiva de seguridad (JCE): por a las leyes de los Estados Unidos, que prohíben exportar software de encriptación de datos, el JCE no venían incluido inicialmente en las versiones del JDK, y existían restricciones para bajarlo de SUN.
- Existían paquetes desarrollados fuera de los Estados Unidos, que implementan todas las especificaciones del JCE y que no están sujetos a sus restricciones legales. Ahora se puede bajar de Oracle, por ejemplo está en:
<http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html>
- OJO: cada versión de java tiene una versión de JCE no restrictiva.
- Lo que se suele hacer es (es solo un ejemplo para una versión determinada):
 - Descomprimir jce_policy-8.zip y copiar local_policy.jar y US_export_policy.jar en /usr/lib/jvm/java-8-oracle/jre/lib/security (suponiendo que tengo java8 y haciendo una copia de seguridad antes), o en Windows a C:\Program Files\Java\jre1.8.0_25\lib\security y C:\Program Files\Java\jdk1.8.0_25\jre\lib\security
 - Me aseguro que todos los ficheros copiado tengan permiso de lectura para todos.

Seguridad con Java: política no restrictiva de seguridad (JCE): Un poco de historia.

- Fuerza criptográfica: la dificultad de descubrir la clave, que depende del cifrado utilizado y la longitud de la clave (una clave más larga proporciona un cifrado más sólido):
 - La fuerza criptográfica limitada en JCE utiliza una clave máxima de 128 bits.
 - El ilimitado utiliza una clave de longitud máxima de 2147483647 bits.
- Solo versiones antiguas del JRE no incluyen los archivos de política de fuerza ilimitada (versiones JRE 8u151, que corresponde al java SE 8 y anteriores): porque algunos países requieren fortalezas criptográficas restringidas. En caso de que la ley de un país permita una fuerza criptográfica ilimitada, es posible agruparla o habilitarla según la versión de Java.
- En mi caso no me hace falta (tengo el java SE 19):

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\ > java -version
java version "19.0.2" 2023-01-17
Java(TM) SE Runtime Environment (build 19.0.2+7-44)
Java HotSpot(TM) 64-Bit Server VM (build 19.0.2+7-44, mixed mode, sharing)
```

Seguridad con Java: política no restrictiva de seguridad (JCE): Un poco de historia.

- En el siguiente código verificamos la fuerza criptográfica.
 - Mediante un algoritmos criptográfico con clave grande, o
 - Comprobando la longitud máxima de clave permitida:
 - `int maxKeySize = javax.crypto.Cipher.getMaxAllowedKeyLength("AES");`
 - Que devuelve 128, en caso de archivos de política limitados.
 - O devuelve 2147483647 cuando la JCE utiliza archivos de políticas ilimitados.
- Como hemos comentado, las versiones de Java 8u151 y anteriores contienen los archivos de políticas en el directorio `JAVA_HOME/jre/lib/security`.
- A partir de la versión 8u151, el JRE proporciona diferentes conjuntos de archivos de políticas. Como resultado, en el directorio `JAVA_HOME/conf/security/policy` hay 2 subdirectorios: limitado e ilimitado.
 - El primero contiene archivos de políticas de fuerza limitada.
 - El segundo contiene ilimitados.

Seguridad con Java: política no restrictiva de seguridad (JCE)

- Compruebo sus funcionamiento mediante PolicyTest.java, antes y después de copiar la JCE sin restricciones (en las versiones modernas funciona por defecto):

```
/**
 * Sacado de D. Hook, "Beginning Cryptography with Java" y adaptado
 */

package com.mycompany.policytest;

import java.security.GeneralSecurityException;
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;

/**
 * A class that does some basic cryptographic operations to confirm what
 * policy restrictions exist in the Java runtime it is running in.
 */
public class PolicyTest
{
    public static void main(String[] args)
    {
        byte[] data = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 };

        SecretKeySpec key64 = new SecretKeySpec(
            new byte[] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 },
            "Blowfish");
    }
}
```

Seguridad con Java: política no restrictiva de seguridad (JCE)

```
try
{
    Cipher      c = Cipher.getInstance("Blowfish/ECB/NoPadding");

    c.init(Cipher.ENCRYPT_MODE, key64);

    c.doFinal(data);

    System.out.println("64 bit test: passed");
}
catch (SecurityException e)
{
    if (e.getMessage() == "Unsupported keysize or algorithm parameters")
    {
        System.out.println("64 bit test failed: unrestricted policy files have not been installed for this runtime");
    }
    else
    {
        System.err.println("64 bit test failed: there are bigger problems than just policy files: " + e);
    }
}
catch (GeneralSecurityException e)
{
    System.err.println("64 bit test failed: there are bigger problems than just policy files: " + e);
}
```

Seguridad con Java: política no restrictiva de seguridad (JCE)

```
SecretKeySpec key192 = new SecretKeySpec(  
    new byte[] { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,  
                0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,  
                0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 },  
    "Blowfish");
```

```
try  
{  
    /* primera forma de probar la política restrictiva de claves */  
    Cipher c = Cipher.getInstance("Blowfish/ECB/NoPadding");  
  
    c.init(Cipher.ENCRYPT_MODE, key192);  
  
    c.doFinal(data);  
  
    System.out.println("192 bit test: passed");  
  
    /* segunda forma de probar la política restrictiva de claves */  
    int maxKeySize = javax.crypto.Cipher.getMaxAllowedKeyLength("AES");  
    System.out.println("maxKeySize: " + maxKeySize);  
}
```

```
catch (SecurityException e)  
{  
    if (e.getMessage() == "Unsupported keysize or algorithm parameters")  
    {  
        System.out.println("192 bit test failed: unrestricted policy files have not been installed for this runtime.");  
    }  
    else  
    {  
        System.err.println("192 bit test failed: there are bigger problems than just policy files: " + e);  
    }  
}  
catch (GeneralSecurityException e)  
{  
    System.err.println("192 bit test failed: there are bigger problems than just policy files: " + e);  
}  
  
System.out.println("Tests completed");  
}
```

- La ejecución de este programa tiene que dar:

64 bit test: passed

192 bit test:passed

Tests completed

- Este código esta en el capítulo 1 del libro de java.
- Recordar que el objetivo del bloque try-catch es detectar y controlar una excepción generada por código en funcionamiento.

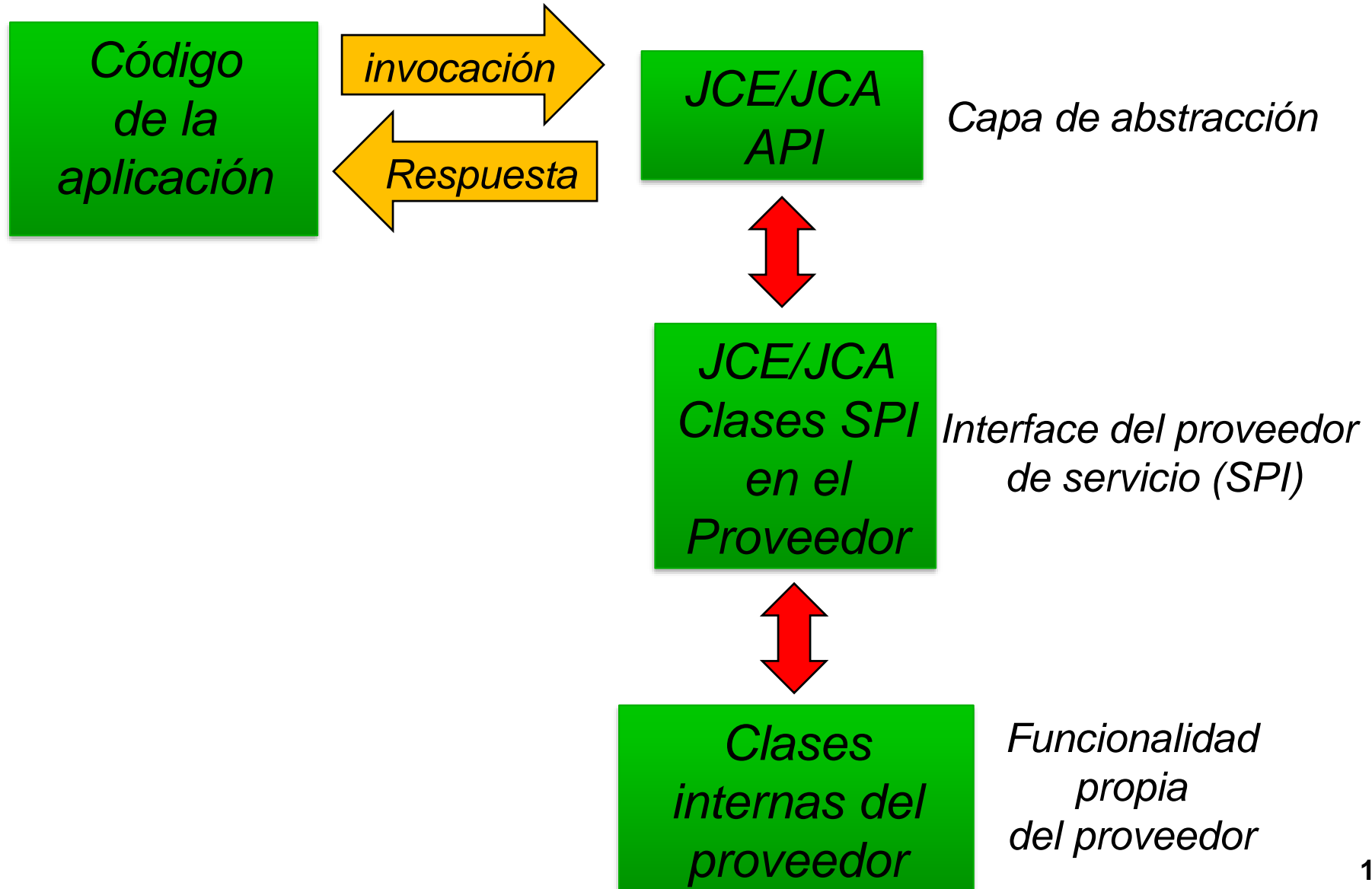
Seguridad con Java:

Las Librerías JCA y JCE

- Debido a las **restricciones** de exportación de software criptográfico que existían en EEUU cuando se diseñaron las librerías criptográficas de Java, JavaSoft descompuso sus API criptográficas en dos librerías:
 - **La librería JCA** (Java Cryptography Architecture, paquete: `java. security.*`): forma parte del runtime JVM y contiene elementos que no estaban sujetos a las restricciones de exportación de EEUU (firmas digitales, funciones Hash, etc).
 - **La librería JCE** (Java Cryptography Extensions , paquete: `javax. crypto. *`): se incluyen funcionalidades criptográficas con restricciones a exportación del gobierno de los EEUU.
 - Solo se podía obtener en EEUU y Canadá, así que otros fabricantes de otros países se pusieron a hacer implementaciones de la librería JCE que distribuían gratuitamente al resto del mundo a través de Internet, como por ejemplo (nosotros utilizaremos la librería por defecto):
 - <http://www.bouncycastle.org/>
 - <http://www.cryptix.org> (hoy en día RIP)
- A partir de la JDK 1.4 cambiaron las leyes y Java (hoy absorbida por Oracle) empezó a exportar por defecto esas funcionalidades de seguridad. Es decir la librería JCE se distribuye ya como una extensión estándar (sus clases `javax. crypto. *` tienen el mismo privilegio que las clases `java. security.*`).

Seguridad con Java:

Las Librerías JCA y JCE



Seguridad con Java: Librerías JCA y JCE

- La JRE contiene la funcionalidad de cifrado en sí. La JCE utiliza archivos de políticas de jurisdicción para controlar la fortaleza criptográfica.
- Como hemos visto, los archivos de políticas constan de dos archivos jar: local_policy.jar y US_export_policy.jar. Gracias a eso, la plataforma Java tiene un control incorporado de la fuerza criptográfica.
- Así, la JCE que se **distribuye** impone cierta política a los tamaños de clave que se pueden utilizar en seguridad. Por ejemplo los tamaños de clave se limitan en general a 128 bits (excepto para el cifrado Triple DES simétrica) y la generación de claves RSA se limita a 2048 bits (esto va cambiando con el tiempo).
- Esto se soluciona descargando los archivos de política sin restricciones, por ejemplo de Oracle (ya lo hemos contado antes):
<http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html>
- **JCA** está integrada por dos tipos de clases:
 - Las clases API de la librería JCA, que son clases que actúan como interfaces en las que se definen una serie de operaciones estándar.
 - Los proveedores de seguridad, que son terceras partes que implementan esas interfaces con algoritmos criptográficos propios.
- **Engine o servicio**: cada una de las operaciones criptográficas que proporcionan las librerías criptográficas de Java, (funciones hash, encriptación, firmas digitales, etc.).

Seguridad con Java:

Las Librerías JCA y JCE

- Cada **servicio** está representado por una clase abstracta (por ejemplo *MessageDigest*, *Cipher*, *Signature*, etc.).
- La implementación (**algoritmo** específico de la clase abstracta) de estos servicios de seguridad se realiza a través de clase derivadas implementadas los diferentes proveedores de seguridad,
 - por defecto SUN o,
 - se puede instalar por ejemplo “The Legion of the Bouncy Castle”.
- Es importante entender que la relación entre las clases abstractas y las clases de los proveedores **no es una relación 1:1**, ya que por ejemplo para la clase abstracta *MessageDigest* hay diferentes implementaciones según el proveedor: *CryptixSHA1MessageDigest*, *SunMessageDigest*, *CryptixMD5MessageDigest*.
- Así el mismo servicio se puede implementar con varios algoritmos y se separan las interfaces del servicio de los proveedores (así se evitan problemas de exportación).

Seguridad con Java:

Las Librerías JCA y JCE

- En la clase abstracta siempre encontramos el método estático `getInstance()` al cual podemos pedir un objeto que implemente esa misma clase, indicando el algoritmo y el proveedor. Por ejemplo BC (que hay que instalarlo):
 - `MessageDigest md = MessageDigest.getInstance("SHA1","BC");`
- Así para calcular el hash de un mensaje podríamos hacer:
 - Creamos los datos de los que queremos calcular el hash:
 - `byte[] dataBytes = "This is test data".getBytes();`
 - Instanciamos la clase `MessageDigest` utilizando un algoritmo concreto dado por el proveedor:
 - `MessageDigest md = MessageDigest.getInstance("SHA1","BC");`
 - Refrescamos *md* con los datos de entrada para crear el *digest*:
 - `md.update(dataBytes);`
 - `byte[] digest = md.digest();`
- **OJO:** si no se pone ningún proveedor, el método `getInstance()` elige el primer proveedor de la lista que implemente ese algoritmo, contenido en el fichero `/usr/lib/jvm/java-7-oracle/jre/lib/security/java.security` que modificamos, o donde se encuentre la “jre (Java Runtime Environment)” que se utiliza (este caso SUN).

Seguridad con Java: Algoritmos disponibles

- Con el código “ListAlgorithms.java” podéis chequear todos los algoritmos criptográficos instalados.

```
package com.mycompany.listalgorithms;

import java.security.Provider;
import java.security.Security;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class ListAlgorithms
{
    /**
     * Print out the set entries, indented, one per line, with the name of the set
     * unindented appearing on the first line.
     *
     * @param setName the name of the set being printed
     * @param algorithms the set of algorithms associated with the given name
     */
    public static void printSet(
        String setName,
        Set algorithms)
    {
        System.out.println(setName + ":");

        if (algorithms.isEmpty())
        {
            System.out.println("        None available.");
        }
        else
        {
            Iterator it = algorithms.iterator();

            while (it.hasNext())
            {
                String name = (String)it.next();

                System.out.println("        " + name);
            }
        }
    }
}
```

Seguridad con Java: Algoritmos disponibles

➤ Con el código “ListAlgorithms.java” podéis chequear todos los algoritmos criptográficos instalados.

```
/**
 * List the available algorithm names for ciphers, key agreement, macs,
 * message digests and signatures.
 */
public static void main(String[] args)
{
    Provider[] providers = Security.getProviders();
    Set ciphers = new HashSet();
    Set keyAgreements = new HashSet();
    Set macs = new HashSet();
    Set messageDigests = new HashSet();
    Set signatures = new HashSet();

    for (int i = 0; i != providers.length; i++)
    {
        Iterator it = providers[i].keySet().iterator();

        while (it.hasNext())
        {
            String entry = (String)it.next();

            if (entry.startsWith("Alg.Alias."))
            {
                entry = entry.substring("Alg.Alias.".length());
            }
        }
    }
}
```

Seguridad con Java: Algoritmos disponibles

- Con el código “ListAlgorithms.java” podéis chequear todos los algoritmos criptográficos instalados.

```
        if (entry.startsWith("Cipher."))
        {
            ciphers.add(entry.substring("Cipher.".length()));
        }
        else if (entry.startsWith("KeyAgreement."))
        {
            keyAgreements.add(entry.substring("KeyAgreement.".length()));
        }
        else if (entry.startsWith("Mac."))
        {
            macs.add(entry.substring("Mac.".length()));
        }
        else if (entry.startsWith("MessageDigest."))
        {
            messageDigests.add(entry.substring("MessageDigest.".length()));
        }
        else if (entry.startsWith("Signature."))
        {
            signatures.add(entry.substring("Signature.".length()));
        }
    }

    printSet("Ciphers", ciphers);
    printSet("KeyAgreements", keyAgreements);
    printSet("Macs", macs);
    printSet("MessageDigests", messageDigests);
    printSet("Signatures", signatures);
}
```

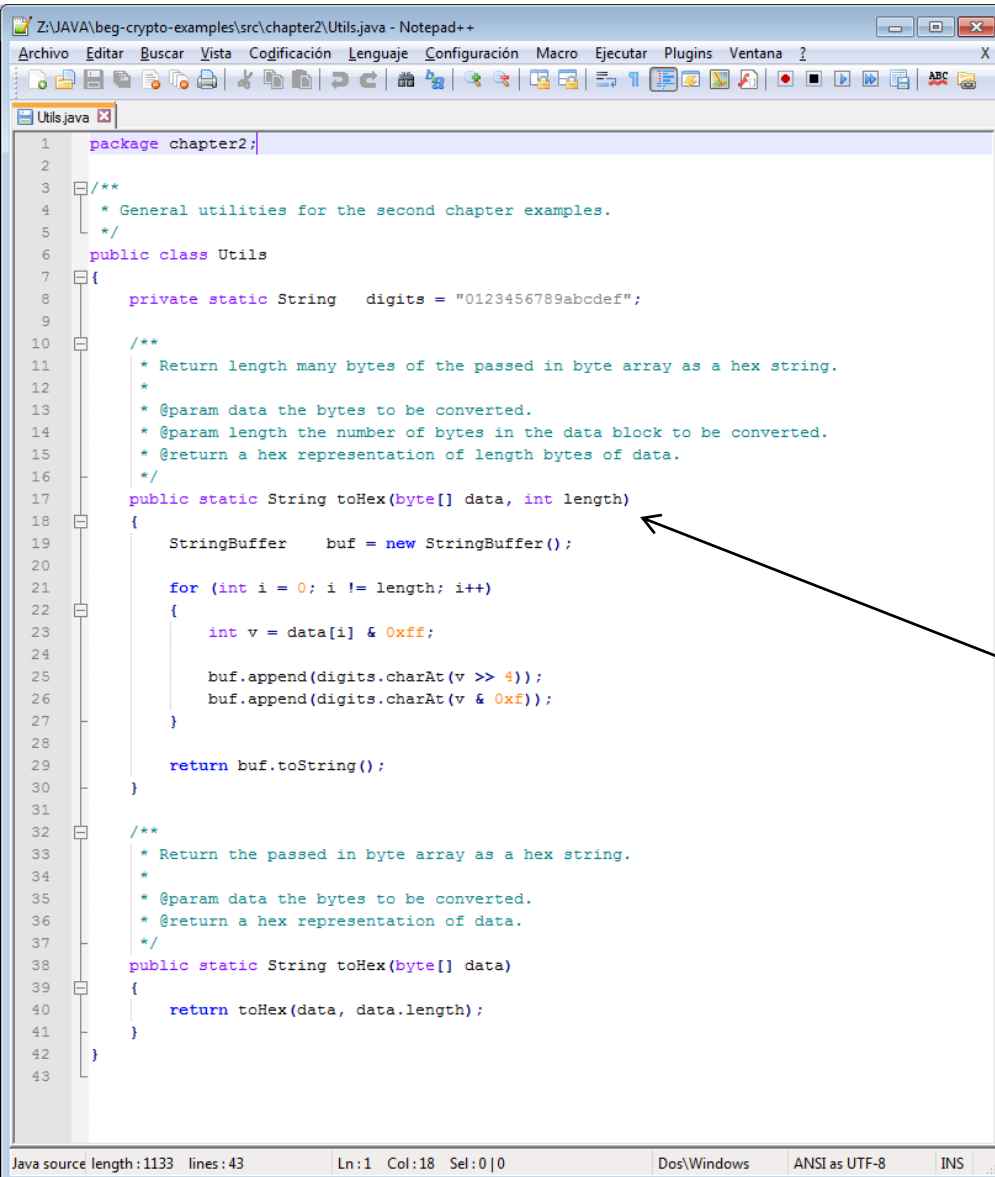
Seguridad con Java: Algoritmos disponibles

Ciphers:

- Que da como salida, la siguiente.
- **Investigar un poco los diferentes algoritmos disponibles.**

```
PBEWithMD5AndDES
2.16.840.1.101.3.4.1
AES/KWP/NoPadding
PBEWithHmacSHA512AndAES_128
AES_128/ECB/NoPadding SupportedKeyFormats
OID.2.16.840.1.101.3.4.1.5
AES_192/CFB/NoPadding
OID.2.16.840.1.101.3.4.1.4
PBEWithSHA1AndRC4_128
OID.2.16.840.1.101.3.4.1.3
PBEWithMD5AndTripleDES
OID.2.16.840.1.101.3.4.1.2
OID.2.16.840.1.101.3.4.1.1
AES_128/KW/NoPadding SupportedKeyFormats
AES SupportedPaddings
OID.2.16.840.1.101.3.4.1.8
OID.2.16.840.1.101.3.4.1.6
RC2
RSA
RC4
AES_128/GCM/NoPadding SupportedKeyFormats
PBEWithHmacSHA512AndAES_256
RC2 SupportedPaddings
AES_256/KW/NoPadding
DES SupportedModes
AES_256/KWP/NoPadding|
.....
.....
```

Seguridad con Java: Cifrados Simétricos



```
1 package chapter2;
2
3 /**
4  * General utilities for the second chapter examples.
5  */
6 public class Utils
7 {
8     private static String digits = "0123456789abcdef";
9
10    /**
11     * Return length many bytes of the passed in byte array as a hex string.
12     *
13     * @param data the bytes to be converted.
14     * @param length the number of bytes in the data block to be converted.
15     * @return a hex representation of length bytes of data.
16     */
17    public static String toHex(byte[] data, int length)
18    {
19        StringBuffer buf = new StringBuffer();
20
21        for (int i = 0; i != length; i++)
22        {
23            int v = data[i] & 0xff;
24
25            buf.append(digits.charAt(v >> 4));
26            buf.append(digits.charAt(v & 0xf));
27        }
28
29        return buf.toString();
30    }
31
32    /**
33     * Return the passed in byte array as a hex string.
34     *
35     * @param data the bytes to be converted.
36     * @return a hex representation of data.
37     */
38    public static String toHex(byte[] data)
39    {
40        return toHex(data, data.length);
41    }
42 }
43
```

- Seguiremos los ejemplos del capítulo 2 del libro : D. Hook, “Beginning Cryptography with Java”.
- Primero resaltar el código **Utils.java** lo único que hace es definir los métodos para imprimir por pantalla en hexadecimal, la estructura interna de los bytes:
 - *public static String toHex(byte[] data, int length),* donde data es un array de bytes, e imprimimos un número de bytes igual a length.
- Ya podéis imaginar que los bytes cifrados son solo leíbles en un ordenador solo **por casualidad**. Por eso hexadecimal.....

Seguridad con Java: Cifrados Simétricos

➤ Ojo con la clase byte de java (recordatorio):

```
class a
{
    public static void main(String[] args)
    {
        // 1 error
        byte a = 0xff;
        System.out.printf("%x\n",a);

        //2 OK
        byte a = (byte)0xff
        System.out.printf("%x\n",a);
    }
}
```

- El tipo de byte Java es un tipo de dato de 8 bits con valores en el intervalo de -128 a +127 (“signed”) . El literal 0xff representa +255 que está fuera de ese rango.
- En el primer ejemplo, está intentando asignar un valor que está fuera de rango a un byte (error).
- En el segundo ejemplo, el cast (byte) realiza una conversión explícita.

Seguridad con Java: Cifrados Simétricos

- ◆ Un ejemplo de cifrado simple con AES de un bloque de 128 bits:
- ◆ SimpleSymmetricExample.java
- ◆ Investigar como cifrar bloque de más de 128 bits.

```
/*
 * Extraído de D. Hook, "Beginning Cryptography with Java" y adaptado
 */
package com.mycompany.simplesymmetricexample;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
```

```
/**
 * Basic symmetric encryption example
 */
public class SimpleSymmetricExample
{
    public static void main(String[] args)
        throws Exception
    {

        byte[] input = new byte[] {
            0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
            (byte)0x88, (byte)0x99, (byte)0xaa, (byte)0xbb,
            (byte)0xcc, (byte)0xdd, (byte)0xee, (byte)0xff };
        byte[] keyBytes = new byte[] {
            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
            0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 };

        // La clase SecretKeySpec acondiciona una clave al tipo
        // de cifrado (en este caso al AES)
        SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");
```

Seguridad con Java: Cifrados Simétricos

- ◆ Un ejemplo de cifrado simple con AES de un bloque de 128 bits:
- ◆ SimpleSymmetricExample.java
- ◆ **Investigar otros modos de operación, además del ECB.**
- ◆ **Estudiar e investigar otros algoritmos simétricos con la librería de java.**

```
// Instanciamos la clase cipher utilizando un algoritmo
// concreto dado por el proveedor BC.
Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");

System.out.println("input text : " + Utils.toHex(input));

// encryption pass

byte[] cipherText = new byte[input.length];

// Inicializamos el cifrador en modo cifrado y con la clave.
cipher.init(Cipher.ENCRYPT_MODE, key);

// Una vez que el objeto cipher está configurado para el cifrado
// se le pasan los datos utilizando el método cipher.update().
int ctLength = cipher.update(input, 0, input.length, cipherText, 0);

ctLength += cipher.doFinal(cipherText, ctLength);

System.out.println("cipher text: " + Utils.toHex(cipherText) + " bytes: " + ctLength);

// decryption pass

byte[] plainText = new byte[ctLength];

// Inicializamos el cifrador en modo descifrado y con la clave.
cipher.init(Cipher.DECRYPT_MODE, key);

int ptLength = cipher.update(cipherText, 0, ctLength, plainText, 0);

ptLength += cipher.doFinal(plainText, ptLength);

System.out.println("plain text : " + Utils.toHex(plainText) + " bytes: " + ptLength);
}
```

Seguridad con Java:

Cifrados Simétricos, Padding

- En el código **SimpleSymmetricPaddingExample.java** tenemos la forma habitual de añadir padding.
- En los documentos **PKCS #5** el *padding* es igual a un número que corresponde con el número de bytes a añadir.
- **PKCS #5 padding** (comparar con *NoPadding* del código anterior).
- Se calcula el tamaño en bytes del texto cifrado con *padding*.

```
/*
 * Extraído de D. Hook, "Beginning Cryptography with Java" y adaptado
 */
package com.mycompany.simplesymmetricpaddingexample;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;

public class SimpleSymmetricPaddingExample
{
    public static void main( String[] args)
        throws Exception
    {
        // Texto a cifrar 24 bytes/192 bits (un bloque y medio de AES, se necesita padding!!!)
        byte[] input = new byte[] {
            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
            0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 };
        // Clave de 192 bits
        byte[] keyBytes = new byte[] {
            0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
            0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
            0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17 };

        SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");

        // Si necesita utilizar padding, podemos utilizar el estandar: PKCS5
        // https://en.wikipedia.org/wiki/PKCS
        // Supongamos un bloque de datos donde el último bloque de entrada es
        // 3 bytes más corto que el tamaño de bloque del cifrado que está utilizando,
        // entonces se agregan 3 bytes de valor 3 a los datos antes de cifrarlo.

        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
```

Seguridad con Java:

Cifrados Simétricos, Padding

- Estudiar otros tipos de “padding” que existan en la librería de java.

```
System.out.println("input : " + Utils.toHex(input));

// encryption pass

cipher.init(Cipher.ENCRYPT_MODE, key);

byte[] cipherText = new byte[cipher.getOutputSize(input.length)];

int ctLength = cipher.update(input, 0, input.length, cipherText, 0);

ctLength += cipher.doFinal(cipherText, ctLength);

System.out.println("cipher: " + Utils.toHex(cipherText) + " bytes: " + ctLength);

// decryption pass

cipher.init(Cipher.DECRYPT_MODE, key);

byte[] plainText = new byte[cipher.getOutputSize(ctLength)];

int ptLength = cipher.update(cipherText, 0, ctLength, plainText, 0);

ptLength += cipher.doFinal(plainText, ptLength);

System.out.println("plain : " + Utils.toHex(plainText) + " bytes: " + ptLength);

}
```

Seguridad con Java:

Cifrados Simétricos, AES en GCM

```
/*
 * Ejemplo de AES en modo GCM
 */
package com.mycompany.ejemploaesgcm;

import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.util.Base64;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.spec.GCMParameterSpec;
import javax.crypto.spec.SecretKeySpec;

public class EjemploAESGCM {
    static String plainText = "Esto es una prueba del modo de cifrado GCM para el AES";
    public static final int AES_KEY_SIZE = 256;
    public static final int GCM_IV_LENGTH = 12;
    public static final int GCM_TAG_LENGTH = 16;

    public static void main(String[] args) throws NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException,
        InvalidAlgorithmParameterException, IllegalBlockSizeException, BadPaddingException {

        // inicializo para generar la clave
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        kg.init(AES_KEY_SIZE);
```

Seguridad con Java:

Cifrados Simétricos, AES en GCM

```
// genero la clave
SecretKey key = kg.generateKey();
byte[] IV = new byte[GCM_IV_LENGTH];
SecureRandom random = new SecureRandom();
random.nextBytes(IV);

// Salida del texto plano
System.out.println("Texto Plano : " + plainText);

// Objeto para cifrar y descifrar
// AES-GCM es un modo de operación de cifrado de bloques
// que proporciona alta velocidad de cifrado autenticado
// e integridad de datos. En el modo GCM, el cifrado de
// bloque se transforma en cifrado de flujo y, por lo tanto,
// no se necesita "padding".
Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");

// Creo las especificaciones de la clave
SecretKeySpec keySpec = new SecretKeySpec(key.getEncoded(), "AES");

// Creo las especificaciones del modo GCM
GCMParameterSpec gcmParameterSpec = new GCMParameterSpec(GCM_TAG_LENGTH * 8, IV);
```

Seguridad con Java:

Cifrados Simétricos, AES en GCM

```
//----- Cifrar -----  
// Inicializo el objeto difrador en modo cifrado  
cipher.init(Cipher.ENCRYPT_MODE, keySpec, gcmParameterSpec);  
  
// Cifro  
byte[] cipherText = cipher.doFinal(Utils.toByteArray(plainText));  
  
// Salida del texto cifrado en hexadecimal  
System.out.println("Texto Cifrado (hexadecimal): " + Utils.toHex(cipherText));  
  
//----- DesCifrar -----  
// Initialize Cipher for DECRYPT_MODE  
cipher.init(Cipher.DECRYPT_MODE, keySpec, gcmParameterSpec);  
  
// Perform Decryption  
byte[] decryptedText = cipher.doFinal(cipherText);  
  
// Salida del texto descifrado  
System.out.println("Texto Descifrado: " + Utils.toString(decryptedText));  
}
```

Seguridad con Java:

Cifrados Asimétricos y Hash

```
1 package chapter3;
2
3 import java.security.NoSuchAlgorithmException;
4 import java.security.NoSuchProviderException;
5 import java.security.SecureRandom;
6
7 import javax.crypto.KeyGenerator;
8 import javax.crypto.SecretKey;
9 import javax.crypto.spec.IvParameterSpec;
10
11 /**
12  * General utilities for the third chapter examples.
13  */
14 public class Utils
15     extends chapter2.Utils
16 {
17     /**
18      * Create a key for use with AES.
19      *
20      * @param bitLength
21      * @param random
22      * @return an AES key.
23      * @throws NoSuchAlgorithmException
24      * @throws NoSuchProviderException
25      */
26     public static SecretKey createKeyForAES(
27         int bitLength,
28         SecureRandom random)
29         throws NoSuchAlgorithmException, NoSuchProviderException
30     {
31         KeyGenerator generator = KeyGenerator.getInstance("AES" );
32
33         generator.init(256, random);
34
35         return generator.generateKey();
36     }
37 }
```

- En el código **Utils.java** tenemos ciertas utilidades que usaremos.
- El método *createKeyForAES* crea una clave para el cifrado AES (hasta ahora las habíamos puesto en el código).
- Utiliza para ello la clase *KeyGenerator*:
 - Instancia *generator*
 - *KeyGenerator.getInstance*
 - Inicializa
 - *generator.init*
 - Genera la clave
 - *generator.generateKey*

Seguridad con Java:

Cifrados Asimétricos y Hash

```
38 /**
39  * Create an IV suitable for using with AES in CTR mode.
40  * <p>
41  * The IV will be composed of 4 bytes of message number,
42  * 4 bytes of random data, and a counter of 8 bytes.
43  *
44  * @param messageNumber the number of the message.
45  * @param random a source of randomness
46  * @return an initialised IvParameterSpec
47  */
48 public static IvParameterSpec createCtrIvForAES(
49     int            messageNumber,
50     SecureRandom   random)
51 {
52     byte[]         ivBytes = new byte[16];
53
54     // initially randomize
55
56     random.nextBytes(ivBytes);
57
58     // set the message number bytes
59
60     ivBytes[0] = (byte) (messageNumber >> 24);
61     ivBytes[1] = (byte) (messageNumber >> 16);
62     ivBytes[2] = (byte) (messageNumber >> 8);
63     ivBytes[3] = (byte) (messageNumber >> 0);
64
65     // set the counter bytes to 1
66
67     for (int i = 0; i != 7; i++)
68     {
69         ivBytes[8 + i] = 0;
70     }
71
72     ivBytes[15] = 1;
73
74     return new IvParameterSpec(ivBytes);
75 }
```

- En el código **Utils.java** tenemos ciertas utilidades que usaremos.
- El método *createCtrIvForAES* crea un IV para el uso en CTR en el cifrado AES.
- El IV se compone de 4 bytes de número de mensaje, 4 bytes de datos aleatorios, y un contador de 8 bytes.
- Esta es la estructura que acepta el modo CTR de AES.

Seguridad con Java:

Cifrados Asimétricos y Hash

```
78 * Convert a byte array of 8 bit characters into a String.
79 *
80 * @param bytes the array containing the characters
81 * @param length the number of bytes to process
82 * @return a String representation of bytes
83 */
```

```
84 public static String toString(
85     byte[] bytes,
86     int length)
87 {
88     char[] chars = new char[length];
89
90     for (int i = 0; i != chars.length; i++)
91     {
92         chars[i] = (char)(bytes[i] & 0xff);
93     }
94
95     return new String(chars);
96 }
97
```

```
98 /**
99 * Convert a byte array of 8 bit characters into a String
100 *
101 * @param bytes the array containing the characters
102 * @return a String representation of bytes
103 */
```

```
104 public static String toString(
105     byte[] bytes)
106 {
107     return toString(bytes, bytes.length);
108 }
109
```

```
110 /**
111 * Convert the passed in String to a byte array by
112 * taking the bottom 8 bits of each character it contains.
113 *
114 * @param string the string to be converted
115 * @return a byte array representation
116 */
```

```
117 public static byte[] toByteArray(
118     String string)
119 {
120     byte[] bytes = new byte[string.length()];
121     char[] chars = string.toCharArray();
122
123     for (int i = 0; i != chars.length; i++)
124     {
125         bytes[i] = (byte)chars[i];
126     }
127
128     return bytes;
129 }
```

- En el código **Utils.java** tenemos ciertas utilidades que usaremos.
- El método *toString* convierte un vector byte de 8 caracteres bit a una cadena.
- Y El método *toByteArray* convierte una cadena a un vector byte, cogiendo los 8 primeros bits de cada carácter.
- El objeto *String* en Java tiene un método llamado *getBytes()* y un constructor que toma una matriz de bytes que parece hacer lo mismo.
- Pero con estos métodos de *String* la conversión de *String* a *Byte* y la conversión de *Byte* a *String* se ve afectada por el *default charset* de la JVM que se está utilizando.

Modo CTR y Manipulación

(*TamperedExample.java*)

```
/*
 * Extraído de D. Hook, "Beginning Cryptography with Java" y adaptado
 */

package com.mycompany.tamperedexample;

import java.security.Key;
import java.security.SecureRandom;

import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;

/**
 * Tampered message, plain encryption, AES in CTR mode
 */
public class TamperedExample
{
    // Aquí tenemos un ejemplo de control de manipulación de un mensaje:
    //-Cifrado
    //-Manipulación (por atacante)
    //-Descifrado

    public static void main(
        String[] args)
        throws Exception
    {
        SecureRandom random = new SecureRandom();
        IvParameterSpec ivSpec = Utils.createCtrIvForAES(1, random);
        Key key = Utils.createKeyForAES(256, random);
        Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
        String input = "Transfer 0000100 to AC 1234-5678";

        System.out.println("input : " + input);
    }
}
```

Modo CTR y Manipulación (*TamperedExample.java*)

```
// encryption step
// Aquí diframos el mensaje: "Transfer 0000100 to AC 1234-5678"
// En el modo CTR
cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);

byte[] cipherText = cipher.doFinal(Utils.toByteArray(input));

// tampering step
// Aquí manipulamos el mensaje
// Sabemos la estructura del mensaje: el byte 9 es el dígito inicial
// de la cantidad a transferir. Sabemos que está a cero lo podemos poner
// a nueve:
// cipherText[9] ^= '0' ^ '9': cuando se descifre el mensaje, el '0'
// que debería dar en esa posición se combina con '0' ^ '9' quedando
// un 9 en esa posición. Recordar que A xor A= 0, y el operador
// "^=" es "xor" sobre los bits y asignación.

cipherText[9] ^= '0' ^ '9';

// decryption step

cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);

byte[] plainText = cipher.doFinal(cipherText);

// Obviamente se puede hacer por el modo de operación utilizado CTR.
// Así se recibe: Transfer 9000100 to AC 1234-5678 en vez de
// Transfer 0000100 to AC 1234-5678.
System.out.println("plain : " + Utils.toString(plainText));
}
```



Explicar en la memoria porque esta manipulación funciona.

Modo CTR, Manipulación, Hash (*Tampered With Digest Example*)

```
/**
 * Extraído de D. Hook, "Beginning Cryptography with Java" y adaptado
 */

package com.mycompany.tamperedwithdigestexample;

import java.security.Key;
import java.security.MessageDigest;
import java.security.SecureRandom;

import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;

/**
 * Tampered message, encryption with digest, AES in CTR mode
 */
public class TamperedWithDigestExample
{
    public static void main(
        String[] args)
        throws Exception
    {
        SecureRandom random = new SecureRandom();
        IvParameterSpec ivSpec = Utils.createCtrIvForAES(1, random);
        Key key = Utils.createKeyForAES(256, random);
        Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
        String input = "Transfer 0000100 to AC 1234-5678";
        MessageDigest hash = MessageDigest.getInstance("SHA3-256");

        System.out.println("input : " + input);
    }
}
```

Modo CTR, Manipulación, Hash (*Tampered With Digest Example*)

```
// encryption step

cipher.init(Cipher.ENCRYPT_MODE, key, ivSpec);

byte[] cipherText = new byte[cipher.getOutputSize(input.length() + hash.getDigestLength())];

int ctLength = cipher.update(Utls.toByteArray(input), 0, input.length(), cipherText, 0);

hash.update(Utls.toByteArray(input));

ctLength += cipher.doFinal(hash.digest(), 0, hash.getDigestLength(), cipherText, ctLength);

// tampering step

cipherText[9] ^= '0' ^ '9';

// decryption step

cipher.init(Cipher.DECRYPT_MODE, key, ivSpec);

byte[] plainText = cipher.doFinal(cipherText, 0, ctLength);
int messageLength = plainText.length - hash.getDigestLength();

hash.update(plainText, 0, messageLength);

byte[] messageHash = new byte[hash.getDigestLength()];
System.arraycopy(plainText, messageLength, messageHash, 0, messageHash.length);

System.out.println("plain : " + Utls.toString(plainText, messageLength) + " verified: "
    + MessageDigest.isEqual(hash.digest(), messageHash));
}
```

- Explicar en la memoria porque esta manipulación NO funciona.
- Estudiar otros tipos de funciones Hash y Mac en java.

Cifrado Asimétrico, RSA

(BaseRSAExample)

```
/*
 * Extraído de D. Hook, "Beginning Cryptography with Java" y adaptado
 */

package com.mycompany.baseraexample;

import java.math.BigInteger;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.RSAPrivateKeySpec;
import java.security.spec.RSAPublicKeySpec;

import javax.crypto.Cipher;

/**
 * Basic RSA example.
 */
public class BaseRSAExample
{
    public static void main(
        String[] args)
        throws Exception
    {
        byte[] input = new byte[] { (byte)0xbe, (byte)0xef };
        Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");

        // Creando un objeto generador de KeyPair para RSA
        KeyPairGenerator keyPairGen = KeyPairGenerator.getInstance("RSA");
```


Cifrado Asimétrico, RSA

(BaseRSAExample)

```
// Inicializar el generador de pares de claves
keyPairGen.initialize(2048); // mínimo 512 bits

// Generar el par de claves
KeyPair pair = keyPairGen.generateKeyPair();

// Obtener la clave pública del par de claves
PublicKey pubKey = pair.getPublic();

// Obtener la clave privada del par de claves
PrivateKey privKey = pair.getPrivate();

System.out.println("input : " + Utils.toHex(input));

// Inicializando un objeto Cipher para cifrado o descifrado
// Orden priv->pub
// cipher.init(Cipher.ENCRYPT_MODE, privKey);
// Orden pub->priv
cipher.init(Cipher.ENCRYPT_MODE, pubKey);

byte[] cipherText = cipher.doFinal(input);

System.out.println("cipher: " + Utils.toHex(cipherText));

// Inicializando el mismo objeto Cipher para descifrado o cifrado
// Orden priv->pub
// cipher.init(Cipher.DECRYPT_MODE, pubKey);
// Orden pub->priv
cipher.init(Cipher.DECRYPT_MODE, privKey);

byte[] plainText = cipher.doFinal(cipherText);

System.out.println("plain : " + Utils.toHex(plainText));
```

- Estudiar e investigar otros esquemas de cifrados asimétricos en java, y además el intercambio de claves DH.