CRIPTOGRAFÍA EN ENTORNOS JAVA

1. JCA (Java Cryptography Architecture)

La JCA es parte del lenguaje Java, como parte del API de seguridad, desde la versión JDC 1.1. Es una especificación del lenguaje que especifica interfaces y clases abstractas que sirven de base para las implementaciones concretas.de algoritmos criptográficos.

Está diseñada de acuerdo con dos principios:
- independencia e interoperabilidad de las implementaciones
- independencia y extensibilidad de los algoritmos.

La independencia de las implementaciones se consigue empleando una arquitectura basada en proveedores (**providers**). Un proveedor se refiere a un paquete oconjunto de paquetes que proporciona una implementación concreta de funcionalidades criptograficas de la API de seguidad de Java.

La interoperabilidad de las implementaciones permite que cada una de ellas pueda usarse con las demás.

Algunos ejemplos de **providers**:

- **SunJCE** (Java Cryptography Extension)
- **BC** (Proyecto Bouncy Castle)

La independencia de los algoritmos se consigue definiendo ***tipos de servicios criptográficos*** y las clases (**engines**) que proporcionen la funcionalidad de estos servicios.

La extensibilidad de los algoritmos establece que los nuevos algoritmos que se incorporen dentro de alguno de los tipos soportados puedan ser añadidos fácilmente.

Algunos ejemplos de **engines**:

- `SecureRandom`
- `MessageDigest`
- `Signature`
- `Cipher`
- `Mac`
- `KeyFactory`
- `SecretKeyFactory`
- `KeyPairGenerator`
- `KeyGenerator`
- `KeyAgreement`

2. Instalación de los proveedores.

El proveedor `SunJCE` es el proveedor estándar de la distribución Java, por lo cual no es necesario instalarlo ni registrarlo.

El proveedor `SunJCE` soporta algoritmos para distintas funcionalidades criptográficas, entre otras:

- **Cifrado simétrico y asimétrico**.
- **Distribución de claves.**
- **Generación de claves.**
- **MAC y hash:** HMAC-MD5, HMAC-SHA1
- **Firma Digital.**
- **Generacion de Certificados Digitales.**

Los ejemplos que están aquí usan `SunJCE`.

El proveedor Bouncy Castle se obtiene del sitio del proyecto www.bouncycastle.org

3. Paquetes

3.1. Package java.security[1]

Provee las clases e interfaces para el framework de seguridad.

Interfaces:

Entre otras:

- `AlgorithmConstraints`
- `Key`
- `KeyStore.Entry`
- `KeyStore.LoadStoreParameter`
- `KeyStore.ProtectionParameter`
- `Policy.Parameters`
- `PrivateKey`
- `PublicKey`

Clases:

Entre otras:

- `KeyFactory`
- `KeyPair`
- `KeyPairGenerator`
- `KeyStore`
- `MessageDigest`
- `SecureRandom`
- `Signature`
- `Timestamp`

3.2. Package javax.crypto[2]

Provee las clases e interfaces para operaciones criptográficas.

Interfaces:

Entre otras:

- `SecretKey`

Clases:

Entre otras:

- `Cipher`
- `EncryptedPrivateKeyInfo`
- `KeyAgreement`
- `KeyGenerator`
- `Mac`
- `SecretKeyFactory`

4. Hash y MAC.

Las clases (**engines)** que se usan son:

`MessageDigest` Para generar el hash de una secuencia de bytes.

`Mac` Para generar hash con clave, es decir Message Authentication Code.

En todos los casos de hash usaremos el mismo ejemplo: hash del texto: "***hace mucho calor hoy***".

[1] https://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html
[2] https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html

4.1. md5

```
import java.security.*;
public class ejemploHashMD5 {
	public static void main(String[] args) throws Exception{
		//Definimos texto plano al que se aplica el hash
		byte[] texto_plano = "hace mucho calor hoy".getBytes();
		//Generacion del hash con algoritmo MD5
		MessageDigest mdMD5 = MessageDigest.getInstance("MD5");
		mdMD5.update(texto_plano);
		byte[] resultado = mdMD5.digest();
		System.out.println("MD5: "+ hexStringFromBytes(resultado));
	}
	/*aquí definir hexStringFromBytes[3]…*/
}
```

4.2. sha1

```
import java.security.*;
public class ejemploHashSha1 {
	public static void main(String[] args) throws Exception{
		//Definimos texto plano al que se aplica el hash
		byte[] texto_plano = "hace mucho calor hoy".getBytes();
		//Generacion del hash con algoritmo MD5
		MessageDigest mdSha1 = MessageDigest.getInstance("SHA-1"[4]);
		mdSha1.update(texto_plano);
		byte[] resultado = mdSha1.digest();
		System.out.println("SHA-1: "+ hexStringFromBytes(resultado));
	}
	/*aquí definir hexStringFromBytes[5]…*/
}
```

[3] Función que transforma la secuencia de bytes en hexadecimal para facilitar la lectura.

```
private static String hexStringFromBytes(byte[] b)
{
  char[] hexChars={'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};
  String hex = "";
  int msb;
  int lsb = 0;
  int j;
  for (j = 0; j < b.length; j++)
  {
      msb = ((int)b[j] & 0x000000FF) / 16;
      lsb = ((int)b[j] & 0x000000FF) % 16;
      hex = hex + hexChars[msb] + hexChars[lsb];
  }
  return(hex);
}
```

[4] Al especificar los nombres de los algoritmos, no es obligado respetar las diferencias entre mayúsculasy minúsculas. Es decir, se toma como equivalentes: “SHA-1” “sha-1” o “Sha-1”

4.3. HMAC MD5

Aquí hay que tener en cuenta que primero debe generarse la clave, para luego efectuar el MAC.

Las clases (**engines)** que se usan son:

`KeyGenerator:` Para generar claves secretas para un algoritmo específico.

`SecretKey:` Clave criptográfica secreta.

`Mac` Para generar hash con clave, es decir Message Authentication Code.

```
import java.security.*;
import javax.crypto.*;//para poder generar y usar la clave
public class ejemploHmacMD5 {
      public static void main(String[] args) throws Exception {
            //Se genera una clave
            KeyGenerator kg = KeyGenerator.getInstance("HmacMD5");
            SecretKey sk = kg.generateKey();

            //Se efectuará MAC con algoritmo HMAC MD5
            Mac mac = Mac.getInstance("HmacMD5");
            //Se inicializa con la clave:
            mac.init(sk);
            //Se efectúa el HMAC
            byte[] texto_plano = "hace mucho calor hoy".getBytes();
            byte[] resultado = mac.doFinal(texto_plano);
            System.out.println("SHA-1: "+ hexStringFromBytes (resultado));
      }
}
```

5. Cifrado Simétrico

Las clases (**engines)** que se usan son:

`SecureRandom:` Para generar números seudoaleatorios.

`Cipher:` Luego de ser inicializada con la clave correspondiente, se usa para el cifrado/descifrado de la información.

`KeyFactory:` Para convertir claves criptográficas opacas tipo **`Key`** en especificaciones de clave y viceversa.

`SecretKeyFactory:` Para convertir claves criptográficas opacas tipo **`SecretKey`** en especificaciones de clave y viceversa. Sólo para claves simétricas.

`KeyGenerator:` Para generar claves secretas para un algoritmo específico.

El primer paso es generar una clave. Para ello, se genera una instancia de un generador de claves para el algoritmo en particular que se quiera usar (DES, AES, etc.).

Luego debe crearse una instancia del "cifrador" `Cipher`. Al hacerlo, se establece el nombre del algoritmo, el modo de cifrado y el modo de padding. Si el cifrado no se establece elige "AES", si el modo de cifrado no se establece, toma "ECB" y el padding estándar es "PKCS5Padding"

Finalmente se efectúa la encripción (o desencripción). Debe previamente inicializarse el "cifrador" con los parámetros ENCRYPT_MODE o DECRYPT_MODE según corresponda.

En el ejemplo, se hace encripción del texto "*Contenido de prueba.*", con DES, ECB, PKCS5 padding.

```
import javax.crypto.*;
public class ejemploCifradoDES {

	public static void main(String[] args) throws Exception{
	//Se genera la clave para DES
	KeyGenerator keygen = KeyGenerator.getInstance("DES");
	SecretKey desKey = keygen.generateKey();

	//Se genera instancia de Cipher
	Cipher desCipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
	//Se inicializa el cifrador para poder encriptar con la clave
	desCipher.init(Cipher.ENCRYPT_MODE,desKey);

	//Texto a encriptar.
	byte[] cleartext = "Contenido de prueba".getBytes();
	//Se encripta
	byte[] ciphertext = desCipher.doFinal(cleartext);
	System.out.println("El cifrado es:"+ new String(ciphertext, "UTF8"));
	System.out.println("Ahora descifra...");

	//Se desencripta
	desCipher.init(Cipher.DECRYPT_MODE,desKey);
	byte[] cleartext_out = desCipher.doFinal(ciphertext);
	System.out.println("El descifrado es:"+new String(cleartext_out, "UTF8"));
	}
}
```

6. Cifrado Asimétrico

Las clases (**engines)** que se usan son:

`SecureRandom:` Para generar números seudoaleatorios.

`Cipher:` Luego de ser inicializada con la clave correspondiente, se usa para el cifrado/descifrado de la información.

`KeyPairGenerator:` Para generar un par de claves (pública y privada) para un algoritmo específico.

`KeyAgreement:` Para acordar y establecer una clave específica que será usada por una operación criptográfica particular.

El primer paso es generar un par de claves (pública y privada). Para ello, se genera una instancia del generador **`KeyPairGenerator`**. Los métodos de inicialización de la clase KeyPairGenerator requieren como mínimo la longitud de la clave. Para generar una clave de 1024 bits de longitud se puede entonces establecer ese parámetro. Luego con el método `generateKeyPair` se termina de generar un objeto `KeyPair`.

Luego debe crearse una instancia del "cifrador" `Cipher`. Al inicializarlo, se establece si es ENCRYPT_MODE, con clave pública o bien DECRYPT_MODE con clave privada.

```
import java.security.*;
import javax.crypto.*;
public class ejemploCifradoRSA {
	public static void main(String[] args) throws Exception{
		//Se genera el par de claves
		KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
		keyGen.initialize(1024);

		KeyPair key = keyGen.generateKeyPair();
```

```
            Key pubKey = key.getPublic();
            Key privKey = key.getPrivate();

            //Se genera instancia de Cipher
            Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
            cipher.init(Cipher.ENCRYPT_MODE, pubKey);

            //Se encriptan los datos
            byte[] data = "Contenido de Prueba".getBytes();
            byte[] cipherText = cipher.doFinal(data);

            //Vemos el cifrado
            System.out.println("El cifrado es:"+ new String(cipherText,
"UTF8"));


            //Se desencriptan los datos
            cipher.init(Cipher.DECRYPT_MODE,privKey);
            byte[] descipherText = cipher.doFinal(cipherText);

            //Vemos el resultado
            System.out.println("El descifrado es:"+ new String(descipherText,
"UTF8"));
      }

   }
```

7. Firma Digital

Las clases (**engines)** que se usan son:

`SecureRandom:` Para generar números seudoaleatorios.

`Signature:` Para firmar digitalmente.

`KeyPairGenerator:` Para generar un par de claves (pública y privada) para un algoritmo específico.

`KeyAgreement:` Para acordar y establecer una clave específica que será usada por una operación criptográfica particular.

Como en cifrado asimétrico, se genera un par de claves (pública y privada).

Una vez obtenido el par de claves, se obtiene una instancia del objeto `Signature` . Por ej, `dsa`.

Con la clave privada, se inicializa la instancia de la firma dsa.

Se efectúa la firma de la información contenida en un array de bytes.

Para verificar la firma se usa la clave pública.

```
import java.security.*;
public class ejemploFirmaDigital {
      public static void main(String[] args) throws Exception{
            //Se genera el par de claves
            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
            SecureRandom random = SecureRandom.getInstance("SHA1PRNG","SUN");
            keyGen.initialize(1024, random);
            KeyPair pair = keyGen.generateKeyPair();

            //Se genera instancia de Firma
            Signature dsa = Signature.getInstance("SHA1withDSA");
            //Se obtiene la clave privada del par
            PrivateKey priv = pair.getPrivate();
```

```
            dsa.initSign(priv);

            //Se firman los datos
            byte[] data = "Estos datos hay que firmar".getBytes();
            dsa.update(data);
            byte[] sign = dsa.sign();

            //Vemos la firma
            System.out.println("La firma es:"+hexStringFromBytes(data));

            //Se obtiene la clave publica del par
            PublicKey pub = pair.getPublic();
            dsa.initVerify(pub);

            //Se verifican los datos
            dsa.update(data);
            boolean verifica = dsa.verify(sign);

            //Vemos el resultado de la firma
            if (verifica) System.out.println("Firma Validada");
            else System.out.println("Firma NO Validada");
      }
      /*aquí definir hexStringFromBytes…*/
}
```

8. Bibliografía y fuentes consultadas

- **Maiorano, Ariel.** *Criptografía. Técnicas de desarrollo para profesionales.* México, Alfaomega, 2009.
  El libro está en la biblioteca ITBA.
  Se pueden obtener los códigos fuente del libro de http://libroweb.alfaomega.com.mx/book/528
- **ORACLE.** *JCA Reference Guide*.

http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html