**ENCRIPTACION**

**MD5**

MD5 permite encriptar una frase, pero no permite desencriptarla. Es útil para guardar password con encriptación MD5 en una base de datos. Cuando un usuario introduce su clave, el código la encripta con MD5 y compara con la que hay en base de datos. De esta forma, la clave en claro no está almacenada en ningún sitio y tampoco es posible recuperarla a partir de la encriptada.

En java se usa la clase MessageDigest para obtener la encriptación MD5. El siguiente ejemplo muestra cómo:

package com.chuidiang.ejemplos.encrypt;

import java.security.MessageDigest;

import org.apache.commons.codec.binary.Base64;

public class MD5Example {

public static void main(String[] args) throws Exception {

MessageDigest md = MessageDigest.getInstance(MessageDigestAlgorithms.MD5);

md.update("texto a cifrar".getBytes());

byte[] digest = md.digest();

// Se escribe byte a byte en hexadecimal

for (byte b : digest) {

System.out.print(Integer.toHexString(0xFF & b));

}

System.out.println();

// Se escribe codificado base 64. Se necesita la librería

// commons-codec-x.x.x.jar de Apache

byte[] encoded = Base64.encodeBase64(digest);

System.out.println(new String(encoded));

}

}

Se obtiene un MessageDigest con MessageDigest.getInstance() indicando que algortimo de encriptación queremos (el parámetro MessageDigestAlgorithms.MD5).

Convertimos a bytes nuestro texto, en este caso "texto a cifrar".getBytes(), y se lo pasamos a MessageDigest con el método update().

Ahora basta con pedirle el texto encriptado con el método digest(). Nos lo devolverá como un array de bytes.

Como los bytes obtenidos pueden no ser legibles si los ponemos como caracteres en la pantalla, detrás de este código simplemente hemos puesto dos formas de sacar la clave encriptada por pantalla de una forma legible:

Como números hexadecimales, uno detrás de otro, que daría algo como 2fea7da4a3b790e7b3c7efd639fdc4

Codificado base 64, con la ayuda de la librería Apache commons-codec-1.8.jar en nuestro ejemplo, que daría algo como Av4KfaSjt5Dns8fv1jn9xA==

Este tipo de transformaciones para visualización evitan tambien, en caso de almacenarlos en fichero o base de datos, los problemas típicos de codificación de caracteres.

**SHA**

SHA es similar a MD5, aunque más moderno y más seguro. El código java sería exactamente igual, pero reemplezando MessageDigestAlgorithms.MD5 por MessageDigestAlgorithms.SHA\_1, MessageDigestAlgorithms.SHA\_256, MessageDigestAlgorithms.SHA\_384 o MessageDigestAlgorithms.SHA\_512.

SHA suele usarse para garantizar que un texto no ha sido modificado. Se obtiene el encriptado SHA del texto y se envía junto con el texto. El receptor, vuelve a calcular el SHA del texto y verifica que coincide con el SHA que ha recibido de origen. Para evitar que alguien mal intencionado modifique el texto y genere el nuevo SHA, el SHA original suele encriptarse con RSA o algún otro mecanismo que garantice quién lo ha generado.

**RSA**

RSA es un algoritmo para encriptado y desencriptado con un par de claves, pública y privada. Lo que se encripta con la clave pública se puede desencriptar con la clave privada y viceversa. Si el que envía el mensaje tiene una de las claves y el que lo recibe tiene la otra, este sistema garantiza:

El que envía el texto encriptado tiene la garantía de que sólo el receptor puede desencriptarlo, ya que es el único que tiene la clave pareja de la suya.

El receptor que desencripta el mensaje tiene la garantía de que ha sido generado por el emisor, ya que es el único que tiene la clave pareja de la suya.

No es un algoritmo eficiente para encriptar textos largos. Se usa para encriptar claves cortas y poder pasarlas de uno a otro de una forma segura, o para encriptar firmas digitales (texto encriptado SHA o MD5. también cortas).

El código java para este tipo de encriptado puede ser el siguiente

package com.chuidiang.ejemplos.encrypt;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.security.Key;

import java.security.KeyFactory;

import java.security.KeyPair;

import java.security.KeyPairGenerator;

import java.security.PrivateKey;

import java.security.PublicKey;

import java.security.spec.KeySpec;

import java.security.spec.PKCS8EncodedKeySpec;

import java.security.spec.X509EncodedKeySpec;

import javax.crypto.Cipher;

public class RSAAsymetricCrypto {

private static Cipher rsa;

public static void main(String[] args) throws Exception {

// Generar el par de claves

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");

KeyPair keyPair = keyPairGenerator.generateKeyPair();

PublicKey publicKey = keyPair.getPublic();

PrivateKey privateKey = keyPair.getPrivate();

// Se salva y recupera de fichero la clave publica

saveKey(publicKey, "publickey.dat");

publicKey = loadPublicKey("publickey.dat");

// Se salva y recupera de fichero la clave privada

saveKey(privateKey, "privatekey.dat");

privateKey = loadPrivateKey("privatekey.dat");

// Obtener la clase para encriptar/desencriptar

rsa = Cipher.getInstance("RSA/ECB/PKCS1Padding");

// Texto a encriptar

String text = "Text to encrypt";

// Se encripta

rsa.init(Cipher.ENCRYPT\_MODE, publicKey);

byte[] encriptado = rsa.doFinal(text.getBytes());

// Escribimos el encriptado para verlo, con caracteres visibles

for (byte b : encriptado) {

System.out.print(Integer.toHexString(0xFF & b));

}

System.out.println();

// Se desencripta

rsa.init(Cipher.DECRYPT\_MODE, privateKey);

byte[] bytesDesencriptados = rsa.doFinal(encriptado);

String textoDesencripado = new String(bytesDesencriptados);

// Se escribe el texto desencriptado

System.out.println(textoDesencripado);

}

private static PublicKey loadPublicKey(String fileName) throws Exception {

FileInputStream fis = new FileInputStream(fileName);

int numBtyes = fis.available();

byte[] bytes = new byte[numBtyes];

fis.read(bytes);

fis.close();

KeyFactory keyFactory = KeyFactory.getInstance("RSA");

KeySpec keySpec = new X509EncodedKeySpec(bytes);

PublicKey keyFromBytes = keyFactory.generatePublic(keySpec);

return keyFromBytes;

}

private static PrivateKey loadPrivateKey(String fileName) throws Exception {

FileInputStream fis = new FileInputStream(fileName);

int numBtyes = fis.available();

byte[] bytes = new byte[numBtyes];

fis.read(bytes);

fis.close();

KeyFactory keyFactory = KeyFactory.getInstance("RSA");

KeySpec keySpec = new PKCS8EncodedKeySpec(bytes);

PrivateKey keyFromBytes = keyFactory.generatePrivate(keySpec);

return keyFromBytes;

}

private static void saveKey(Key key, String fileName) throws Exception {

byte[] publicKeyBytes = key.getEncoded();

FileOutputStream fos = new FileOutputStream(fileName);

fos.write(publicKeyBytes);

fos.close();

}

}

Vamos con una explicación.

Obtenemos una instancia de KeyPairGenerator con KeyPairGenerator.getInstance("RSA") y a partir de ella obtenemos una clave pública y una clave privada con keyPairGenerator.generateKeyPair()

Guardamos las claves en fichero, simplemente para ver cómo se hace. Cada clave tiene un método key.getEncoded() que nos devuelve la clave en forma de array de bytes. Nos basta con guardar este array de bytes en fichero, como hace el método saveKey() de nuestro código.

La forma de recuperar la clave es distinta según esta clave sea pública o privada, por ello la existencia de dos métodos loadPrivateKey() y loadPublicKey(). La diferencia es que se debe usar una clase hija de EncodedKeySpec para obtener la clave a partir de los bytes leídos en fichero y no vale la misma clase hija. Hay varias posibilidades, pero para clave pública podemos usar X509EncodedKeySpec y para la clave privada podemos usar PKCS8EncodedKeySpec, ambas hijas de EncodedKeySpec. Una vez que tenemos estas instancias y la de keyFactory, podemos usar el método keyFactory.generatePrivate(keySpec) o keyFactory.generatePublic(keySpec) para recuperar las clave privada o pública, respectivamente.

Para encriptar nuestro texto, obtenemos un cifrador rsa con Cipher.getInstance("RSA/ECB/PKCS1Padding"). A este cifrador se le inicializa con el texto y la clave a usar para cifrar, con rsa.init(Cipher.ENCRYPT\_MODE, publicKey), es decir, le indicamos que se ponga en modo de encriptar y que use la clave que le pasamos de parámetro. Con rsa.doFinal(text.getBytes()) le pasamos el texto a encriptar como array de bytes y nos devuelve el array de bytes encriptado.

Si queremos ahora desencriptar, volvemos a inicializar nuestro cifrador rsa con rsa.init(Cipher.DECRYPT\_MODE, privateKey), es decir, le decimos que se ponga en modo descifrar y la clave que tiene que usar. Como encriptamos con la clave pública, necesitamos la clave privada para desencriptar.

Simplemente como añadido al comentario de que RSA no es eficiente para encriptado y desencriptado, hemos sacado por pantalla los bytes del texto encriptado. Un texto tan pequeño saca la siguiente friolera de bytes

1562de02a18c260b8315c5cad57b91e7ef194c97bb8d0abb68e60cedf4d9cefcf8977d5d822f14d094ec4ccd81df2d4e

329e70f06633d66fd5f5c204b22b0ec1587be8bd62a46fa53808387119748296fbe2fa1f89d3c8a599a5ab9df511ffa

c27cfb6f2b1ca072d237d4ba991b6fae795441c27c6b0d48881953f

así que puedes imaginar lo que daría un texto largo.

La criptografía simétrica solo utiliza una clave para cifrar y descifrar el mensaje, que tiene que conocer el emisor y el receptor previamente y este es el punto débil del sistema, la comunicación de las claves entre ambos sujetos, ya que resulta más fácil interceptar una clave que se ha transmitido sin seguridad (diciéndola en alto, mandándola por correo electrónico u ordinario o haciendo una llamada telefónica).

Teóricamente debería de ser más fácil conocer la clave interceptándola que probándola una por una por fuerza bruta, teniendo en cuenta que la seguridad de un mensaje cifrado debe recaer sobre la clave y nunca sobre el algoritmo (por lo que sería una tarea eterna reventar la clave, como comenté en un ejemplo de ataque por fuerza bruta).

Para poner un ejemplo la máquina Enigma (que era una maquina de cifrado electromecánica que generaba abecedarios según la posición de unos rodillos que podrían tener distintas ordenes y posiciones) usaba un método simétrico con un algoritmo que dependía de una clave (que más que clave parece un ritual) que está formada por: los rotores o rodillos que usaba, su orden y la posición de cada anillo, siendo esto lo más básico.

La máquina Enigma contaba también con un libro de claves que contenía la clave del día y hacia un poco más difícil encontrar la clave, pero no es una clave lo suficientemente segura como para que no se pudiese reventar, sobretodo cuando los ingleses gracias a los polacos consiguieron el algoritmo, por este motivo la mayoría de los días conseguían la clave.

Y otro inconveniente que tiene este sistema es que si quieres tener un contenido totalmente confidencial con 10 personas tienes que aprenderte o apuntarte (siendo esta forma menos segura) las 10 claves para cada persona.

**Criptografía asimétrica**

La criptografía asimétrica se basa en el uso de dos claves: la pública (que se podrá difundir sin ningún problema a todas las personas que necesiten mandarte algo cifrado) y la privada (que no debe de ser revelada nunca).

Sabiendo lo anterior, si queremos que tres compañeros de trabajo nos manden un archivo cifrado debemos de mandarle nuestra clave pública (que está vinculada a la privada) y nos podrán mandar de forma confidencial ese archivo que solo nosotros podremos descifrar con la clave privada.

Puede parecer a simple vista un sistema un poco cojo ya que podríamos pensar que sabiendo la clave pública podríamos deducir la privada, pero este tipo de sistemas criptográficos usa algoritmos bastante complejos que generan a partir de la frase de paso (la contraseña) la clave privada y pública que pueden tener perfectamente un tamaño de 2048bits (probablemente imposible de reventar).

Como os habréis dado cuenta solo cifra una persona (con la clave pública) y la otra se limita a mirar el contenido, por lo que la forma correcta de tener una comunicación bidireccional sería realizando este mismo proceso con dos pares de claves, o una por cada comunicador.

Otro propósito de este sistema es también el de poder firmar documentos, certificando que el emisor es quien dice ser, firmando con la clave privada y verificando la identidad con la pública.

**Diferencias entre criptografía simétrica y asimétrica**

Para empezar, la criptografía simétrica es más insegura ya que el hecho de pasar la clave es una gran vulnerabilidad, pero se puede cifrar y descifrar en menor tiempo del que tarda la criptografía asimétrica, que es el principal inconveniente y es la razón por la que existe la criptografía híbrida.