```
      new InputStreamReader(System.in)).readLine();
      nameCallback.setName(username);
    } else if (cb instanceof PasswordCallback) {
    PasswordCallback passwordCallback = (PasswordCallback)cb;
      System.out.print( passwordCallback.getPrompt() + "? ");
      System.out.flush();
      String password = new BufferedReader(
       new InputStreamReader(System.in)).readLine();
      passwordCallback.setPassword(password.toCharArray());
      password = null;
    } else {  throw new UnsupportedCallbackException(cb, "Unsupported Callback
Type");  }
   }
  }
}
```

Puede encontrar todos los archivos necesarios en el archivo ejemplos.jar. Para instalar los ejemplos copie ejemplos.jar en el directorio que desee y ejecute desde la línea de comandos jar xvf ejemplos.jar. Algunos ejemplos ya están compilados podrá ver como funcionan ejecutándolos escribiendo run

capítulo 4

Encriptación y desencriptación

Introducción

En este tutorial, se intentara que el lector consiga una base sólida y duradera sobre los principios de seguida. Por lo tanto aparte de instruir en el uso de una API concreta, también intentamos transmitir un conocimiento mas profundo. Para ello no basta con decir esto se hace de tal forma, sino que en ocasiones requiere que nos adentremos en los aspectos matemáticos o algorítmicos de los procesos. Este es el caso del capítulo 4. Primero daremos una visión independiente sobre la criptografía, y después pasaremos a implementar algunos ejemplos con Java y JCE. Si el lector no esta interesado en comprender los fundamentos de la criptografía y prefiere hacer uso directamente de las

herramientas y APIs puede saltarse la siguiente sección.

criptografía

Terminología y notación

Un poco de historia ...

Las bases fundamentales de la criptografía, fueron establecidos por

- Claude Shannon : En un articulo titulado "A mathematical theory of communication " donde introduce las bases de la teoria de la información.
- Whitfield Diflie actualmente ingeniero de Sun Microsystem, fue el descubridor en 1975 del concepto de criptografía de clave publica.

las teoría de la criptografía se basa en las siguientes teorías matemáticas (que no serán analizadas en este tutorial):

- teoría de la información
- teoría de números
- teoría de la complejidad algorítmica
- teoría de probabilidad estadística

En la bibliografía básica se suele hacer referencia a los distintos roles que puede tener un usuario, haciendo uso de algURL de personas, asi que tenemos los siguientes personajes con los que iremos familiarizando durante esta sección. Reparto :

- Alice y Bob
- Carol (tercera persona en protocolos de mas de dos usuarios)
- Dave (cuarta persona en protocolos de mas de 3 usuarios)
- Mallory "la mala" (atacante maliciosa)
- Trent (Arbitro de confianza)
- Peggy (demostrador)
- Victor (la persona que comprueba)

Estos nombres son estándares de hecho.

Conceptos básicos y tipos de funciones según sus claves

Cuando hablamos de encriptación debemos tener algunos conceptos claros.

- **Mensaje :** Son los datos que actualmente con conciernen. Normalmente será un texto plano. Denotado por M
- **Mensaje cifrado :** Es el mensaje M encriptado. LLamamos al mensaje cifrado C
- **Espacio de mensajes** : Son todos los mensajes posibles, en la mayoria de los casos suele ser infinito, pero en otros la longitud puede estar limitada.
- **Espacio de mensajes cifrado** : Lo mismo que el punto anterior pero para mensajes cifrados.
- **Espacio de claves** Son el conjunto de todas las claves posibles. Se representa por la letra K (mayúscula, la k minúscula representa una clave en concreta)

Matemáticamente, la encriptación no es mas que una aplicación desde el dominio de M al rango de C. Y la desencriptación es la función inversa.

- Expresamos la encriptación como C = E (M) .
- La desencriptacion como M = D(C)
- M= D(E(M)).

Un ejemplo de lo anterior : Un mensaje M = "hola", una función de encriptacion escribir la siguiente letra del alfabeto, Y una función de desencriptado escribir la letra anterior del alfabeto nos quedaria. M ="hola", C= D(M) ="ipob", M = D(E(M)) = "hola".

Es importante notar la diferencia entre un protocolo y un algoritmo en términos de criptografía.

- Un algoritmo es el conjunto de pasos necesarios para transformar un dato en un otro dato.
- Un protocolo es el conjunto de todos los pasos entorno al transporte de la actividad criptográfica. Los protocolos usan necesariamente uno o mas algoritmos criptográficos.

Un algoritmo de encriptación poderoso , no garantiza la confidencialidad del mensaje. Por ejemplo supongamos un algoritmo de cifrado irrompible, y un débil protocolo, si una persona intercepta el mensaje y rompe el débil protocolo, este tendrá acceso a la clave para desencriptar el mensaje, ya que la clave no puede estar encriptada, lo cual hace al algoritmo de encriptación irrompible inútil

Criptosistemas de clave privada (algoritmos simétricos).

Normalmente no se suele usar un método tan sencillo de cifrado, ya que este es lineal y solo depende de un factor (Una entrada siempre produce la misma salida) . Un paso mas allá es usar una clave para realizar el cifrado. denotándolo asi

- Expresamos la encriptación como C = E {k}(M) .
- La desencriptacion como M = D{k}(C)
- M= D{k}(E{k}(M)).
- Pero no es lo mismo cifrar y descifrar con dos claves distintas. Entonces M != D{k1}(E{k2}(M), donde k1 != k2 (mas adelante veremos que esto si se puede cumplir si el cifrado es asimétrico).

Estos son los conceptos matemáticos básico para entender la criptografía.

Un ejemplo moderno de este tipo de sistema es el algoritmo DES

Criptosistemas de clave publica (algoritmos asimétricos).

Podemos dividir el tipo de encriptaciones según usen o no la misma contraseña, para encriptar y desencriptar.

- Decimos que la encriptación es simétrica que para desencriptar, M= D{k}(E{k}(M)) .
- Si usamos dos claves distintas entonces no hallamos ante encriptación asimétrica. M != D{k1}(E{k2}(M). A la llave que usamos para encriptar (k1) la llamamos "clave publica" y la clave que se usa para el descifrado (k2) es la "clave privada"

Computacionalmente no debe ser posible transformar una clave publica en privada, veamos por que :

1 Sea k-privada la clave privada

2 Sea k-publica la clave publica

3 Sea X ( ) una función computacionalmente factible que transforme cualquier clave publica en una privada.

4 Sea D(k-privada) {} la función de desencriptación

5 Por definición, M= D{k-privada}(E{k-publica}(M))

6 X(k-publica) = k-privada, entonces ---> D'{k-publica} ( ) = D{X(k-publica)}() = D{k-priv}( ) , esto quiere decir que obtenemos una función de desencriptación a partir de la clave publica. Lo que nos lleva a

7 **M = D'{k-pub}(E{k-pub}(M))**. Por lo cual hemos lo hemos reducido a una encriptación simétrica !!!!!

El uso de las claves, esto nos permite hacer publicas la clave para cifrar, con la que los usuarios podran cifrar, pero solo aquellos que posean la clave privada podran descifrar el mensaje. Por lo cual podemos recibir mensajes secretos sin tener que compartir la clave.

Clave publica vs clave privada

Las posibilidades de los algoritmos de clave publica nos permiten varias opciones que no permite los algoritmos de clase privada :

- Servicios de autentificación
- firma digital
- Publicar una clave de cifrado, que puede ser distribuida sin riesgos.

Pero los algoritmos de clave privada también tienen sus ventajas.

- Los algoritmos simétricos son mas rápidos que los de clave publica, por lo que en mensajes largos es mejor usar uno de clave privada.

Otros tipos de Funciones

Existen algunos tipos de funciones que siendo muy importantes para algunos algoritmos y protocolos , no son funciones de encriptación en si mismas. Las funciones de un solo sentido son aquellas que tienen una solución sencilla y de bajo coste computacional, pero que requieren un gran esfuerzo realizar la función inversa.

Otro tipo de "función", son las hash. **Las funciones hash no sirven para asegurar la confidencialidad, sino para asegurar la integridad del mensaje**, es decir para saber que el contenido no ha sido alterado por un atacante.

La función de resumen hash toma un mensaje de x-bits de longitud variable, lo transforma en otro de h bits de longitud fija (h =H(x), este tipo de funciones deben cumplir las siguientes condiciones, si se desean usar para propósitos criptográficos.

- Como la entrada es de longitud variable, por lo que necesitara tener mecanismos que detecte y evite el desbordamiento.
- devolverá una cadena de longitud fija h
- para cualquier entrada, el "resumen" debe ser sencillo de calcular.
- la función debe ser de un solo sentido
- **colisión** debe de ser lo mas difícil posible encontrar valores de resumen repetidos , es decir que dado un x e y se minimice los casos de H(x) = H(y)

podemos hacer dos distinciones en las funciones hash , se puede demostrar

- Débilmente libre de colisiones , cuando existe alguna colisión
- Fuertemente libre de colisiones

Llamamos huella al mensaje resumido, El caso ideal seria aquel que cambiando un bit de la huella del mensaje, supusiere una alteración del 50 por ciento de los bits del mensaje original.

Que hace seguro un protocolo?

Veamos tres definiciones para poder afrontar este tema

- Clave
- Password palabra que es conocida por el usuario. que le sirve para autentificarse y obtener permisos. Es débil ante los ataques, por ejemplo los sistemas que usan password de 64-bits, lo cual representa palabras de 8 caracteres. Por desgracia muchas implementaciones usan estos 8 caracteres directamente como clave. La fuerza de estos algoritmos reside en suponer que el atacante ignora 2^64 posibilidades se ha escogido. Otra debilidad es que los password no suelen ser escogido al azar. Lo que provoca que la probabilidad de que aparezcan ciertas letras, y las posiciones que estas ocupen siguen un patrón calculable. Por ejemplo en un usuario español no suele usar la letra w y mucho menos dos w seguidas. Por lo cual es útil usar una función para crear una nueva clave a partir de la entrada del usuario.
- Passphrase normalmente tiene el mismo uso que el password, diferenciándose de este principalmente en la longitud, entre 20 y 100 caracteres.

La fuerza bruta es uno de los métodos usados para romper los algoritmos. Este método se basa en probar todas las claves posibles hasta dar con la que descifra el mensaje. En algunos casos se puede reducir el espacio de claves, a través de un análisis del mensaje, pero esto tampoco suele aportar grandes beneficios. El atacante debe saber cuando ha obtenido el mensaje correcto, aunque esto no suele ser difícil. Cuanto mayor sea el numero de bits de la clave, mayor será el espacio de claves, esto consigue que el tiempo que tarde en romperse el algoritmo por fuerza bruta se incremente de forma exponencial.

El diccionario español tiene unas 100000 entradas, con la capacidad de computo actual solo se tardaría unos segundos en comprobarlas todas, como ya comentamos las personas no suelen memorizar claves al azar, sino que usan palabras con un significado especial.

análisis criptográfico.

En este punto también debemos entender una serie de conceptos básicos, estas características son heredadas de la estructura de texto plana.

- **Entropía :** La entropía es el exceso de información subyacente que se encuentra dentro del mensaje cifrado. Un ejemplo es la emisión codificada de Canal +, el mensaje (la imagen normal) se encuentra dentro de la imagen que vemos codificada, pero el exceso de información hace imposible distinguir el sonido o la imagen, ya que esta información excesiva corrompe el formato.
- **cadencia del lenguaje :** La cantidad de información subyacente que añade cada

caracter añadido a un mensaje. por ejemplo el español añade. Podría pensar se que la cantidad de información solo depende de cuantos caracteres posee el código ASCII, pero eso es falso, ya que también depende de la distribución de probabilidad que tiene cada letra. Por ejemplo una letra E (la mas común en el español) y la letra W no tienen la misma probabilidad de aparecer en un mensaje. El problema se agudiza por el hecho de que las parejas de letras todavía son mas comunes, por ejemplo en español nunca ocurre la pareja wq.

Los tipos de ataque posibles contra un algoritmo, depende sobre todo que posea el atacante

- Solo con el texto cifrado : En este caso normalente el atacante intenta deducir cual será el texto original. Normalmente usa un ataque por fuerza bruta contra el espacio de claves.
- Mensaje y mensaje cifrado : En este caso es mucho mas sencillo hallar las claves y asi poder descifrar otros mensajes.

Los algoritmos de cifrado tienen dos modos de ir realizando estas operación :

- **Cifrado del flujo :** Este cifrado coge el mensaje original y va cifrando bit a bit (o byte a byte), asi el el bit(byte) cifrado depende solo del bit (byte) original y de la clave.
- **Cifrado por bloques :** El algoritmo coge un bloque del mensaje original (normalmente mas de 64 bits), el mismo bloque de texto y la misma clave creara el mismo bloque cifrado

Introducción a la criptografía con Java

Introducción

Desde la aparición de JDK 1.4 , java nos ofrece la posibilidad de trabajar con un framework para criptografía incluido en el núcleo de la JVM. El framework JCE (Java™ Cryptography Extension) nos ofrece las siguientes características

- Soporte para cifrado simétrico (DES, RC2, y IDEA)
- Soporte para cifrado simétrico (RSA)
- Cifrado por bloques o por flujo
- Algoritmos MAC (Message Authentication Code)
- Funciones de resumen (funciones hash como MD5 y SHA1)

- Generación de claves
- encriptación Basada en Password (PBE), transforma un password en una clave robusta mediante procesos aleatorios
- Acuerdo de claves : es un protocolo, para transmisiones entre dos o mas partes, mediante el cual se pueden establecer una acuerdo sobre las claves de cifrado sin intercambiar información secreta

JCE fue un framework opcional hasta la versión 1.4 y apareció por primera vez con la llegada de Java 2

Las clases básicas de JCE son

- Cipher
- Las clases Cipher Stream (CipherInputStream, CipherOutputStream)
- KeyGenerator
- SecretKeyFactory
- KeyAgreement
- Mac

Distribuidas en los siguientes paquetes

- javax.crypto
- javax.crypto.interfaces

javax.crypto.spec

Integridad de los mensajes

Veamos como podemos defender nuestros mensajes, mediante el uso de funciones de resumen (hash), para ello estudiaremos las diferentes especificaciones que no ofrece JDK 1.4 en JCE**.**

Anteriormente ya vimos en que consistian y que condiciones debían tener las funciones de resumen para ser útiles para su uso en programas de cifrado. Ahora veremos el uso de funciones estándar como MD5.

JDK1.4 nos ofrece los siguientes algoritmos de resumen

- **MD2** y **MD5** (que son algoritmos de 128-bits)
- **SHA-1** (160-bit)

- **SHA-256, SHA-383,** y **SHA-512**, (256, 383, y 512 bits, respectivamente)

Los mas común es usar MD5 o SHA1

La clase MessageDigest maneja el resumen de los mensajes. Esta clase tiene un constructor protegido , por lo que debemos acceder a ella mediante un método llamado getInstance (String algotithm), donde el parámetro representa el algURL del algoritmo con el que deseamos resumir.

Veamos un ejemplo sencillo de una clase que crea un resumen cifrado con una clave

```
[1]  public static byte[] getKeyedDigest(byte[] buffer, byte[] key) {
        try {
[2]            MessageDigest md5 = MessageDigest.getInstance("MD5");
[3]            md5.update(buffer);
[4]           return md5.digest(key);
        } catch (NoSuchAlgorithmException e) {
        }
[5]       return null;
     }
```

Analicémoslo paso a paso

[1] que le es pasada como parámetro, el otro parámetro es un array de bytes (podría ser sustituido sin problemas por un String).

[2] Creamos una instancia del objeto MessageDigest.

[3] El método update( ) nos sirve para indicarle al objeto que realizara el resumen cual será el String o array de bytes a resumir.

[4] Y el método digest(Secret key) se encarga de hacer el resumen cifrado con la clave.

[5] devuelve un null por que se produjo un error.

El siguiente ejemplo muestra por consola, el resumen del mensaje "hola mundo soy un texto plano" realizado por SHA y por MD5 asi como los datos del Provider, como vemos aquí el resumen no se realiza cifrado con una clave.

```
public class MessageDigestExample {
  public static void main (String[] args) throws Exception {
    if (args.length !=1) { System.exit(1); }
    byte[] textoPlano = args[0].getBytes("UTF8");
    System.out.println( "\n" + messageDigest.getProvider().getInfo() );

    MessageDigest messageDigest = MessageDigest.getInstance("MD5");
```

```
    messageDigest.update( textoPlano);
    System.out.println( "\nresumen: " );
    System.out.println( new String( messageDigest.digest(), "UTF8") );

    System.out.println("\n ------------------------------------\n");

    MessageDigest md2= MessageDigest.getInstance("SHA");
    messageDigest2.update( textoPlano);
    System.out.println( "\nresumen:" );
    System.out.println( new String( messageDigest2.digest(), "UTF8") );
  }
}
```

Veamos mas a fondo este ejemplo. primero nos fijamos en que creamos un String con el parámetro UFT8, esto es por que algunos caracteres del resumen pueden dar problemas al salir por pantalla.

1 messageDigest.getProvider().getInfo() nos da la información del Provider

2 Con getInstance obtenemos dos instancias una del algoritmo MD5 y otra SHA, con las que podremos hace el resumen del mensaje.

3 con update, introducimos el mensaje que queremos resumir

4 digest() realizamos la acción de resumen.

Esta es la salida que me aparece a mi, los datos que pueden cambiar son los del Provider.

```
C:\MisPruebas\Seguridad\JCE>java MessageDigestExample "Hola mundo soy un texto plano"
SUN (DSA key/parameter generation; DSA signing; SHA-1, MD5 digests; SecureRandom; X.509 certificates; JKS keystore; PKIX CertPathValidator; PKIX CertPathBuilder; LDAP, Collection CertStores)
resumen:
?Z?5h?n?m*+Eh_?
 ------------------------------------
resumen:
???,4?? ?w?aK$7??tb?
```

Vemos que las salidas son diferentes tanto en contenido como en longitud, 128 bits que

codifican 16 caracteres para el MD5 y 160 bits que generan los 20 caracteres, si necesitásemos un resumen mayor (por ejemplo para evitar una colisión, usaríamos SHA 256 o un SHA de mayor tamaño)

Para generar claves adecuadas podemos usar

Veamos un ejemplo

```
public class MessageAuthenticationCodeExample {
  public static void main (String[] args) throws Exception {
    if (args.length !=1) { System.exit(1);  }
    byte[] textoPlano = args[0].getBytes("UTF8");

    System.out.println( "\n Generando la clave....................." );
    KeyGenerator keyGen = KeyGenerator.getInstance("HmacMD5");
    SecretKey MD5key = keyGen.generateKey();
    System.out.print( "Clave generada" );

    Mac mac = Mac.getInstance("HmacMD5");
    mac.init(MD5key);
    mac.update(textoPlano);

    System.out.println( "\nMAC: " );
    System.out.println( new String( mac.doFinal(), "UTF8") );

    System.out.println( "\n-------------------------------\n" );

    System.out.println( "\n Generando la clave....................." );
    KeyGenerator keyGen2 = KeyGenerator.getInstance("HmacSHA1");
    SecretKey SHA1key = keyGen.generateKey();
    System.out.print( "clave generada" );

    Mac mac2 = Mac.getInstance("HmacSHA1");
    mac2.init(SHA1key );
```

```
    mac2.update(textoPlano);

    System.out.println( "\nMAC: " );
    System.out.println( new String( mac2.doFinal(), "UTF8") );
  }
}
```

Este ejemplo tarda mas de un minuto en ejecutarse, esto se debe a que se generan numero seudo-aleatorios a través de capturas del reloj interno La salida de consola esta vez nos da el siguiente resultado.

```
C:\MisPruebas\Seguridad\JCE>java MessageAuthenticationCodeExample "Probando el generador"
Generando la clave.................Clave generada
MAC:
*???  @Z¶ ? n?2
-------------------------------
Generando la clave.................Clave generada
MAC:
 ??? ? ??H??o?????
```

En el ejemplo hemos usado tres clases hasta ahora desconocidas veámoslas a fondo

La clase KeyGenerator

La clase KeyGenerator es usada para generar claves secretas para usarse en algoritmos simétricos.

Como mucha de las clases de esta API, KeyGenerator se instancia a través de una factoría (el patrón factory no es tratado en este tutorial), a través del método getInstance

- public static KeyGenerator getInstance(String algorithm) . especifica el algoritmo de cifrado
- public static KeyGenerator getInstance(String algorithm, String provider) especifica el proveedor y el algoritmo de cifrado.

Los algoritmos para los que generar claves esta disponible son

- AES
- Blowfish

- DES
- DESede
- HmacMD5
- HmacSHA1

Todos los algoritmos comparten el concepto de aleatorio y de tamaño de la clave par especificar estos atributos tenemos los método init. los tres primeros casos son útiles para todos, y los dos siguientes se usan para configurar parámetros específicos del algún algoritmo.

- public void init(SecureRandom random);
- public void init(int keysize);
- public void init(int keysize, SecureRandom random);
- public void init(AlgorithmParameterSpec params);
- public void init(AlgorithmParameterSpec params, SecureRandom random);

para crear la clave debemos crear un objeto de la clase SecretKey (para almacenar la clave) y usar el método generateKey( ); Por ejemplo

```
SecretKey clave = mikeyGenerator.generateKey();
```

La clase Mac

La clase Mac, nos proporciona las funcionalidades para autentificar los mensajes.

En la clase Mac, también se crea a través de una factoría.

- public static Mac getInstance(String algorithm);
- public static Mac getInstance(String algorithm, String provider);

Iniciamos la clase Mac con los métodos

- public void init(Key key);
- public void init(Key key, AlgorithmParameterSpec params);

Confidencialidad del mensaje con clave privada.

Introducción

Ahora que hemos asegurado el mensaje de posibles ataques, es importante que el atacante, no pueda ver su contenido. Veremos como cifrar con algoritmos de clave simétrica. JDK 1.4 soporta los siguientes algoritmos :

- **DES.** DES (Data Encryption Standard) desarrollado por IBM en los 70. Es un cifrado de bloque de 56-bit
- **TripleDES.** Consiste en aplicar el algoritmo DES tres veces (encriptar desencriptar encriptar) con dos claves dando un resultado de 112 bits.
- **AES.** Es el algoritmo que reemplazo a DES. Creado por Joan Daemen y Vincent Rijmen. Es un cifrado por bloque de 128-bit con claves de longitud 128, 192 o 256 bits.
- **RC2, RC4,** and **RC5.**
- **Blowfish.** Fue creado por Bruce Schneier y hace un cifrado por bloques con claves de longitud variable desde 32 a 448 bits (en múltiplos de 8), Diseñado para ser eficiente en computadores.
- **PBE.** PBE (Password Based Encryption) puede ser usado con algoritmos de clave privada y funciones de resumen.

Si tenemos una librería de otro proveedor que no sea Sun Microsystem, podemos tener algunas opciones mas, para comprobar que opciones podemos elegir usaremos

```
String[] algoritmos = getCryptoImpls("Cipher");
// para verlos por consola
for (int i =0; i<algoritmos .length ; i++)
   System.out.println(algoritmos[i]);
.....
```

Un ejemplo de salida seria

```
Blowfish
DESede
PBEWithMD5AndTripleDES
TripleDES
DES
PBEWithMD5AndDES
```

La clase Cipher

La clase Cipher, se usa para cifrar mediante algoritmos de clave simétrica. Como ya vimos es muy normal instanciar clases a través de una factoría. Este es otro ejemplo , aquí también usaremos getInstance( ) :

- public static Cipher getInstance(String transformation);
- public static Cipher getInstance(String transformation, String provider);

Una transformación tiene la forma :

- ***"algoritmo / modo / relleno "***
- ***"algoritmo"***

Ejemplos : Cipher c1 = Cipher.getInstance("*DES/ECB/PKCS5Padding*"); o Cipher c1 = Cipher.getInstance("*DES*");

Los modos son la forma de trabajar del algoritmo. Existen varios modos :

- Ninguno
- ECB (Electronic Code Book)
- CBC (Cipher Block Chaining)
- CFB (Cipher Feedback Mode)
- OFB (Output Feedback Mode)
- PCBC (Propagating Cipher Block Chaining)

Como hemos visto, un algoritmo puede cifrar por bloques de una longitud determinada, cuando el mensaje es un múltiplo de dicha longitud no existe ningún problema. Pero cuando el mensaje no es un múltiplo, el ultimo bloque es menor que la longitud necesaria para realizar el cifrado, entonces se realiza un relleno de ese bloque. A este relleno se le llama padding , Existen varias formas de rellenar los bloques de menor tamaño.

- No rellenar
- PKCS5
- OAEP
- SSL3

Para mas información sobre cada uno modos o padding, consulte la API

Esquema básico de cifrado con clave privada.

Veamos un esquema para guiarnos.

Primero crearemos la clave

- **KeyGenerator.getInstance("DES"),**
- **.init(56),**
- **.generateKey():**

Creamos un objeto Chiper y lo configuramos con los parámetros deseados

- **Cipher.getInstance("DES/ECB/PKCS5Padding"):**
- **.init(Cipher.ENCRYPT_MODE, clave):**

Realizamos el cifrado

- **.doFinal(textoPlano):**

Configuramos otra vez el objeto Cipher

- **.init(Cipher.DECRYPT_MODE, clave)**

Y desciframos

- **.doFinal(cipherText)**

Generar claves es muy sencillo, solo tenemos que indicar con que algoritmo deseamos que sea compatible nuestra clave el tamaño y llamar al método Observen este segmento de código:

```
try {
    //clave de tamaño de 56 bits para des
    KeyGenerator keyGen = KeyGenerator.getInstance("DES");
    keyGen.init(56);
    SecretKey clave1 = keyGen.generateKey();

    // Genera clave para  Blowfish
    keyGen = KeyGenerator.getInstance("Blowfish");
    SecretKey clave2 = keyGen.generateKey();

    // Genera clave para triple DES
    keyGen = KeyGenerator.getInstance("DESede");
    SecretKey clave3 = keyGen.generateKey();

 } catch (java.security.NoSuchAlgorithmException e) {  }
```

Veamos un ejemplo mas largo

```
public class PrivateExample {
  public static void main (String[] args) throws Exception {
    if (args.length !=1) { System.exit(1); }
    byte[] textoPlano = args[0].getBytes("UTF8");
    KeyGenerator keyGen = KeyGenerator.getInstance("DES");
    keyGen.init(56);
    Key clave = keyGen.generateKey();
```

```
    Cipher cifrar = Cipher.getInstance("DES/ECB/PKCS5Padding");

    cifrar.init(Cipher.ENCRYPT_MODE, clave);
    byte[] cipherText = cifrar.doFinal(textoPlano);
    System.out.println( new String(cipherText, "UTF8") );

    cifrar.init(Cipher.DECRYPT_MODE, clave);
    byte[] newPlainText = cifrar.doFinal(cipherText);
    System.out.println( new String(newPlainText, "UTF8") );
  }
}
```

Hemos visto como cifrar un mensaje, pero esto no es lo único que podemos. Si recordamos la interface Serializable, nos permitia crear un Stream de un Objeto, por lo tanto , nosotros podemos cifrar este Stream, veamos un ejemplo que eso siempre ayuda:

```
public class ClaseSecreta implements java.io.Serializable {
    String s = "Esto clase es secreta";
}
....
try {
    // Creamos una llave;
    SecretKey clave= KeyGenerator.getInstance("DES").generateKey();

    //  configuramos el cifrado
    Cipher cifrar = Cipher.getInstance("DES");
    cifrar .init(Cipher.ENCRYPT_MODE, llave);

    // encritamos
    SealedObject so = new SealedObject(new ClaseSecreta(), cifrar );

    // Obtenemos el algoritmo con el que se cifro nos devolverá DES.
    String nombreAlgo = so.getAlgorithm();

    // configuramos el descifrado,
    // he creado otro cipher por que normalmente no se usa el mimso q lo creo
    Cipher descifrar= Cipher.getInstance(nombreAlgo );
    descifrar.init(Cipher.DECRYPT_MODE, llave);
```

```
        // Desciframos
        ClaseSecreta o = (ClaseSecreta)so.getObject(descifrar);

    } catch (java.io.IOException e) {
    } catch (ClassNotFoundException e) {
    } catch (javax.crypto.IllegalBlockSizeException e) {
    } catch (javax.crypto.BadPaddingException e) {
    } catch (javax.crypto.NoSuchPaddingException e) {
    } catch (java.security.NoSuchAlgorithmException e) {
    } catch (java.security.InvalidKeyException e) {
    }
```

Bueno esto es muy bonito, pero .....A que te gustaría tener una clase que dada una String lo cifrase, pues vamos a por ella

```
public class DesEncriptar {
        String          Algoritmo;
        SecretKey  clave;

        DesEncriptar ( String Algoritmo, SecretKey clave)   {
            this.Algoritmo = Algoritmo;
            this.clave= clave;
        }

        DesEncriptar (String Algoritmo){
            this.Algoritmo = Algoritmo;
            this.clave= new KeyGenerator.getInstance("DES").generateKey();
         }


       public String encriptar(String str) {
              try {
                                //-----------    Pasamos a Bytes para hacer bien el cifrado
                              byte[] cadenaByte= str.getBytes("UTF8");

                  Cipher cifrar= Cipher.getInstance(Algoritmo);
                    cifrar.init(Cipher.ENCRYPT_MODE, clave);
                    byte[] enc = cifrar.doFinal(cadenaByte);         // ciframos

                              // ----------   Pasamos aBase64 para obtener un String
```

```
                return new sun.misc.BASE64Encoder().encode(enc);
            } catch (javax.crypto.BadPaddingException e) {
            } catch (IllegalBlockSizeException e) {
            } catch (UnsupportedEncodingException e) {
            } catch (java.io.IOException e) {
        }
        return null;

    }

    public String desencriptar(String str) {
        try {
                        // ------------   Obtenemos los Bytes del String encriptado
            byte[] dec = new sun.misc.BASE64Decoder().decodeBuffer(str);

            Cipher descifrar= Cipher.getInstance(Algoritmo);
            descifrar.init(Cipher.DECRYPT_MODE, clave);
            byte[] cadenaByte= dcipher.doFinal(dec);

                        // -------------  Volvemos a pasarlo a forma de cadena.
            return new String(cadenaByte, "UTF8");

        } catch (javax.crypto.BadPaddingException e) {
        } catch (IllegalBlockSizeException e) {
        } catch (UnsupportedEncodingException e) {
        } catch (java.io.IOException e) {
        }
        return null;
    }
}
```

La clase anterior esta diseñada para poder encriptar o desencriptar repetidamente, Imaginemos que Bob tiene una lista de nombres de personas invitadas por Alice a una fiesta. Veamos como tratariamos esa lista con nuestra clase, hemos supuesto que Bob conoce la clave secreta de Alice, que Alice cifro la lista con DES y que quiere leer el contenido de la lista.

```
// la variable listaNombres contiene los nombres cifrados
DesEncriptar desen = new DesEncriptar ("DES",claveAlice);
for (int i = 0; i<listaNombres.length; i++)
    listaNombres[i] = desen.desencriptar(listaNombres[i] );
```

```
// y ya esta !!!!
```

Veamos como Alice cifro la lista

```
// la variable listaNombres contiene los nombres sin cifrar
DesEncriptar desen = new DesEncriptar ("DES",claveAlice);
for (int i = 0; i<listaNombres.length; i++)
    listaNombres[i] = desen.encriptar(listaNombres[i] );
// y ya esta y solo se diferencia en tres letras!!!! .... ya solo quedaria enviarla
```

Obviamente esto tiene un problema, Alice y Bob tienen que conocer la clave

Confidencialidad del mensaje con clave publica.

La claves privadas tienen un problema, Supongamos que Alice quiere comunicarse con Bob, y no desea que Mallory pueda leer el mensaje. Por lo cual decide cifrar el mensaje, para ello crear una clave de cifrado, y aquí llega el problema , solo esa clave rompe el cifrado, por lo que Bob necesita conocer esa clave, por lo que tiene que enviarsela,de ese modo pone en peligro el cifrado, ay que la clave es infromación sensible. Si Mallory intercepta la clave, el cifrado será inútil.

La encriptación asimétrica elimina el problema de compartir información sensible. Ya que la clave de cifrado se puede compartir sin problemas, y la clave de descifrar solo la tiene que poseer un el receptor. Imaginemos que Alice y bob deben mantener una conversacion secreta que harian

1 Alice crea su clave privada y una clave publica de cifrado y se la manda a Bob sin tener que tomar precauciones

2 Bob crea su clave privada y su clave publica y se la manda a Alice sin o tomar precauciones.

3 Alice crea un mensaje y lo cifra con la clave publica de Bob y le manda el mensaje

4 Bob recibe el mensaje y lo descifra con su clave privada

5 Bob crea la respuesta la cifra con la clave publica de Alice y se lo manda a Bob y asi hasta que terminen .....

Podemos usar dos algoritmos para el cifrado asimetrico

- RSA**.** Es el algoritmo mas popular de cifrado asimetrico, pero no es soportado por JDK 1.4 por lo que debe instalarse una libreria de un tercero por ejemplo BouncyCastle
- Diffie-Hellman. A este algoritmo se le conoce como acuerdo de claves No puede

ser usado para la encritación, pero puede usarse para permitir a las dos partes derivar una clave secreta comprartiendo información por canales publicos, . Y esta clave ser usada en cifrado simetrico.

Esquema basico de cifrado y descifrado con clave publica.

Veremos que es muy parecido al uso de claves privadas.

Primero crearemos la clave

- **KeyPairGenerator.getInstance("RSA"),**
- **.initialize(1024),**
- **.generateKeyPair():**

Creamos un objeto Chiper y lo configuramos con los parámetrosdeseados

- **Cipher.getInstance("RSA/ECB/PKCS1Padding")**
- **.init(Cipher.ENCRYPT_MODE, clave.getPublic()):**

Realizamos el cifrado

- **.doFinal(textoPlano):**

Configuramos otra vez el objeto cipher

- **.init(Cipher.DECRYPT_MODE, clave.getPrivate())**

y desciframos

- **.doFinal(cipherText)**

Crear el par de claves publica/privada es casi tan sencillo como generar una calve privada, fijemonos en el siguiente segmento de codigo. He intentado que se parezca al de la seccion anterior para ver las escasas diferencias.

```
try {
    // par de claves1024-bit Digital Signature Algorithm (DSA)
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
    keyGen.initialize(1024);
    KeyPair ParClaves1 = keyGen.genKeyPair();
    PrivateKey clavePrivada1 = keypair.getPrivate();
    PublicKey  clavePublica1 = keypair.getPublic();

    // par de claves 576-bit DH
    keyGen = KeyPairGenerator.getInstance("DH");
    keyGen.initialize(576);
```

```
    KeyPair ParClaves2 = keyGen.genKeyPair();
    PrivateKey clavePrivada2= keypair.getPrivate();
    PublicKey  clavePublica2 = keypair.getPublic();


    // par de claves  1024-bit RSA
    keyGen = KeyPairGenerator.getInstance("RSA");
    keyGen.initialize(1024);
    KeyPair ParClaves3 = keyGen.genKeyPair();
    PrivateKey clavePrivada3 = keypair.getPrivate();
    PublicKey  clavePublica3= keypair.getPublic();

  } catch (java.security.NoSuchAlgorithmException e) {
  }
```

Veamos un ejemplo mas largo

```
public class PublicExample {
  public static void main (String[] args) throws Exception {
  if (args.length !=1) {  System.exit(1);    }
  byte[] textoPlano = args[0].getBytes("UTF8");

  KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
  keyGen.initialize(1024);
  KeyPair clave = keyGen.generateKeyPair();

  Cipher cifrar = Cipher.getInstance("RSA/ECB/PKCS1Padding");
  System.out.println( "\n" + cifrar.getProvider().getInfo() );

  System.out.println("\n ------------- Texto encriptado ")
  cifrar.init(Cipher.ENCRYPT_MODE, clave.getPublic());
  byte[] cipherText = cifrar.doFinal(textoPlano);
  System.out.println( new String(cipherText, "UTF8") );

  System.out.println("\n ------------- Texto normal ")
  cifrar.init(Cipher.DECRYPT_MODE, clave.getPrivate());
```

```
    byte[] newPlainText = cifrar.doFinal(cipherText);
    System.out.println( new String(newPlainText, "UTF8") );
  }
}
```

Veamos la salida, fijemonos en que esta vez el proveedor no es SUN

```
c:\MisPruebas\seguridad\Jce> java PublicExample "Hola mundo texto plano"
BouncyCastle Security Provider v1.12
---------------Texto encriptado :
Ajsd843*342l,AD;LKJL;1!*AD(XLKASD498asdjlkkKSFJHDuhpja;d(kawe#kjalfcas,
.asd+,1LKSDJf;khaouiwheyahdsl87458q9734hjfc*nuywe
--------------- Texto normal :
Hola mundo texto plano
```

Optimizando la comunicacion cifrada.

Como vimos anteriormente, el cifrado con claves asimetricas, soluciona el problema de compratir informacion sensible, pero añade dos problemas

- Incrementa la complejidad en la comunicacion, ya que debemos crear dos claves, y distinguir entre la publica y la privada. Aunque este problema es despreciable.
- Aumenta el tiempo de cifrado, este problema puede resultar bastante serio, cuando ciframos mensajes mas grandes que una simple cadena de caracteres. que pasa si ciframos un objeto que ocupa varios megas ? o un archivo de texto enriquecido (RTF).

Para solucionar el segundo problema podemos usar un esquema diferente en la comunicacion. Veamoslo.

1 Alice crea un par de claves asimetricas (una publica y una privada) asimetricas y se la manda a Bob. (recuerde que con par no nos referimos a dos, sino a al conjunto de claves que son coincidentes, es decir que la clave privada descifra el cifrado de la publica)

2 Bob Crea una clave privada (simetrica) y la cifra con la clave publica de ALICE (la clave asimetrica que genero la primera vez)

3 Alice descifra la clave privada simetrica (la que creo Bob) con su clave privada asimetrica.

4 Alice y Bob ya poseen una clave privada simetrica, y segura ya que que cifrada al enviarse. Ahora ya pueden comunicarse con un ifrado simetrico, el cual es mucho

mas rapido que el asimetrico.

La firma digital

Como sabe Bob que el mensaje realmente lo ha enviado Alice? Podria ser Mallory haciendose pasar po Alice, o podria haber modificado la respuesta que dia Alice. Para esto usamos la firma digital. La firma digital es un bloque de bytes que identifican inequivocamente al emisor y al contenido del mensaje.

Como funciona : Veamos primero un ejemplo de una conversacion privada entre Alice y Bob,

1 Alice crea una clave privada y una clave publica

2 Alice manda a Bob su clave publica, sin ninguna proteccion ya que no es informacion sensible,

3 Bob crea una clave privada y una publica

4 Bob manda a Alice su clave publica.

5 Alice escribe su mensaje y lo encripta con la clave publica de Bob

6 Alice utiliza una funcion de resumen sobre su mensaje y lo encripta con **SU PROPIA CLAVE PRIVADA** , este resumen cifrado es la firma.

7 Alice manda a Bob el mensaje y la firma a Bob

8 Bob descifra el mensaje con su clave privada.

9 Bob verifica la procedencia y la integridad del mensaje de Alice, usando la **CLAVE PUBLICA DE ALICE**

10 Si se verifica entonces el mensaje proviene de alice y ademas no ha sido alterado.

Si Mallory hubiese modificado el mensaje, o simplemente lo hubiese enviado ella, a firma no se verificaria, Por que ? Puede darse dos casos

- Si Mallory modifico el mensaje, el resumen del mensaje que recibe Bob, no coincide con el reumen que existe en la firma
- Si el mensaje no es de Alice, aunque el resumen coincide con el contenido del mensaje, Bob no puede verificarlo ya que la clave publica de Alice solo sirve para verificar un mensaje firmado por la clave privada de Alice. La cual desconoce Mallory por lo cual la tuvo que firma con otra.

Este sistema es muy seguro, pero presenta una situación en la que Mallory puede hacerse pasar por Alice. Recordemos que una funcion de resumen puede producir colisiones, es decir, dos mensajes distintos pueden producir el mismo resumen. pro lo que si mallory es capaz de encontrar una colision para el mensaje, podria escribir un nuevo mensaje cifrarlo

con la clave publica de Bob y dejar la firma del mensaje original de Alice. Este caso es muy improbable que se de, primero es muy dificil encontrar una colisión, y segundo es mucho mas dificil que esa colision tenga sentido. Por ejemplo Alice escribe a Bob para confirmar una comprar "Si, quiero realizar la compra" y Mallory desea realizar la compra ella. por lo que deberia encontrar una colision y que ademas expresase lo contrario algo como "No deseo realizar la compra" y existen una posibilidades minimas de que exista esa (u otra colision) que pueda expresar eso.

Otro metodo para asegurarnos la autenticidad de quien proviene el mensaje seria usar la Apir CerthPath y el concurso de una tercera parte.

Esquema basico para realizar y comprobar una firma digital.

Creamos una clave

- **KeyPairGenerator.getInstance("RSA")**
- **.initialize(1024)**
- **.generateKeyPair()**

Creamos una firma del objeto y la inicializamos

- **Cipher.getInstance("MD5WithRSA")**
- **.initSign(clave.getPrivate())**

Calculamos la firma apartir del un texto plano y firmamos

- **.update(textoPlano)**
- **.sign()**

Configuramos la verificación y verificamos la firma

- **.initVerify(clave.getPublic())**
- **.verify(signature): Verifies the signature.**

Veamos una clase que nos crea la firma de un String:

```
public static byte[] createSignature(PrivateKey clave, String Str) {
    try {
        //-----------    Pasamos a Bytes para hacer bien el cifrado
        byte[] cadenaByte= str.getBytes("UTF8");

        Signature sig = Signature.getInstance(clave.getAlgorithm());
        sig.initSign(clave);
```

```
        sig.update(cadenaByte, 0, cadenaByte.length);
        return sig.sign();
      } catch (SignatureException e) {
      } catch (InvalidKeyException e) {
      } catch (NoSuchAlgorithmException e) {
      }
      return null;
    }
```

El ejemplo anterior, es una forma de crear una firma digital aunque no es la unica, si es la mas correcta. En el siguiente ejemplo crear la firma es aun un poco mas sencillo.Veamos ahora ejemplo del uso de la firma digital, tanto firmar, como verificar :

```
public class EjemploFirma {
  public static void main (String[] args) throws Exception {
    if (args.length !=1) {  System.exit(1);  }
    byte[] textoPlano = args[0].getBytes("UTF8");

    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
    keyGen.initialize(1024);
    KeyPair clave= keyGen.generateKeyPair();
   clave
    Signature sig = Signature.getInstance("MD5WithRSA");
    sig.initSign(clave.getPrivate());
    sig.update(textoPlano);
    byte[] signature = sig.sign();

    System.out.println( sig.getProvider().getInfo() );
    System.out.println( "\n Firma :" );
    System.out.println( new String(signature, "UTF8") );

    sig.initVerify(clave.getPublic());
    sig.update(textoPlano);

    try {
```

```
    if (sig.verify(signature)) {    System.out.println( "Verificación positiva" );      }
    else System.out.println( "Fallo la verificacion" );
  } catch (SignatureException se) { System.out.println( "Fallo la verificacion" );   }
 }
}
```

Firmar un objeto.

e205. Signing a Java Object

Un objeto "serialiable" se puede firmar con una llave privada. Cuando se ha creado una copia firmada, los cambios en el no afectan al objeto firmado, esto es logico ya que si no la firma no coincidiria con el objeto firmado al estar modificado

```
public class micalse implements Serializable {
   String s = "mi cadena";
   int i = 123;
}
try {
   //  creamos el par de llaves para un algoritmo DSA con clave de 1024 bits
   KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
   keyGen.initialize(1024);
   KeyPair keypair = keyGen.genKeyPair();
   PrivateKey privateKey = keypair.getPrivate();
   PublicKey publicKey = keypair.getPublic();
} catch (NoSuchAlgorithmException e) {
}

// Creamos una instancia del SignedObject
SignedObject so = null;
try {
   Serializable o = new micalse();
   Signature sig = Signature.getInstance(privateKey.getAlgorithm());
    // firmamos
   so = new SignedObject(o, privateKey, sig);
} catch (NoSuchAlgorithmException e) {
} catch (SignatureException e) {
} catch (InvalidKeyException e) {
} catch (IOException e) {
```

```
}
```

Para verificar un objeto firmado.

```
try {
        // creamos una instancia de Signature indicandole las llaves en vez del algoritmo.
    Signature sig = Signature.getInstance(publicKey.getAlgorithm());

    boolean b = so.verify(publicKey, sig);

    // Obtenemos el objeto
    micalse o = (micalse)so.getObject();
} catch (SignatureException e) {
} catch (InvalidKeyException e) {
} catch (NoSuchAlgorithmException e) {
} catch (ClassNotFoundException e) {
} catch (IOException e) {
}
```

Con eto terminamos el apartado de criptografía, se nos quedan cosas por ver, pero estas se salen de los limites del tutorial.

capítulo 5

Seguiridad en comunicaciones sockets

En desarrollo. Si tienes alguna sugerencia o aportacion sobre este tema, realiza una **anotación**.

capítulo 6

Seguridad en entornos J2ME

En desarrollo. Si tienes alguna sugerencia o aportacion sobre este tema, realiza una **anotación**.