Egibide

Unidad Didáctica 5:

TÉCNICAS DE PROGRAMACIÓN SEGURA

1. Seguridad

Los aspectos fundamentales de la seguridad en las comunicaciones digitales son los siguientes:

- **Integridad:** asegura que los datos no se alteran.
- **Confidencialidad:** evita que terceros lean la información.
- **Autenticación:** verifica la identidad del emisor.
- **No repudio:** garantiza que el emisor no puede negar el envío.

2.Programación segura.

La programación segura implica una forma de trabajar que minimiza las vulnerabilidades en el código y protege el *software* contra posible ciberamenazas, es decir, no solo es importante crear un código funcional, sino que también los programadores deben centrarse en integrar las prácticas adecuadas para hacerlo seguro.
Algunos aspectos claves de la programación segura son los siguientes:

- Validación de entradas de usuario para prevenir ataques de inyección en los que los ciberdelincuentes inyectan código malicioso a través de formularios
- Manejo de errores seguro, de forma que no se revele información confidencial que pueda ser utilizada de forma inadecuada por un ciberdelincuente.
- Gestión de memoria segura para evitar errores como desbordamientos de búfer.
- Autenticación segura para proteger la identidad de los usuarios y restringir los accesos no autorizados
- Cifrado para proteger la información confidencial.
- Actualizaciones y parches de seguridad que corrijan vulnerabilidades conocidas.
- Pruebas de seguridad regulares para identificar posibles vulnerabilidades.
- Validar en cada paso, gestión o acción los privilegios o permisos del usuario.

2.1 Validación de entradas.

Una vía importante de errores de seguridad y de inconsistencia de datos dentro de una aplicación se produce a través de los datos que introducen los usuarios. Un fallo de seguridad muy frecuente, consiste en los errores basados en buffer overflow, se producen cuando se desborda el tamaño de una determinada variable, vector, matriz, etc. y se consigue acceder a zonas de memoria reservadas.

La validación de datos permite:

- Mantener la consistencia de los datos. Por ejemplo, si a un usuario le indicamos que debe introducir su DNI éste debe tener el mismo formato siempre.

- Evitar desbordamientos de memoria buffer overflow. Al comprobar el formato y la longitud del campo evitamos que se produzcan los desbordamientos de memoria.

La validación de datos en Java es un proceso fundamental para asegurarse de que los datos insertados o procesados cumplan con ciertos criterios antes de ser utilizados en la lógica del programa. Existen varias formas de hacer validación de datos en Java, desde las más simples utilizando condicionales hasta las más complejas usando expresiones regulares y clases de validación.

Validación de tipos de datos de Entrada ,usando Scanner

Cuando se obtiene la entrada del usuario a través de la consola, es común validar que el dato insertado sea del tipo esperado. Debemos verificar si el valor que leemos corresponde al tipo de datos esperado por la clase Scanner. Para eso nos ayudaremos de la función **.hasNextXYZ**() en donde XYZ es el tipo de datos que nos interesa. Esta función devolverá verdadero (true) o falso (false) según corresponda, (hasNextInt(), hasNextBoolean(), hasNextDouble(), hasNextByte() .

Validación de un número entero

```
import java.util.Scanner;

public class ValidacionDatos {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int numero = 0;

        // Validación de que el número sea un entero
        System.out.print("Introduce un número entero: ");
        while (!scanner.hasNextInt()) {
            System.out.println("¡Por favor introduce un número válido!");
            scanner.next(); //
        }
        numero = scanner.nextInt();

        System.out.println("El número introducido es: " + numero);

        scanner.close();
    }
}
```

Validación Usando Expresiones Regulares

Las expresiones regulares (regex) son útiles para validar formatos específicos como correos electrónicos, números de teléfono, etc.

Las expresiones regulares permiten definir exactamente el formato de la entrada de datos.

El API de Java para expresiones regulares está incluido de base en el JDK. Para usar expresiones regulares se usa el package java.util.regex. Contiene las clases Pattern y Matcher y la excepción PatternSyntaxException.

Clase Pattern: Un objeto de esta clase representa la expresión regular, también llamadas plantillas. Contiene el método compile(String regex) que recibe como parámetro la expresión regular y devuelve un objeto de la clase Pattern.

La clase Matcher: Esta clase compara el String y la expresión regular. Contienen el método matches(CharSequence input) que recibe como parámetro el String a validar y devuelve true si coincide con el patrón. El método find() indica si el String contienen el patrón.

Los objetos de la clase PatternSyntaxException sirven para describir patrones de expresiones regulares no validos.

Listado de los símbolos que puede usar y qué significa cada uno de ellos:

Expresión	Descripción
.	Un punto indica cualquier carácter
^expresión	El símbolo ^ indica el principio del String. En este caso el String debe contener la expresión al principio.
expresión$	El símbolo $ indica el final del String. En este caso el String debe contener la expresión al final.
[abc]	Los corchetes representan una definición de conjunto. En este ejemplo el String debe contener las letras a ó b ó c.
[abc][12]	El String debe contener las letras a ó b ó c seguidas de 1 ó 2
[^abc]	El símbolo ^ dentro de los corchetes indica negación. En este caso el String debe contener cualquier carácter excepto a ó b ó c.
[a-z1-9]	Rango. Indica las letras minúsculas desde la a hasta la z (ambas incluidas) y los dígitos desde el 1 hasta el 9 (ambos incluidos)
A|B	El carácter | es un OR. A ó B
AB	Concatenación. A seguida de B

Metacaracteres

Expresión	Descripción
\d	Dígito. Equivale a [0-9]
\D	No dígito. Equivale a [^0-9]
\s	Espacio en blanco. Equivale a [\t\n\x0b\r\f]
\S	No espacio en blanco. Equivale a [^\s]
\w	Una letra mayúscula o minúscula, un dígito o el carácter _ Equivale a [a-zA-Z0-9_]

En Java debemos usar doble barra invertida \\.

Cuantificadores

Expresión	Descripción
{X}	Indica que lo que va justo antes de las llaves se repite X veces
{X,Y}	Indica que lo que va justo antes de las llaves se repite mínimo X veces y máximo Y veces. También podemos poner {X,} indicando que se repite un mínimo de X veces sin límite máximo.
*	Indica 0 ó más veces. Equivale a {0,}
+	Indica 1 ó más veces. Equivale a {1,}
?	Indica 0 ó 1 veces. Equivale a {0,1}

Para utilizar las expresiones regulares debemos realizar los siguientes pasos:

1. Importamos la librería:
   import java.util.regex.;*
2. Definimos Pattern y Matcher:
   Pattern pat=null;
   Matcher mat=null;
3. Compilamos el patrón a utilizar. pat=Pattern.compile(patron); donde el patrón a comprobar es la parte más importante ya que es donde tenemos que indicar el formato que va a tener la entrada.
   A modo de ejemplo, a continuación se muestran varios ejemplos de expresiones.
   Teléfono (formato 000-000000)
   pat=Pattern.compile("[0-9]{3}-[0-9]{6}");
   DNI
   pat=Pattern.compile("[0-9]{8}-[a-zA-Z]");
   Cadenas de texto validas
   pat=Pattern.compile("PSP","AD","IPE");
   Comprobar si el String cadena esta formado por minimo 5 letras mayusculas o minusculas y un maximo de 10.
   pat = Pattern.compile("[azAZ]{5,10}");
   Comprobar si el String cadena solo contiene los caracteres V o F
   pat = Pattern.compile("(V|F)+");
   Le pasamos al evaluador de expresiones el texto a comprobar.
   mat=pat.matcher(texto_a_comprobar);
4. Comprobamos si hay alguna coincidencia:
   if(mat.find())
   {
   // Coincide con el patrón
   }
   Else
   {
   // NO coincide con el patrón
   }

Hay ocasiones en las que se quiere ser mas exhaustivo en la búsqueda del patrón, para ello se utiliza `mat.matches()` para hacer la comparación.

2.2 Excepciones

Una excepción es un evento que ocurre durante de la ejecución de un programa e interrumpe el flujo normal de las instrucciones.
Es importante gestionar las posibles excepciones que puedan ocurrir en el programa para evitar cualquier tipo de fallos en su ejecución. De forma general, para incluir el manejo de excepciones en un programa hay que realizar los siguientes pasos:

- Dentro de un bloque try insertar el flujo normal de las instrucciones.
- Estudiar los errores que pueden producirse durante la ejecución de las instrucciones y comprobar que cada uno de ellos provoca una excepción.
- Capturar y gestionar las excepciones en bloques catch.

```
try {
   FileReader f = new FileReader("datos.txt");
} catch (IOException e) {
   System.out.println("No se pudo abrir el archivo.");
}
```

2.3 Ficheros de registro

Los ficheros de registro o logs, permiten almacenar información sobre las distintas acciones que realiza la aplicación. Estos ficheros permiten realizar un seguimiento detallado de lo que ocurre en el sistema.
El sistema de logging en Java es una herramienta poderosa para registrar información sobre la ejecución de una aplicación, permitiendo a los desarrolladores monitorear y depurar el comportamiento del sistema.
Java incorpora la librería java.util.logging que permite utilizar la clase logger para llevar un control de todos los eventos que ocurren a lo largo de la ejecución de la aplicación.
El sistema de logging en Java se compone de varios componentes que interactúan entre sí para gestionar el registro de eventos y mensajes. Los componentes más importantes son:

- **Logger**: Es el objeto principal que se utiliza para generar los mensajes de log. Un Logger es responsable de manejar el flujo de mensajes de log, así como de asignar un nivel de severidad a esos mensajes. Cada clase o componente de la aplicación puede tener su propio Logger.
- **Handler**: Los Handler son los componentes encargados de definir el destino de los mensajes de log. Por ejemplo, los mensajes de log pueden ser enviados a la consola, a un archivo, a una base de datos, a un servidor remoto, etc. Algunos de los Handler más comunes son:
    - ConsoleHandler: Envía los mensajes de log a la consola.
    - FileHandler: Envía los mensajes de log a un archivo.
    - SocketHandler: Envía los mensajes a través de un socket a otro sistema.
- **Formatter**: Los Formatter definen cómo se presenta el mensaje de log. Por ejemplo, un SimpleFormatter puede mostrar la fecha, la hora, el nivel de log y el mensaje, mientras que un XMLFormatter presenta los mensajes en formato XML.

La forma en que opera el framework de logging de Java es la siguiente:

- Creamos un objeto estático de tipo Logger desde el cual enviaremos los mensajes a registrar.

- Creamos un objeto ConsoleHandler y se lo agregamos al Logger, de modo que los mensajes aparezcan automáticamente en la consola.
- Creamos un objeto FileHandler y se lo agregamos al Logger, este Handler en particular enviara los mensajes al archivo que le indiquemos.
- Creamos un objeto SimpleFormatter y lo establecemos en el FileHandler, de este modo los logs se escriban como texto plano simple, de no indicarlo se escribirán en formato XML por defecto
- Para registrar algo llamamos al método log del Logger indicamos el nivel del log y el mensaje que deseamos registrar, esto automáticamente reportara la fecha, hora, el nombre completo de la clase y el método y en el caso de mensajes de nivel grave la línea de código donde se genero el reporte.

Para trabajar con ficheros de registro de Java debemos realizar los siguientes pasos:

a. Importamos la librería:
```
import java.util.logging.*;
```

b. Buscar o crear el logger que queremos utilizar:
```
Logger logger=Logger.getLogger("MyLog");
```

c. Una vez definido el logger debemos asociarlo a un fichero log:
```
FileHandler fh= new FileHandler("c:\\MyLogFile.log",true);
```

Donde MyLogFile.log es el fichero donde se guardan el registro de la aplicación. Y la variable true indica que los registros se añadan a los ya existentes, si establecemos false, el fichero se reinicia.

d. Establecer si queremos visualizar los mensajes de log por pantalla. En caso de no querer ver los mensajes por pantalla lo indicamos de la siguiente forma:
```
logger.setUseParentHandlers(false);
```

e. Establecer el formato del fichero. Los tipos de formato pueden ser:
   i. Fichero de texto simple(SimpleFormatter)
```
SimpleFormatter formato = new SimpleFormatter();
fh. setFormatter(formato);
```

   ii. Fichero en formato XML(`XMLFormatter`)
```
XMLFormatter formato = new XMLFormatter ();
```

f. Establezco el nivel de seguridad de las actividades que quiero registrar:

Niveles de seguridad de los registros

Nivel	Importancia
SEVERE	7 (Máxima)
WARNING	6
INFO	5
CONFIG	4
FINE	3
FINER	2
FINEST	1 (Mínima)

Por ejemplo, si deseo registrar solo los registros mas graves ejecutamos

```
logger.setLevel(Level.SEVERE);
```

Además, de los niveles anteriores, si queremos registrar todos los eventos utilizaremos:

```
Logger.setLevel(Level.ALL);
```

Y si no queremos registrar ningún evento, ejecutaremos:

```
Logger.setLevel(Level.OFF);
```

Una vez inicializado correctamente el logger, el siguiente paso es ir registrando en la aplicación los diferentes registros. Por ejemplo, para añadir un registro de nivel de importancia medio (WARNING) lo creamos de la siguiente forma:

logger.log(Level.WARNING,”Mi primer log”);
Para cargar los logs en vuestro fichero:
logger.addHandler(fh);

3. Seguridad en Java

Java incorpora un amplio conjunto de APIs y herramientas para implementar seguridad en aplicaciones: control de acceso, criptografía, certificados, firmas digitales y comunicaciones seguras.
Estas funcionalidades se agrupan principalmente en:

1. Control de acceso y permisos
2. Criptografía y funciones resumen
3. Cifrado simétrico y asimétrico
4. Firmas digitales
5. Certificados digitales
6. Comunicaciones seguras (SSL/TLS, JSSE)
7. Autenticación y autorización con JAAS

3.1 APIs de Seguridad en Java

Java proporciona tres pilares principales de seguridad:

- JCA (Java Cryptography Architecture)
  Incluye las clases básicas de criptografía (resúmenes, firmas, claves...).
- JCE (Java Cryptography Extension)
  Extiende JCA e incorpora algoritmos de cifrado (AES, RSA, etc.).
- JSSE (Java Secure Socket Extension)
  Implementa SSL y TLS para comunicaciones seguras.
- JAAS (Java Authentication and Authorization Service)
  Permite autenticación y autorización centrada en el usuario.

3.2 Proveedores Criptográficos

Un proveedor criptográfico es una librería que implementa algoritmos de seguridad (hash, cifrado, firmas).
Java usa JCA/JCE para definir las reglas, pero los proveedores contienen la implementación real.

Ejemplo de proveedores:

- SUN (proveedor interno incluido en el JDK)
- BouncyCastle (muy usado ampliamente)

Cada proveedor implementa clases denominadas *engines*:

Engine	Función
MessageDigest	Hash o función resumen
Signature	Firmas digitales
KeyFactory	Manipulación de claves
KeyPairGenerator	Generación de claves públicas/privadas
Cipher	Cifrado y descifrado simétrico/asimétrico
SecureRandom	Generación de números aleatorios seguros

Cada engine se obtiene con:

```
Cipher c = Cipher.getInstance("AES");
MessageDigest md = MessageDigest.getInstance("SHA-256");
```

También se puede hacer la petición del algoritmo con el nombre del proveedor:

```
MessageDigest md = MessageDigest.getInstance("SHA-256", "SUN");
```

4. Funciones resumen

Un *Message digest* o resumen de mensaje, más conocidos como funciones hash, es una marca digital de un bloque de datos. Genera una huella digital de los datos.
Existe un gran número de algoritmos diseñados para procesar esto resúmenes, los dos más conocidos son SHA-256,SHA-512(seguros) y MD5(obsoleto).

De un resumen cabe destacar las siguientes características:

- Para el mismo algoritmo, el resumen siempre tiene el mismo tamaño, independientemente del tamaño de los datos que se haya usado para generarlo. (MD5:128 bits, SHA-1:160 bits,SHA-256:256 bits,SHA-512:512 bits)
- Es imposible recuperar la información original a partir de un resumen.
- Es computacionalmente inviable encontrar dos mensajes que tengan el mismo valor de resumen. Matemáticamente es altamente improbable, pero no imposible.
- Un pequeño cambio en los datos resumidos genera un resumen completamente diferentes.

Usos:

- Verificación de integridad
- Almacenamiento seguro de contraseñas
- Validación de archivos descargados

4.1 MessageDigest

La clase *MessageDigest* permite a las aplicaciones implementar algoritmos de resumen criptográficamente seguros como SHA-256 o SHA-512
Para generar un hash en java se procede de la siguiente forma:
1. Se crea un objeto de la clase *MesageDigest* con el método estático *getInstance()* de la misma clase, especificando el nombre del algoritmo.
2. Se añaden datos con el método *update()*. Se puede añadir un byte o un array de bytes. Este método se puede invocar varias veces para ir añadiendo nuevos datos.
3. Se obtiene el valor de hash con el método *digest()*.
4. Si se quisiera calcular un nuevo hash, se invocaría el método *reset()* para volver a empezar el proceso.
A continuación podemos ver un ejemplo

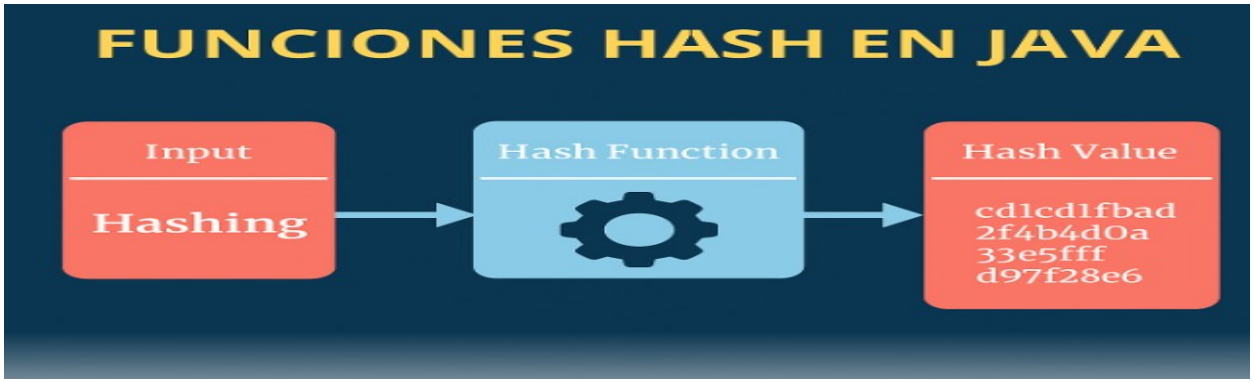
```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class PruebaMessageDigest {

    public static void main(String[] args) {
        String texto = "Esto es un texto plano.";

        try {
            MessageDigest md = MessageDigest.getInstance("SHA-256");
            md.update(texto.getBytes());
            byte[] digest = md.digest();
            System.out.println("Resumen (hex): " + toHexadecimal(digest));

        } catch (NoSuchAlgorithmException e) {
            System.err.println("Algoritmo no disponible.");
        }
    }
/*Este método convierte un array de bytes en una cadena hexadecimal legible, asegurándose de que cada byte se represente con exactamente dos caracteres y usando mayúsculas. */
    private static String toHexadecimal(byte[] hash) {
        StringBuilder hex = new StringBuilder();
        for (byte b : hash) {
            String h = Integer.toHexString(b & 0xFF);
            if (h.length() == 1) hex.append("0");
            hex.append(h);
        }
        return hex.toString().toUpperCase();
    }
}
```



5.Cifrado de datos

El cifrado transforma información legible en datos ilegibles que solo pueden recuperarse usando la clave correcta.

Tipos de encriptado

En la industria de la seguridad informática, hay dos tipos de cifrado de datos muy usados, llamados cifrado simétrico y asimétrico. La finalidad de ambas es la de encriptar los datos, pero son diferentes en cuanto a la forma en la que trabajan, sus ventajas y sus limitaciones.

Diferencias entre su uso en cifrado simétrico y asimétrico

Aspecto	Simétrico (AES)	Asimétrico (RSA)
Clave	Una sola clave secreta	Clave pública / clave privada
Velocidad	Muy rápido	Mucho más lento
Uso	Cifrar datos	Cifrar claves simétricas
Tamaño máximo de datos	Ilimitado	Muy limitado (depende de bits de la clave)
Seguridad	Depende de la clave	Basada en problemas matemáticos (RSA/ECC)

5.1 La clase Cipher

La clase `Cipher` es el motor principal de cifrado y descifrado en Java.
Pertenece a la Java Cryptography Extension (JCE) y sirve como punto de entrada para todos los algoritmos de cifrado, tanto simétricos (AES, DES…) como asimétricos (RSA).

¿Que es Cipher?

Es un *engine* que representa un algoritmo de cifrado.
Java define la interfaz, pero la implementación real la proporcionan los proveedores criptográficos (SUN, BouncyCastle, etc.).

Cada vez que se quiere cifrar o descifrar algo, se debe:

1. Solicitar una instancia del algoritmo
2. Inicializar el Cipher con la clave y el modo
3. Ejecutar la operación mediante doFinal()

5.1.1 Fases para trabajar con Cipher

- **Obtener una instancia del algoritmo**

  Se especifica el algoritmo, o el algoritmo + modo + padding:

  ```
  Cipher cipher = Cipher.getInstance("AES");
  Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
  Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
  ```

  También se puede especificar proveedor:

  ```
  Cipher cipher = Cipher.getInstance("AES", "BC");
  ```

  > Padding: es un concepto clave en criptografía, sobre todo en cifrado de bloques. Cuando un algoritmo de cifrado de bloques (como AES, DES) procesa datos, trabaja con bloques de tamaño fijo. Si el mensaje no llena exactamente un bloque, se debe añadir relleno (padding) para completar el bloque. Esto asegura que todos los bloques tengan el tamaño necesario para el cifrado.

- **Inicializar el Cipher**

  El modo determina qué queremos hacer:

  - Cipher.ENCRYPT_MODE → cifrar
  - Cipher.DECRYPT_MODE → descifrar

  Ejemplo simétrico (AES):

  ```
  cipher.init(Cipher.ENCRYPT_MODE, claveSimetrica);
  ```

  Ejemplo asimétrico (RSA):

  ```
  cipher.init(Cipher.ENCRYPT_MODE, clavePublica);
  ```

- **Ejecutar la operación**

  Todas las operaciones de cifrado y descifrado finalizan con:

  ```
  byte[] resultado = cipher.doFinal(datos);
  ```

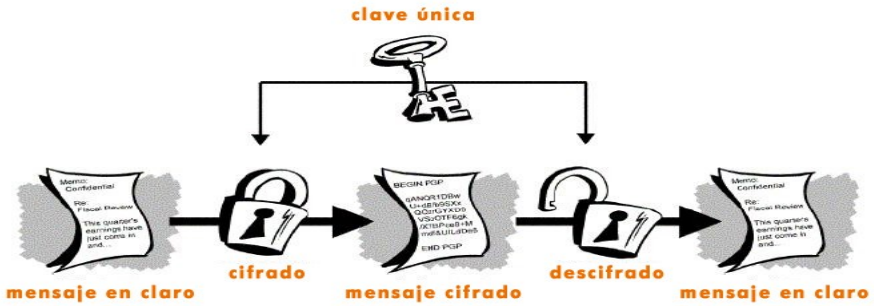
  Este método realiza todo lo necesario internamente:

  - aplicar el modo
  - rellenar con padding si es necesario
  - dividir en bloques
  - procesar todo el flujo de datos

5.2 Cifrado simétrico

El **cifrado simétrico**, también conocido como algoritmo de clave privada o de clave compartida, usa la misma clave para encriptar y desencriptar. Los algoritmos de cifrado de claves simétricas se consideran menos costosos de producir y no requieren tanta potencia de computación para encriptarlos y descifrarlos, lo que significa que se tarda menos en decodificar los datos.

El problema principal es el intercambio seguro de la clave. Por lo que si un usuario no autorizado consigue las claves, podrá descifrar los mensajes y los datos que se envíen entre las partes.



Entre los algoritmos de cifrado simétrico más utilizados se encuentran:

a) AES (Advanced Encryption Standard)

- **Uso**: Es el algoritmo de cifrado simétrico más utilizado en la actualidad, especialmente en aplicaciones que requieren una alta seguridad.
- **Tamaño de clave**: 128 bits, 192 bits, o 256 bits.
- **Características**: AES es altamente eficiente y seguro, utilizado en una amplia variedad de aplicaciones como:
    - VPNs (Virtual Private Networks).,SSL/TLS ,Encriptación de archivos y discos ,Banca on-line y pagos electrónicos.

b) DES (Data Encryption Standard)

- **Uso**: Fue ampliamente utilizado, pero ha quedado obsoleto debido a sus vulnerabilidades (el tamaño de la clave es solo de 56 bits, lo que permite ataques de fuerza bruta).
- **Características**: Aunque ya no se recomienda para nuevas implementaciones, aún se usa en sistemas heredados en algunos casos.

c) 3DES (Triple DES)

- **Uso**: Es una mejora de DES que aplica tres pasadas de cifrado con DES, usando tres claves diferentes (aunque algunas implementaciones usan la misma clave para las tres pasadas).
- **Características**: Aunque más seguro que DES, **3DES** también es más lento y no es tan eficiente como AES, por lo que se ha ido reemplazando gradualmente.

d) Blowfish

- **Uso**: Fue diseñado como un algoritmo de reemplazo para DES. Aunque ya no es tan popular como AES, aún se utiliza en algunas aplicaciones de seguridad.
- **Características**: Cifrado simétrico de bloques con claves de longitud variable (32-448 bits).
- **Aplicación**: Se encuentra en algunos productos de software y protocolos de cifrado.

Ejemplo de cifrado simétrico con AES:

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class CifradoSimetrico {

    public static byte[] cifrar(String texto, SecretKey clave) throws Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, clave);
        return cipher.doFinal(texto.getBytes());
    }

    public static String descifrar(byte[] cifrado, SecretKey clave) throws Exception {
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.DECRYPT_MODE, clave);
        return new String(cipher.doFinal(cifrado));
    }

    public static void main(String[] args) {
        try {
            KeyGenerator kg = KeyGenerator.getInstance("AES");
            kg.init(128);
            SecretKey clave = kg.generateKey();

            String original = "Mensaje secreto";
            byte[] cifrado = cifrar(original, clave);
            String descifrado = descifrar(cifrado, clave);

            System.out.println("Original: " + original);
            System.out.println("Descifrado: " + descifrado);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.3 Cifrado asimétrico

El **cifrado asimétrico**, también conocido como criptografía de claves públicas,utiliza dos claves para su proceso de cifrado: la clave pública, utilizada para el cifrado, y la clave privada, utilizada para el descifrado. Estas claves están relacionadas, conectadas. La clave pública está disponible para cualquiera que necesite encriptar una información. Esta clave no sirve para el

proceso de descifrado. Un usuario necesita tener una clave secundaria, es decir, la clave privada, para descifrar esta información. De este modo, la clave privada sólo la tiene el actor responsable de descifrar la información.

Los algoritmos mas usados son:

a) RSA

- **Uso**: Uno de los algoritmos de cifrado asimétrico más utilizados.
- **Características**:
  - **RSA** es comúnmente utilizado en la autenticación y el intercambio de claves (por ejemplo, en el establecimiento de conexiones seguras).
  - Se usa en sistemas de firma digital, como en **SSL/TLS**, **PGP** y **S/MIME**.
  - **RSA** también se utiliza para cifrar claves simétricas que luego se utilizan en el cifrado de datos reales.
- **Limitación**: Aunque RSA es muy seguro, es relativamente lento comparado con los algoritmos de cifrado simétrico, por lo que generalmente se usa solo para el intercambio de claves, no para cifrar grandes volúmenes de datos directamente.

b) ECC (Elliptic Curve Cryptography)

- **Uso**: Basado en curvas elípticas, ECC es cada vez más popular por su eficiencia en términos de seguridad y tamaño de clave.
- **Características**:
  - Para la misma seguridad que RSA, **ECC** puede utilizar claves mucho más pequeñas (por ejemplo, una clave ECC de 256 bits es equivalente a una clave RSA de 3072 bits).
  - Es ampliamente utilizado en aplicaciones móviles y dispositivos con recursos limitados.

Ejemplo de cifrado asimetrico con RSA:

```
import java.security.*;
import javax.crypto.Cipher;

public class CifradoAsimetrico {

    public static KeyPair generarClaves() throws Exception {
        KeyPairGenerator gen = KeyPairGenerator.getInstance("RSA");
        gen.initialize(2048);
        return gen.generateKeyPair();
    }

    public static byte[] cifrar(String msg, PublicKey clave) throws Exception {
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, clave);
        return cipher.doFinal(msg.getBytes());
    }

    public static String descifrar(byte[] msg, PrivateKey clave) throws Exception {
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.DECRYPT_MODE, clave);
        return new String(cipher.doFinal(msg));
    }

    public static void main(String[] args) {
        try {
            KeyPair claves = generarClaves();
            String original = "Mensaje secreto";

            byte[] cifrado = cifrar(original, claves.getPublic());
            String descifrado = descifrar(cifrado, claves.getPrivate());

            System.out.println("Descifrado: " + descifrado);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.4 Gestión de claves con el paquete java.security.

En la generación de claves se utilizan números aleatorios seguros, que son números aleatorios que se generan en base a una semilla. Esto permite crear algoritmos seguros, pues será muy difícil determinar los valores generados sin conocer la semilla.
El paquete java.security proporciona las siguientes clases para la gestión de claves:

- El interface Key, permite la representación de claves, su almacenamiento y envío de forma serializada dentro de un sistema. Se trata de un interface Serializable y proporciona entre otros los siguientes métodos:
  - getAlgorithm(). Devuelve el nombre del algoritmo con el que se ha generado la clave (RSA,DES, etc.).
  - getEncoded(). Devuelve la clave como un array de bytes.
  - getFormat(). Devuelve el formato con el que está codificada la clave.

- La clase KeyPairGenerator permite la generación de claves públicas y privadas (asimétricas). Genera objetos del tipo KeyPair, que a su vez contienen un objeto del tipo PublicKey y otro del tipo PrivateKey.
    - El método initialize() permite establecer el tamaño de la clave y el número aleatorio a partir del cual será generada.
- La clase KeyGenerator permite la generación de claves privadas (simétricas). Genera objetos de tipo SecretKey.
    - El método init() permite establecer el tamaño de la clave y el número aleatorio a partir del cual será generada.

Ejemplos:
Generacion de llaves simetrico

- Aleatoria:
  ```
  import javax.crypto.KeyGenerator;
  import javax.crypto.SecretKey;

  KeyGenerator keyGen = KeyGenerator.getInstance("AES");
  keyGen.init(256); // tamaño de la llave en bits
  SecretKey key = keyGen.generateKey();
  ```

- Derivada de contraseña:
  ```
  import javax.crypto.spec.SecretKeySpec;
  import java.security.MessageDigest;

  String password = "miContraseñaSegura";
  MessageDigest sha = MessageDigest.getInstance("SHA-256");
  byte[] keyBytes = sha.digest(password.getBytes("UTF-8"));
  SecretKeySpec keySpec = new SecretKeySpec(keyBytes, "AES");
  ```

Genracion de llaves asimetrico:

```
import java.security.KeyPair;

import java.security.KeyPairGenerator;

import java.security.PrivateKey;

import java.security.PublicKey;


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

keyGen.initialize(2048); // tamaño de la llave
```

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

PublicKey publicKey = pair.getPublic();

PrivateKey privateKey = pair.getPrivate();
```

6. Firma Digital

La firma digital hace referencia a un método de criptografía que asocia la identidad del emisor al mensaje que se transmite a la vez que se comprueba la integridad del mensaje. Esta firma digital se consigue mediante la aplicación de una función hash. A continuación, se emplea un sistema de clave pública para codificar mediante la clave privada el resumen obtenido anteriormente para autentificar el mensaje.

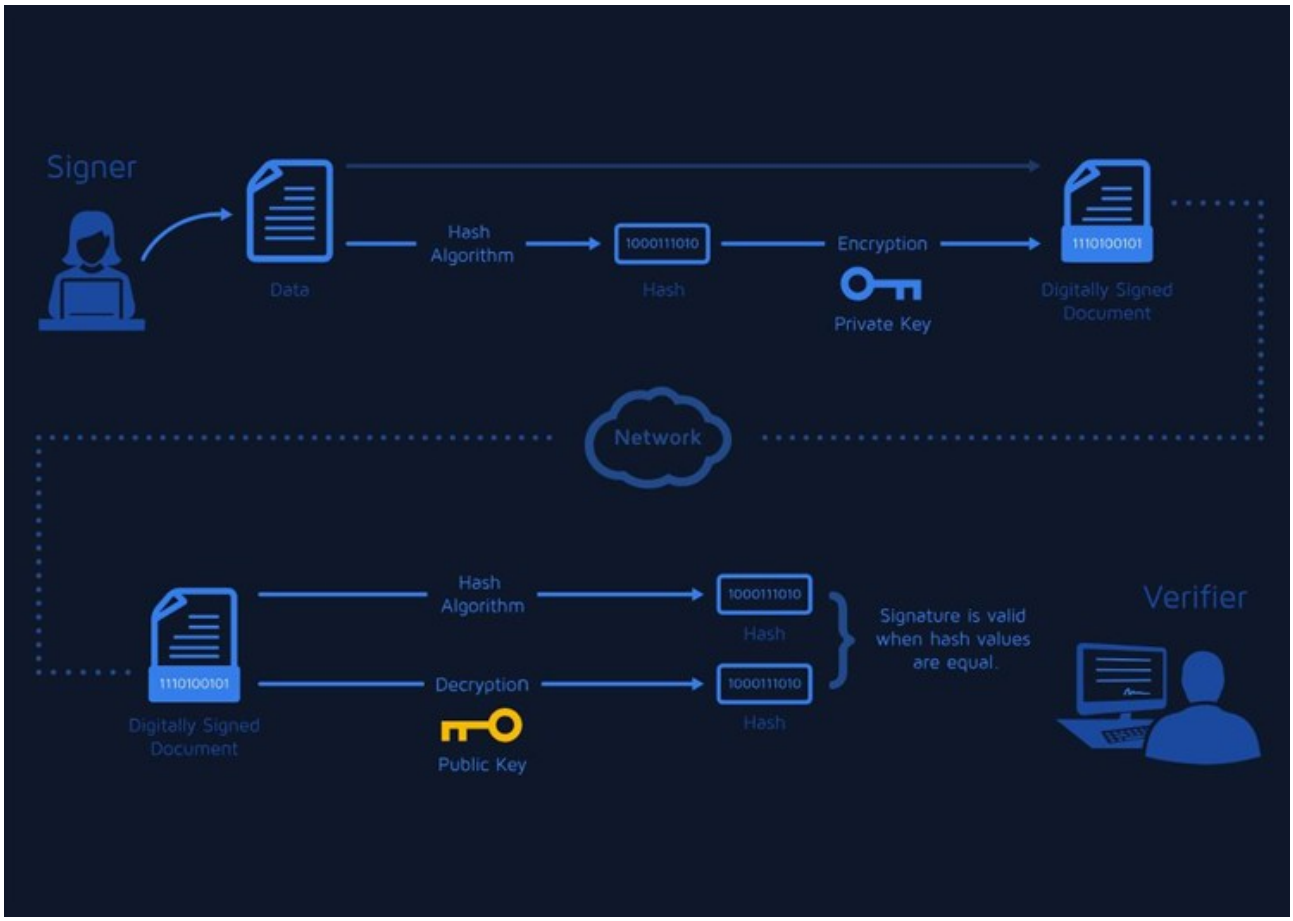
La firma garantiza:

- Autenticidad(quien envía el mensaje)
- Integridad(no ha sido modificado)
- No repudio(el remitente no puede negarlo)

La firma electrónica o digital se realiza siguiendo los siguientes pasos:

- Se calcula el resultado de una función resumen sobre el documento que vamos a firmar, por ejemplo SHA.
- El resultado obtenido de aplicar la función resumen al documento se encripta. Para ello se utiliza la clave privada de nuestra pareja de claves publica-privada. Esto permite asegurarnos de que la única persona que ha podido firmar el documento soy yo, el único que conoce la clave privada.
- El resultado de este valor es el que se conoce como firma digital. La firma digital nada tiene que ver con el cifrado del documento. En ningún momento hemos cifrado el archivo, y es que si pensamos en el proceso de la firma manuscrita sucede que cuando firmamos un papel no lo estamos cifrando. Esto no quiere decir que no puedo, también, cifrar y ademas firmar el documento. También podemos deducir que dos documentos distintos firmados digitalmente por una misma persona tendrán firmas digitales distintas, pues los valores resumen del documento nunca serán iguales.

- El receptor del documento, desencripta el resultado de la función resumen utilizando la clave publica del firmante y vuelve a calcular la función resumen sobre el documento. Al ser imposible que dos documentos que no sean idénticos generen la misma función resumen, el receptor asegura que el que firmó el documento es quien dice ser.



Programación de firmado digital

La biblioteca de Java incluye la clase *Signature* (*java.security.Signature*) para realizar operaciones de firmado digital. Para ello, se requiere de un par de claves.

Se usa de forma similar al cálculo de funciones hash, y sirve además para realizar verificación de firma.

Se debe indicar el nombre (alias) de la combinacion de algormitmo de HASH y del algoritmo asimétrico a utilizar. Por ejemplo, *MD2withRSA, MD5withRSA o SHA1withRSA*. El nombre del algoritmo debe especificarse, ya que no hay ningún valor predeterminado.

Los métodos más importantes de la clase Signature son:

METODO	TIPO DE RETORNO	DESCRIPCIÓN
getInstance (String algorithm)	static Signature	Método estático para crear objetos de clase Signature. Recibe como parámetro el nombre del algoritmo de firma digital que se desea emplear
initSing (PrivateKey privateKey)	Void	Inicializa el objeto para la operación de firmado. Se le debe pasar como argumento la clave privada
initVerify (PublicKey publicKey)	Void	Inicializa el objeto para la operación de verificación de firma. Se le debe pasar como argumento la clave pública
update (byte[] input)	Void	Actualiza el contenido del objeto, incluyendo la información pasada como parámetro. Si este método se invoca varias veces va acumulando toda la información suministrada
sign()	byte[]	Firma los datos contenidos en el objeto, usando la clave privada con la que ha sido inicializado. Devuelve la firma resultante. Al finalizar este método el objeto queda reiniciado
verify (byte[] signature)	Boolean	Comprueba si la firma pasada como parámetro verifica la integridad de los datos contenidos en el objeto, usando la clave pública con la que ha sido inicializado. Devuelve verdadero o falso, indicando el resultado de la verificación. Al finalizar este método el objeto queda reiniciado

En el siguiente ejemplo vamos a implementar el firmado digital mediante el algoritmo DSA.

Los pasos seguidos para realizar la firma de un mensaje y después verificarla son los siguientes:

- Generar las claves públicas y privadas mediante la clase KeyPairGenerator:
    - La PrivateKey la utilizaremos para firmar.
    - La PublicKey la utilizaremos para verificar la firma.
- Realizar la firma digital mediante la clase Signature y un algoritmo asimétrico, por ejemplo DSA.
    - Crearemos un objeto Signature.
    - Al método initSign() le pasamos la clave privada.
    - El método update() creará el resumen de mensaje.
    - El método sign() devolverá la firma digital.
- Verificar la firma creada mediante la clave pública generada.
    - Al método initVerify() le pasaremos la clave pública.

- Con update() se actualiza el resumen de mensaje para comprobar si coincide con el enviado.
- El método verify() realizará la verificación de la firma.

```
public static void main(String[] args) {
   try {
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
      //SE CREA EL PAR DE CLAVES PRIVADA Y PÚBLICA
      KeyPair par = keyGen.generateKeyPair();
      PrivateKey clavepriv = par.getPrivate();
      PublicKey clavepub = par.getPublic();
      //FIRMA CON CLAVE PRIVADA EL MENSAJE
      //AL  OBJETO Signature SE LE SUMINISTRAN LOS DATOS A FIRMAR
      Signature dsa = Signature.getInstance("SHA256withDSA");
      dsa.initSign(clavepriv);
      String mensaje = "Este mensaje va a ser firmado";
      dsa.update(mensaje.getBytes());

      byte[] firma = dsa.sign(); //MENSAJE FIRMADO
      //EL RECEPTOR DEL MENSAJE
      //VERIFICA CON LA CLAVE PUBLICA EL MENSAJE FIRMADO
      //AL OBJETO signature sE LE suministra los datos a verificar
      Signature verificadsa = Signature.getInstance("SHA256withDSA");
      verificadsa.initVerify(clavepub);
      verificadsa.update(mensaje.getBytes());
      boolean check = verificadsa.verify(firma);
      if (check) {
         System.out.println("FiRMA VERIFICADA CON CLAVE PÚBLICA.");
      } else {
         System.out.println("FiRMA NO VERIFICADA");
      }

       } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
       } catch (SignatureException e) {
            throw new RuntimeException(e);
       } catch (InvalidKeyException e) {
            throw new RuntimeException(e);
   }
```

}

7. Certificados Digitales

Un certificado digital es como una identidad digital para personas, sistemas o servicios.

Permite a los usuarios y servicios identificarse, estando confirmado por una autoridad de certificación.

Se basan en criptografía de clave pública (RSA, ECC, etc.) y son emitidos por una Autoridad de Certificación (CA) confiable.

Un certificado electrónico sirve, por tanto, para firmar digitalmente, para garantizar la integridad de los datos trasmitidos y su procedencia, y autenticar la identidad del usuario de forma electrónica ante terceros, además de cifrar datos para que solo el destinatario del documento pueda acceder a su contenido.

La estructura de un certificado digital contiene:

- Identidad del propietario: nombre, correo, organización, etc.
- Clave pública del propietario.
- Datos de la CA emisora: quién lo emitió, validez.
- Fechas de validez: inicio y fin.
- Firma digital de la CA: garantiza que el certificado es auténtico y no ha sido modificado.

Escenario de comunicación segura

1. Emisor (cliente) quiere enviar un mensaje firmado a un receptor (servidor).
2. El emisor posee un certificado digital con su clave pública y la firma de la CA.
3. El receptor recibe:
    - El mensaje
    - La firma digital
    - El certificado del emisor
4. El receptor verifica:
    - La firma de la CA sobre el certificado → asegura que el certificado es válido y no ha sido falsificado.
    - La firma del mensaje usando la clave pública del certificado → asegura que el mensaje realmente vino del emisor y no fue modificado.

Diferencia entre clave pública y certificado

- Una clave pública sola solo permite verificar firmas o cifrar mensajes.(La foto del DNI)

- Un certificado añade información de identidad y la garantía de una CA: no solo es la clave, sino que alguien de confianza dice "esta clave pertenece a X".(El DNI completo, foto mas validación de autoridad competente)

7.1 Herramienta keytool

- Es una utilidad incluida con Java JDK.
- Sirve para gestionar claves y certificados.
- Permite crear keystores, generar pares de claves y certificados auto-firmados, importar y exportar certificados.

7.1.2. Como usar keytool:

- Crear un keystore y generar un certificado auto-firmado

  Supongamos que queremos un certificado para un servidor SSL local.

  keytool -genkeypair -alias servidor -keyalg RSA -keysize 2048 \

  -keystore AlmacenSSL.jks -validity 365

  Explicación de las opciones:

  - -genkeypair → genera un par de claves (pública y privada).
  - -alias servidor → nombre que identifica esta clave dentro del keystore.
  - -keyalg RSA → algoritmo de clave pública (RSA en este caso).
  - -keysize 2048 → tamaño de la clave en bits (mínimo recomendado 2048).
  - -keystore AlmacenSSL.jks → archivo donde se guarda la clave y el certificado.
  - -validity 365 → validez del certificado en días.

  Al ejecutar el comando, keytool te pedirá:

  1. Contraseña del keystore
  2. Información del propietario (nombre, organización, país…)
  3. Contraseña de la clave (puede ser la misma que la del keystore)

     Esto genera un certificado auto-firmado, útil para pruebas.

- Ver el contenido del keystore

  Para revisar qué contiene tu keystore:

  keytool -list -v -keystore AlmacenSSL.jks

  - -list → lista el contenido

- -v → modo detallado

- Te mostrará el alias, el certificado, fecha de validez y huella digital.
- Exportar certificado para compartir

  Si quieres que otros clientes confíen en tu certificado:

  keytool -exportcert -alias servidor -keystore AlmacenSSL.jks -file certificado.cer

  - certificado.cer → contiene la clave pública y los datos del certificado, pero no la clave privada.

  Este archivo se puede importar en otro truststore de Java para confiar en tu servidor.
- Importar un certificado en un truststore

  Para que un cliente confíe en un servidor:

  keytool -importcert -alias servidor -file certificado.cer -keystore truststore.jks

  Esto agrega el certificado al truststore del cliente.

  El cliente puede verificar la autenticidad del servidor durante una conexión SSL.

- Resumen del flujo típico

  1. Servidor: genera un keystore con su clave privada y certificado.
  2. Servidor: utiliza javax.net.ssl.keyStore y keyStorePassword en Java para habilitar SSL.
  3. Cliente: importa el certificado público en su truststore y puede validar la conexión SSL.

Servicios en red seguros: protocolos SSL, TLS y SSH

Cada vez que accedemos a una página cuya URL empieza por HTTPS, estamos utilizando un protocolo de criptografía para acceder a los contenidos de dichas páginas de forma confidencial.

En Internet la mayoría de estas comunicaciones hace uso de los protocolos SSL (*Secure Sockets Layer*) y su sucesor TLS (*Transport Layer Security*).

SSL y TLS ofrecen una interfaz de programación basada en *sockets stream*, muy similar a la vista anteriormente.

SSL y TLS agregan las siguientes características de seguridad:

- Uso de criptografía asimétrica para el intercambio de claves de sesión.

- Uso de criptografía simétrica para asegurar la confidencialidad de la sesión.
- Uso de códigos de autenticación de mensajes (resúmenes) para garantizar la integridad de los mensajes.

Pasos en una comunicación TLS:

- El cliente y el servidor negocian la versión de TLS y los algoritmos de cifrado.
- El servidor envía su certificado (emitido por una autoridad de confianza).
- El cliente valida ese certificado.
- Ambos generan claves de sesión temporales.
- Toda la comunicación siguiente viaja cifrada.

SSL y TLS garantizan el establecimiento y mantenimiento de una sesión segura.

La identidad del elemento servidor está garantizada (autenticación). Esto se consigue gracias a un certificado de servidor y al cifrado asimétrico

La privacidad de las comunicaciones está garantizada (confidencialidad). Esto se consigue gracias a una clave de sesión, simétrica.

Los mensajes que se intercambian no pueden ser alterados de ninguna manera (integridad). Esto se consigue mediante cálculo de resúmenes.

8. COMUNICACIONES SEGURAS CON JAVA. JSSE

JSSE (Java Secure Socket Extension) es un conjunto de paquetes que permiten el desarrollo de aplicaciones seguras en Internet.
Proporciona un marco y una implementación para Java de los protocolos SSL y TSL e incluye funcionalidad de encriptación de datos autenticación de servidores integridad de mensajes autenticación de clientes.
Con JSSE, los programadores pueden ofrecer intercambio seguro de datos entre un cliente y un servidor que ejecuta un protocolo de aplicación, tales como HTTP, Telnet o FTP, a través de TCP/IP.
Las clases de JSSE se encuentran en los paquetes javax.net y javax.net.ssl.

SSLSocket y SSLServerSocket

Las clases SSLSocket y SSLServerSocket representan sockets seguros y son derivadas de las ya conocidas Socket y ServerSocket respectivamente.
JSSE tiene dos clases SSLServerSocketFactory y SSLSocketFactory para la creación de sockets seguros. No tienen constructor, se obtienen a través del método estático getDefault().
Para obtener un socket servidor seguro o *SSLServerSocket*:

```
SSLServerSocketFactory sfact = (SSLServerSocketFactory)
SSLServerSocketFactory.getDefault();
```

```
SSLServerSocket servidorSSL = (SSLServerSocket) = sfact.createServerSocket(puerto);
```

El método createServerSocket(int puerto) devuelve un socket de servidor enlazado al puerto especificado.

Para crear un *SSLSocket*:

```
SSLSocketFactory sfact = (SSLSocketFactory) SSLSocketFactory.getDefault();
SSLSocket Cliente = (SSLSocket) sfact.createSocket(Host, puerto);
```

El método createSocket (String host, int puerto) crea un socket y lo conecta con el host y el puerto especificados.

Cuando dos socket SSL intentan establecer conexión (un socket cliente y un socket servidor), lo primero que hacen es "presentarse" el uno al otro y cada uno de ellos comprueba que el otro es de "confianza". Si todo va bien, la conexión se establece. Si uno no confía en el otro, la conexión no se establece.
Para establecer esa confianza se debe crear un certificado en el servidor y añadirlo a los certificados de confianza del cliente.

- El servidor debe tener su propio certificado. Si no lo tenemos, se puede generar primero una pareja de claves con la herramienta keytool, que viene incluida en el JDK de Java. La herramienta guardará la pareja de claves en un almacén (el cual tiene su propia clave).
- Después generaremos un certificado a partir de esa pareja.
- El código del servidor necesitará indicar el fichero donde se almacenan las claves y la clave para acceder a ese almacén.
- El cliente necesitará indicar que confía en el certificado del servidor. Dicho certificado del servidor puede estar guardado (por ejemplo) en el almacén de claves del cliente.

Vamos a ver cómo realizar estas operaciones previas con la herramienta keytool
Primero las acciones a realizar en el servidor:

El servidor genera una pareja de claves que almacena en un fichero llamado "almacenSSL.jks" Dentro del fichero se indica una alias para poder referirnos a esa clave facilmente.

```
keytool -genkey -keyalg RSA -alias claveSSL -keystore "C:\AlmacenSSL.jks" -storepass 1234567
```

keytool está en el directorio bin de donde tengamos instalado java.
Con la opción **-genkey** le estamos diciendo que genere un certificado.
-keyalg RSA le indicamos que lo queremos encriptado con el algorítmo RSA
-alias claveSSL. El certificado se meterá en un fichero de almacén de certificados que podrá contener varios certificados. Este alias es el nombre con el que luego podremos identificar este certificado concreto dentro del almacén. Podemos poner cualquier nombre que nos de una pista de qué es ese certificado.
-keystore AlmacenSSL.jks. Este es el fichero que hará de almacén de certificados. Si no existe se crea, si ya existe se añade el certificado con el alias que se haya indicado.
-storepass 1234567. El almacén está protegido con contraseña, para acceder a él necesitamos la contraseña. Si el almacén no existe, se crea usando esta contraseña, por lo que

deberemos recordarla. Si ya existe, debemos proporcionar la contraseña que tuviera ese almacén.

```
C:\Users\usuario1>keytool -genkey -alias claveSSL -keyalg RSA -keystore AlmacenS
SL -storepass 1234567
¿Cußles son su nombre y su apellido?
  [Unknown]:  Eider
¿Cußl es el nombre de su unidad de organizaci¾n?
  [Unknown]:  PSP
¿Cußl es el nombre de su organizaci¾n?
  [Unknown]:  arriaga
¿Cußl es el nombre de su ciudad o localidad?
  [Unknown]:  Gasteiz
¿Cußl es el nombre de su estado o provincia?
  [Unknown]:  Alava
¿Cußl es el c¾digo de paýs de dos letras de la unidad?
  [Unknown]:  ES
¿Es correcto CN=Eider, OU=PSP, O=arriaga, L=Gasteiz, ST=Alava, C=ES?
  [no]:  si

Introduzca la contrase±a de clave para <claveSSL>
        (INTRO si es la misma contrase±a que la del almacún de claves):
```

El servidor genera su "certificado", es decir un fichero que de alguna forma indica quien es él. El certificado se almacena en un fichero llamado AlmacenSSL.jks y a partir de él queremos generar el certificado de un alias creado previamente con nombre certiifcado.cer.

```
keytool -export -keystore AlmacenSSL.jks -alias claveSSL -file Certificado.cer
```

-export es para exportar el certificado
-keystore AlmacenSSL.jks indica en qué almacén está el certificado que queremos exportar
-alias claveSSL es el identificador del certificado dentro del almacén. Debe ser el mismo alias que pusimos cuando lo creamos.
-file Certificado.cer es el nombre del fichero donde queremos que se guarde el certificado que vamos a extraer.

Una vez que tenemos el fichero exportado es necesario incorporarle al nuevo almacenamiento para permitir realizar la validación. Se importa el certificado del servidor indicando que pertenece a la lista de certificados confiables. A continuación se crea un keystore de nombre UsuarioAlmacenSSL con la clave 890123 y se incorpora el fichero de certificado Certificado.cer.

```
keytool -import -alias claveSSL -file Certificado.cer -keystore UsuarioAlmacenSSL -storepass 890123
```

-import para indicar que queremos meter un certificado existente en un almacén.
-alias claveSSL es el identificador que queremos dar al servidor dentro del almacén de certificados de confianza del cliente. Hemos puesto otra vez claveSSL, pero al ser un almacén distinto de el del servidor, podríamos poner otro nombre.
-file Certificado.cer El certificado que queremos importar
-keystore UsuarioAlmacenSSL el almacén de certificados de confianza del cliente, se creará si no existe.

-storepass 890123 Clave para el almacén, si el almacén existe debe ser la que diéramos en el momento de crearlo. Si no existe, el almacén se creará protegido con esta clave.

Ahora debemos añadir unas propiedades del JSSE en nuestro código.

Las propiedades son:
javax.net.ssl.keyStore: Ubicación del fichero de almacén de claves de Java.
javax.net.ssl.keyStorePassword: Contraseña para acceder a la clave privada en el fichero de almacén de claves especificado por javax.net.ssl.keyStore.
javax. net.ssl.trustStore: Ubicación del fichero de almacén de claves que contiene la colección de certificados de confianza.
Javax.net.ssl.keyStorePassword: Contraseña para abrir el fichero de almacén de claves especificado por javax. net.ssl.trustStore.

En el servidor:
System.setProperty("javax.net.ssl.keyStore","AlmacenSSL.jks");
System.setProperty("javax.net.ssl.keyStorePassword","1234567");

Antes de ejecutar el programa cliente necesitamos colocar el certificado en el keystore del usuario, para ello lo exportamos a un fichero, le llamamos Certificado.cer:

Tendriamos que añadir en el cliente:
System.setProperty("javax.net.ssl.trustStore","UsuarioAlmacenSSL");
System.setProperty("javax.net.ssl.trustStorePassword","890123");

A continuación se muestra el programa servidor que crea una conexión sobre un socket servidor seguro y que atenderá hasta 4 conexiones de clientes que se identificarán con un certificado válido. El servidor espera las conexiones, de cada cliente que se conecta recibe un mensaje y a continuación le envía un saludo. El código es el siguiente:

En el lado del servidor:

```
int puerto = 6000;

System.setProperty("javax.net.ssl.keyStore","C:\\Users\\HEZITZAILE\\
Desktop\\SimetricoSSL\\AlmacenSSL.jks");

 System.setProperty("javax.net.ssl.keyStorePassword","123456");


      SSLServerSocketFactory sfact = (SSLServerSocketFactory)
SSLServerSocketFactory.getDefault();
```

```
        SSLServerSocket servidorSSL = null;
        try {
           servidorSSL = (SSLServerSocket) sfact.createServerSocket(puerto);
        } catch (IOException e) {
           throw new RuntimeException(e);
        }
        SSLSocket clienteConectado = null;
                //FLUJO de entrada de cliente
        DataInputStream flujoEntrada = null;
        System.out.println("Esperando al cliente" + i);

//Flujo de salida al cliente
DataOutputStream flujoSalida = null;

    for (int i = 1; i < 5; i++) {
        try {
         clienteConectado = (SSLSocket) servidorSSL.accept();
         flujoEntrada = new
         DataInputStream(clienteConectado.getInputStream());
            //EL cliente envia un mensaje
        System.out.println("Recibien del cliente" + i + "\n\t" +
         flujoEntrada.readUTF());
        flujoSalida = new     DataOutputStream(clienteConectado.getOutputStream());

          } catch (IOException e) {
             throw new RuntimeException(e);
          }

//envio un saludo al cliente
          try {
             flujoSalida.writeUTF("HOla cliente desde el servidor");
             //cerrar streams y sockets
             flujoEntrada.close();
             flujoSalida.close();
             clienteConectado.close();
             servidorSSL.close();
          } catch (IOException e) {
             throw new RuntimeException(e);
          }
       }
```

En el lado del cliente:

```
String host = "localhost";
int puerto = 6000;

System.out.println("Programa cliente iniciado..");
System.setProperty("javax.net.ssl.trustStore", "C:\\Users\\HEZITZAILE\\
Desktop\\SimetricoSSL\\UsuarioAlmacenSSL");

System.setProperty("javax.net.ssl.trustStorePassword", "123456");


SSLSocketFactory sfact = (SSLSocketFactory) SSLSocketFactory.getDefault();
SSLSocket Cliente = null;
try {
   Cliente = (SSLSocket) sfact.createSocket(host, puerto);
   //Creo flujo de salida
   DataOutputStream flujoSalida = new DataOutputStream(Cliente.getOutputStream());
   //envio texto al servidor
   flujoSalida.writeUTF("texto del cliente al server");
   //creo flujo de entrada del server
   DataInputStream flujoEntrada = new DataInputStream(Cliente.getInputStream());
   //Leo la informacion del server
   System.out.println("Recibiendo del server \t\n" + flujoEntrada.readUTF());
   //cerrar
   flujoEntrada.close();
   flujoSalida.close();
   Cliente.close();

} catch (IOException e) {
   throw new RuntimeException(e);
}
```

9. Control de acceso con Java. JAAS

JAAS (servicio de autorización y autenticación de Java) es una interfaz que permite a las aplicaciones Java acceder a servicios de control de autenticación y acceso. Se puede utilizar para dos propósitos:
para la autenticación, de usuarios: para determinar de forma fiable y segura quién está ejecutando nuestro código Java, y para la autorización de los usuarios: para asegurarse de que quien lo ejecuta tiene los permisos necesarios para realizar las acciones.

En el proceso de autenticación y autorización mediante JAAS están involucradas las siguientes clases e interfaces:

- **LoginContext**, contexto de inicio de sesión: inicia y gestiona el proceso de autenticación mediante la creación de un Subject. La autenticación se hace llamando al método login().
- **LoginModule**, módulo de conexión: es la interfaz que debe implementarse para definir los mecanismos de autenticación en la aplicación. Se deben implementar los siguientes métodos: initialize(), login(), comit(), abort() y logout(). Se encarga de validar los datos en un proceso de autenticación.
- **Subject,** clase que representa a un ente autenticable dentro de la aplicación (entidad, usuario, sistema).
- **Principal**, clase que representa los atributos que posee cada Subject recuperado una vez que se efectúa el ingreso a la aplicación. Un Subject puede contener varios principales.
- **CallBackHandler,** interfaz que se debe implementar cuando se necesita recibir del usuario la información para la autenticación, se encarga de la interacción con el usuario para obtener los datos de autenticación. Al implantarla se debe desarrollar el método handle().

Los paquetes en los que están disponibles las clases e interfaces principales de JAAS son

- javax.security.auth.*: contiene las clases de base e interfaces para los mecanismos de autenticacion y autorización.
- Javax.security.auth.callback.*: contiene las clases e interfaces para definir credenciales de autenticacion de la aplicación.
- Javax.security.auth.login.*: contiene las clases para entrar y salir de un dominio de aplicación.
- Javax.security.auth.spi.*: contiene interfaces para un proveedor de JAAS para implementar módulos JAAS.

9.1 AUTENTICACIÓN

El proceso básico de autenticación con JAAS consta de los siguientes pasos:

- Creación de una instancia de LoginContext, uno o más LoginModule son cargados basándose en el archivo de configuración de JAAS.
- La instalación de cada LoginModule es opcionalmente provista con un CallbackHandler que gestionará el proceso de continuación con el usuario para obtener los datos con los que este tratará de autenticarse.
- Invocación del método login() del LoginContext el cual invocará el método login() del LoginModule.
- Los datos del usuario se obtienen por medio del CallbackHandler.
- El LoginModule comprueba los datos introducidos por el usuario y los valida. Si la validación tiene éxito el usuario queda autenticado.

9.2 AUTORIZACIÓN

La autorización de JAAS extiende la arquitectura de seguridad de Java centrada en el código y se basa en el uso de políticas de seguridad para especificar cuáles son los permisos de control de acceso que se concederán para ejecutar un código. Los permisos se otorgarán no solo en función de qué código se está ejecutando, sino también en quién lo está ejecutando.
Cuando una aplicación utiliza la autenticación de JAAS para autenticar al usuario (u otra entidad, como un servicio), se crea un Subject como resultado que representara al usuario autenticado. Un Subject se compone de un conjunto de principales (clase Principal), donde cada principal representa un atributo para ese usuario. Por ejemplo un Subject puede tener dos principales, uno es el nombre y el otro el DNI.
Para que la autorización JAAS tenga lugar, se requiere lo siguiente:

- El usuario debe autenticarse, ya hemos visto como se hace.
- En el fichero de políticas se deben configurar entradas para los principales.
- Se debe asociar al Subject el contexto de control de acceso actual usando los métodos doAs() o doAsPrivileged() de la clase Subject.

Servicio de Autenticación y Autorización de Java (JAAS) - Definición y explicación (techlib.net)