MÓDULO PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TEMA 5

UTILIZACIÓN DE TÉCNICAS DE PROGRAMACIÓN SEGURA (II)

CICLO DE GRADO SUPERIOR INFÓRMATICA

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Autor: Luis Miguel Lestón López a partir de los materiales generados por el Ministerio de Educación para este módulo. Obra derivada.

1.- Introducción

Cuando desarrollamos aplicaciones con comunicaciones, por ejemplo a través de Internet, debemos proporcionar seguridad tanto a la aplicación como a los datos transmitidos, ya que las operaciones que se realizan a través de la red podrían ser interceptadas, y por tanto manipuladas por personas desautorizadas.

La **seguridad de una aplicación y de los datos transmitidos** dependerá de su diseño, la selección de protocolos de comunicación y el método de autentificar al usuario.

Los **objetivos de seguridad** que debe proporcionar una aplicación con comunicaciones seguras son los siguientes:

- **Confidencialidad**. Consiste en garantizar que los datos transmitidos por la aplicación sólo van a estar disponibles para las entidades (personas o procesos) autorizadas a acceder a dicha información. Si esos datos son interceptados por entidades desautorizadas, éstas no podrán acceder a la interpretación de esa información.
- **Integridad**. Consiste en garantizar que los datos originales no han sido modificados o alterados por alguna entidad no autorizada durante su transmisión.
- **Autenticación**. Consiste en asegurar que la entidad emisora es quien dice ser.
- **No repudio**. Consiste en asegurar que las acciones de un usuario han sido realizadas exactamente por dicho usuario. El no repudio garantiza la participación de las partes en una comunicación. En toda comunicación existe un emisor y un receptor, por lo que se pueden distinguir dos tipos de repudio:
    - **No repudio en origen**: garantiza que la persona que envía el mensaje no puede negar que es el emisor del mismo, ya que el receptor tendrá pruebas del envío.
    - **No repudio en destino**: el receptor no puede negar que recibió el mensaje, porque el emisor tiene pruebas de la recepción del mismo.
- **Autorización**. Trata de asegurar que una entidad puede realizar una acción en concreto, esto es, la autorización valida las acciones del usuario para verificar si tiene privilegios para realizar dicha acción.

Veamos los 4 primeros objetivos con un **ejemplo sencillo**:

- Cuando Antonio le envía información a Isabel, debe asegurarse de que esa información no es alterada o manipulada por el camino (Integridad de la información transmitida).
- La información es para Isabel, por tanto nadie más debe entender el mensaje (Confidencialidad).
- Debe haber alguna indicación de que el mensaje proviene de Antonio, esto es, debe haber alguna prueba de ello (Autenticación). La autenticación de que el mensaje proviene de Antonio es proporcionado por lo que se denomina firma digital.

- ¿Quién garantiza la identidad y la firma de Antonio? Esto se logra mediante un certificado digital, que autentica la firma de Antonio.
- Además, es igualmente importante garantizar que Antonio no pueda negar la autoría del envío del mensaje, indicando que alguien lo envío en su nombre. (No repudio). Este objetivo también se logra con la firma digital.

En esta unidad veremos la forma de conseguir estos objetivos de seguridad utilizando **criptografía**, uno de los pilares básicos sobre los que descansan la mayoría de las soluciones de seguridad.

2.- Criptografía

¿Qué es la criptografía? El término **criptografía** viene del griego "cripto" (secreto) y "grafía" (escritura), por lo que su significado es "escritura secreta". La finalidad de la criptografía es enmascarar o codificar una información original, utilizando alguna técnica, de manera que el resultado sea ininteligible para las personas no autorizadas.

La criptografía se ha usado a lo largo de los años para mandar mensajes confidenciales o privados. Básicamente, cuando alguien quiere mandar información confidencial de forma secreta, actúa de la siguiente manera:

- Aplica técnicas criptográficas para poder "enmascarar" el mensaje.
- Manda el mensaje "enmascarado" por una línea de comunicación que se supone insegura y puede ser interceptada.
- Solo el receptor autorizado podrá leer el mensaje "enmascarado".

Otra disciplina relacionada con la criptografía es el **criptoanálisis**, que analiza la robustez de los sistemas criptográficos y comprueba si realmente son seguros. Para ello, se intenta romper la seguridad que proporciona la criptografía, deshaciendo el sistema y accediendo de esta forma a la información secreta en su formato original. Esta disciplina es una herramienta muy poderosa que permite mejorar los sistemas de criptografía constantemente y ayuda a desarrollar otros nuevos más efectivos.

Por último, y para terminar de recorrer estas disciplinas, debes conocer el término **criptología**, que es la ciencia que engloba tanto las técnicas de criptografía como las de criptoanálisis.

2.1.- Encriptación de la información

La **encriptación o cifrado de la información** es el proceso por el cual la información o los datos a proteger son traducidos o codificados como algo que parece aleatorio y que no tiene ningún significado (datos encriptados).

La **desencriptación o descifrado** es el proceso inverso, en el cual los datos encriptados son convertidos nuevamente a su forma original.

A continuación, te indicamos los conceptos o términos asociados con la encriptación:

- **Texto llano o claro**: es la información original, la que no está cifrada o encriptada.
- **Criptograma o texto cifrado**: es la información obtenida tras la encriptación.

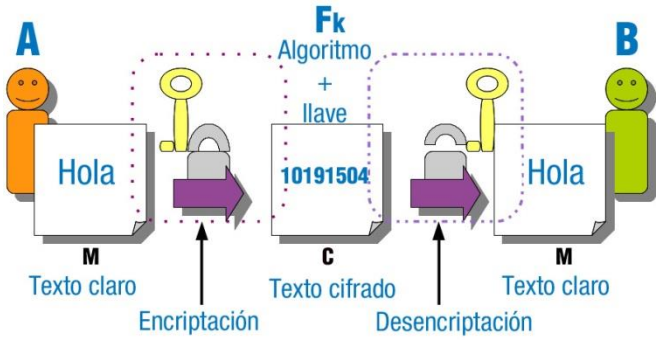
- **Algoritmo criptográfico o algoritmo de cifrado**: es un conjunto de pasos u operaciones, normalmente una función matemática, usado en los procesos de encriptación y desencriptación. Asociado al algoritmo hay una **clave o llave** (un número, palabra, frase, o contraseña).
- La **clave** controla las operaciones del algoritmo dentro del proceso de cifrado y descifrado, de manera que usando el mismo algoritmo con llaves diferentes, se obtienen textos cifrados o criptogramas diferentes.
- La clave puede ser simétrica (misma llave para cifrar y descifrar) o asimétrica (llaves diferentes para cifrar y descifrar).

Resumiendo, podemos representar este proceso de la siguiente forma:

$$\text{Cifrado} \rightarrow F_K(M)=C$$

Lo que indica que el cifrado consiste en aplicar un algoritmo F, controlado por una llave K, sobre un mensaje o texto claro **M**, para obtener un criptograma o texto cifrado **C**. El descifrado, será el proceso inverso.

El siguiente esquema, muestra un ejemplo muy sencillo de encriptación con clave simétrica, donde **el algoritmo se representa mediante un candado y la clave mediante una llave**.



En este ejemplo podrían ser:

- Algoritmo: posición de la letra en el alfabeto representada con dos dígitos.
- Clave: 3.
- Algoritmo Cifrado: posición de la letra en alfabeto (dos dígitos) + 3.
- Descifrado: posición de la letra en alfabeto (dos dígitos) - 3.

Así tendríamos que: H: 7 + 3 → **10**, o: 16+3 → **19**, l: 12+3 → **15**, a: 1+3 → **04**.

2.2.- Principios criptográficos

Las propiedades deseables de un sistema criptográfico las podemos resumir en los siguientes principios, que fueron enunciados por Auguste Kerckhoffs en 1883 y que dicen:

- Si el sistema no es teóricamente irrompible, al menos debe serlo en la práctica.
- La efectividad del sistema no debe depender de que su diseño permanezca en secreto.
- La clave debe ser fácilmente memorizable de manera que no haya que recurrir a notas escritas.
- Los criptogramas deberán dar resultados alfanuméricos.
- El sistema debe ser operable por una única persona.
- El sistema debe ser fácil de utilizar.

El segundo principio recibe el nombre de **principio de Kerckhoffs** y establece que **la seguridad o fortaleza del sistema criptográfico debe depender exclusivamente de mantener en secreto la clave y no de ocultar el diseño del sistema**, que puede ser público y conocido. De hecho, en la actualidad, la mayoría de los algoritmos utilizados en criptografía son de dominio público.

Ese principio fue reformulado posteriormente por Claude Shannon de la siguiente forma:

- **El adversario o enemigo conoce el sistema.**

A esta última formulación se la conoce como la máxima de **Shannon** y es el principio más adoptado por los criptólogos en oposición a la llamada seguridad a través de la oscuridad, basada en la ocultación del diseño e implementación del sistema criptográfico a los usuarios.

Entonces, ¿de qué depende realmente la **seguridad de un sistema criptográfico**? Depende de dos factores:

- El **diseño o robustez del algoritmo**. Cuanto menos fallos tenga el algoritmo, más seguro será el sistema.
- La **longitud de la clave utilizada**. A mayor longitud de la clave, mayor seguridad proporcionará el sistema. Actualmente, esta longitud se estima que debe ser como mínimo de 128 bits, existiendo algoritmos que permiten seleccionar el tamaño de la clave.

En la actualidad, se utilizan fundamentalmente dos métodos criptográficos: el conocido como **cifrado simétrico o de clave privada** y el conocido como **cifrado asimétrico o de clave pública**. Veremos estos métodos en los siguientes apartados.

2.3.- Criptografía de clave privada o simétrica

Este método de encriptación utiliza una clave secreta, conocida solo por emisor y receptor, para encriptar la información. Se la denomina también criptografía simétrica porque **la clave utilizada para la encriptación y desencriptación es la misma**. Es adecuada para garantizar confidencialidad.

A continuación te indicamos los principales aspectos de la **criptografía de clave privada o simétrica**:

- **Características**:
  - La clave es privada o secreta, solo la conocen las partes involucradas.
  - Se utiliza la misma clave para el cifrado y descifrado. Así, según el gráfico de arriba, cuando A le envía información a B, utiliza la clave privada para cifrar el mensaje, que solo podrá descifrar B si también conoce la clave. Por tanto, A debe hacer llegar a B la clave privada.
- **Ventaja**:
  - Son algoritmos muy rápidos y que no aumentan el tamaño del mensaje; por tanto adecuados para cifrar grandes volúmenes de datos.
- **Inconvenientes**:
  - El problema de la distribución de claves: el receptor debe conocer la clave que se va a utilizar, lo que implica que el emisor se la debe enviar y no es posible utilizar medios inseguros para el intercambio de claves.
  - Otro inconveniente es el gran número de claves que se necesitaría si el grupo de personas que puede comunicarse de forma privada es muy grande, pues se necesitaría una clave diferente cada dos personas del grupo.
- Algunos **ejemplos de algoritmos** de este tipo son: AES o Rijndael (Nuevo estándar mundial), DES, 3-DES, DES-X, IDEA y RC5.

2.4.- Criptografía de clave pública o asimétrica

La criptografía de clave pública surge para solucionar el problema de distribución de claves que plantea la criptografía de clave privada, permitiendo que emisor y receptor puedan acordar una clave en común sobre canales inseguros. Se la denomina también criptografía asimétrica porque **las claves utilizadas para la encriptación y desencriptación son diferentes**. Es adecuada para garantizar además de confidencialidad, la autenticación de los mensajes y el no repudio.

A continuación, te indicamos los aspectos más destacados de la **criptografía de clave pública o asimétrica**.

- **Características**:
  - Cada parte posee una pareja de claves, una pública, conocida por todos, y su inversa privada, conocida solo por su poseedor.
  - Cada pareja de claves, son complementarias: lo que cifra una de ellas, solo puede ser descifrado por su inversa.
  - Esa pareja de claves sólo se puede generar una vez, de modo que se puede asumir que no es posible que dos personas hayan obtenido casualmente la misma pareja de claves.
  - Al cifrar un mensaje con la clave pública, tan solo podrá descifrarlo quien posea la clave privada inversa a esa clave pública. Así, según el gráfico de arriba, cuando A le envía información a B, utiliza la clave pública de B para cifrar el mensaje, que solo podrá descifrar B con su clave privada.
  - Conocer la clave pública no permite obtener ninguna información sobre la clave privada, ni descifrar el texto que con ella se ha cifrado.
  - Cifrar un mensaje con la clave privada equivale a demostrar la autoría del mensaje, autenticación, nadie más ha podido cifrarlo utilizando esa clave privada. El mensaje cifrado con una clave privada podrá descifrarlo todo aquel que conozca la clave pública inversa. Por tanto, si A enviase un mensaje cifrado con su clave privada, podrá descifrarlo todo el que conozca la clave pública de A y además sabrá que el mensaje proviene de A.
- **Ventaja**:
  - No existe el problema de distribución de claves, ya que cada parte posee su juego de claves.
- **Inconvenientes**:
  - Son más lentos que los algoritmos simétricos.
  - Garantizar que la clave pública es realmente de quien dice ser su poseedor, lo que se conoce como el problema del 'ataque del hombre en el medio' o man in the middle. (Es un ataque en el que el atacante es capaz de observar e interceptar mensajes entre las dos partes sin que ninguna de ellas sepa que el enlace entre ellos ha sido violado).
- Algunos ejemplos de algoritmos de este tipo son: DSA, RSA(estándar de facto), algoritmo de Diffie-Hellman.

Generalmente se utiliza una combinación de ambas: criptografía asimétrica para negociar una clave privada con la que después se comunicarán los datos.

2.5.- Resumen de mensajes, firma digital y certificados digitales

Otras técnicas criptográficas son las siguientes:

- **Resumen de mensajes, basado en funciones HASH**. Un algoritmo de resumen toma como entrada un mensaje de longitud variable y lo convierte en un resumen de longitud fija, cuyas principales características son: siempre debe proporcionar la misma salida, debe ser aleatorio y unidireccional.
    - El principal uso que tienen es proporcionar integridad.
    - Combinado con técnicas de clave pública se obtiene una forma eficiente de proporcionar identificación.
    - Los algoritmos HASH o de resumen más usados son MD5 y SHA.
- **Firmas digitales**. Son el equivalente digital de las firmas personales. Se basan en criptografía de clave pública y resumen de mensajes o funciones HASH.
    - Para la transmisión de un mensaje entre un emisor y un receptor, el emisor transmitirá, junto con el texto deseado, la firma digital del mensaje cuya finalidad es comprobar la integridad del mensaje y la autenticidad del emisor.
    - Para ello, el emisor debe disponer de una clave pública y otra privada. Cuando desee enviar un mensaje a un receptor, codificará el mensaje con una función HASH cuya salida la cifrará con su clave privada, generando así la firma digital que transmitirá al receptor junto con el texto deseado. El receptor, separará el mensaje recibido en dos partes: el texto y la firma. Usará la clave pública del emisor para descifrar la firma y, al texto que recibe le aplicará la misma función HASH que el emisor, comparando la salida de su función con el mensaje descifrado incluido en la firma. Si coinciden, quedará probada la integridad del mensaje y la autenticidad del emisor.
- **Certificados digitales**. Pretenden resolver el problema de la confianza entre las partes, delegando la responsabilidad en un tercero. Es decir, un certificado digital no es más que un mensaje firmado por una parte de una conversación, especie de notario digital que autentifica a las partes, que certifica que una clave pública pertenece a quién dice ser su poseedor. Para ello, los certificados digitales, según el estándar **X.509**, deben contener entre otra la siguiente información: número de versión, número de serie del certificado, información del algoritmo del emisor, emisor del certificado, periodo de validez, información sobre el algoritmo de clave pública, firma digital de la autoridad emisora.

Con todo esto, se introduce el nuevo concepto de **Entidad Certificadora**, que son organizaciones que se responsabilizan de la validez de los certificados, teniendo que, además de crearlos, proporcionar un mecanismo que permita su revocación, su suspensión, la búsqueda de certificados y la comprobación del estado del certificado.

2.6.- Principales aplicaciones de la criptografía

Podemos resumir las principales aplicaciones de la criptografía en las siguientes:

- **Seguridad de las comunicaciones**. La criptografía aplicada a las redes de computadores, permite establecer canales seguros sobre redes que no lo son. Además, con la potencia de cálculo actual y empleando algoritmos de cifrado simétrico (que se intercambian usando algoritmos de clave pública) se consigue la privacidad sin perder velocidad en la transferencia.
- **Identificación y autentificación**. Gracias al uso de firmas digitales y otras técnicas criptográficas es posible identificar a un individuo o validar el acceso a un recurso en un entorno de red, con más garantías que con los sistemas de usuario y clave tradicionales.
- **Certificación**. La certificación es un esquema mediante el cual agentes fiables (como una entidad certificadora) validan la identidad de agentes desconocidos (como usuarios reales). El sistema de certificación es la extensión lógica del uso de la criptografía para identificar y autentificar cuando se emplea a gran escala.
- **Comercio electrónico**. Gracias al empleo de canales seguros y a los mecanismos de identificación se posibilita el comercio electrónico, ya que tanto las empresas como los usuarios tienen garantías de que las operaciones no pueden ser espiadas, reduciéndose el riesgo de fraudes y robos.

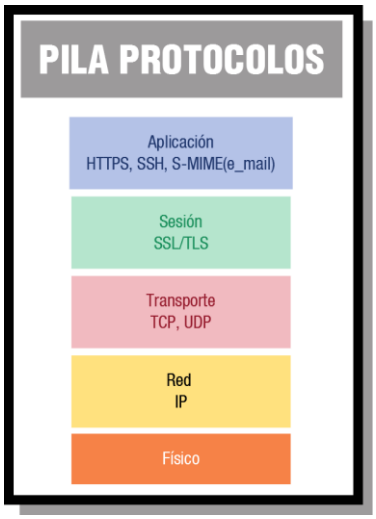
3.- PROTOCOLOS SEGUROS DE COMUNICACIONES

Si combinamos la criptografía con tecnologías de comunicación en red, entonces hablaremos de protocolos seguros de comunicaciones o protocolos criptográficos.

Algunos de estos protocolos son los siguientes:

- SSL: Proporciona comunicación segura en una conexión cliente/servidor, frente a posibles ataques en la red, como por ejemplo el problema que ya te comentamos al hablar de la criptografía asimétrica, conocido como man in the middle u "hombre en el medio".
- TLS: Es una evolución de SSL, que amplía los algoritmos criptográficos que puede utilizar.

En el siguiente esquema puedes ver el nivel, según la pila TCP/IP, en el que se utilizan estos protocolos:



Tanto SSL como TLS son protocolos criptográficos que se ejecutan en una capa intermedia entre un protocolo de aplicación, y un protocolo de transporte como TCP o UDP, por lo que pueden ser utilizados para el cifrado de protocolos de aplicación como Telnet, FTP, SMTP, IMAP o el propio HTTP.

Cuando un protocolo de aplicación, como HTTP o Telnet se ejecuta sobre un protocolo criptográfico como SSL o TLS, se habla de la versión segura de ese protocolo, por ejemplo:

- SSH: Protocolo usado exclusivamente en reemplazo de Telnet, para comunicaciones seguras.
- HTTPS: Protocolo usado exclusivamente para comunicaciones seguras de hipertexto.

Por tanto, podemos decir que el protocolo HTTPS no es ni más ni menos que el protocolo HTTP, pero ejecutándose sobre el protocolo criptográfico SSL, y por ello se dice que HTTPS es un protocolo seguro.

3.1.- Protocolo criptográfico SSL/TLS

El **protocolo SSL** (Secure Sockets Layer) fue diseñado originalmente por la empresa Netscape para **proporcionar comunicaciones seguras** entre un navegador y un servidor Web, pero se puede utilizar en cualquier otro tipo de **conexión cliente/servidor**. El protocolo **SSL proporciona los siguientes servicios de seguridad**:

- Autenticación.
- Confidencialidad.
- Integridad.

El **protocolo TLS** (Transport Layer Security) **es una evolución del protocolo SSL, una versión avanzada**, que proporciona más algoritmos criptográficos y mayor seguridad frente a nuevos ataques, pero opera de igual forma que SSL.

¿Cómo funciona el protocolo SSL?

Antes de poder garantizar una comunicación segura entre un cliente y un servidor, se deben **acordar o negociar una serie de parámetros en la comunicación en**

unas fases previas, es lo que se conoce como acuerdo o handshake (apretón de manos), el SSL/TLS Handshake Protocol. Por otra parte, el protocolo SSL/TLS Record Protocol especifica la forma de encapsular los datos transmitidos y recibidos, incluso los de negociación.

Te indicamos de manera simplificada las **fases que se siguen con SSL**.

1. Fase inicial. Negociar los algoritmos criptográficos a utilizar.
2. Fase de autenticación y clave de sesión. Intercambio de claves y autenticación mediante certificados, utilizando criptografía asimétrica. Se crea la clave para cifrar los datos transmitidos con criptografía simétrica, para agilizar las transmisiones.
3. Fase fin. Verificación del canal seguro.
4. A partir de aquí, comunicación segura.

Una vez negociados los parámetros de comunicación queda establecido el canal seguro, pero si falla la negociación, la comunicación o canal no se establece.

En su funcionamiento se pueden utilizar entre otros, los siguientes algoritmos:

- Algoritmos de cifrado simétrico: DES, 3DES, RC2, RC4, IDEA.
- Algoritmos de clave pública: RSA.
- Algoritmos de resumen: MD5, SHA.
- Certificados: DSS, RSA.
- Clave de sesión distinta en cada transacción.

Se trata por tanto de un **protocolo de cifrado híbrido**, que utiliza criptografía asimétrica en las fases de negociación y criptografía simétrica en la transmisión de datos.

El éxito del protocolo SSL/TLS se debe fundamentalmente a la expansión del comercio electrónico en Internet, pero también es utilizado para crear redes privadas virtuales VPN.

3.2.- Otros protocolos seguros

Ya te hemos indicado anteriormente que HTTPS, es la versión segura del protocolo HTTP. La S final indica que es seguro y proviene del uso del protocolo SSL. El protocolo HTTPS es utilizado por entidades bancarias, tiendas en línea y cualquier otro servicio que requiera el envío de información sensible.

La **idea principal de HTTPS es la de crear un canal seguro sobre una red insegura, utilizando cifrado basado en SSL/TLS, el puerto 443** y proporcionando protección frente ataques como el del "hombre en el medio" o man-in-the-middle, siempre que se utilicen métodos de cifrado adecuados y que el certificado del servidor sea verificado y resulte de confianza.

La confianza inherente de HTTPS está basada en una Autoridad de Certificación que viene preinstalada en el software del navegador, de manera que una conexión HTTPS a un sitio web puede ser validada sí y solo si ocurre lo siguiente:

- El usuario confía en la autoridad de certificación.
- El sitio web proporciona un certificado válido.
- El certificado identifica correctamente el sitio web.
- Cada uno de los nodos involucrados en Internet son confiables o el usuario confía en que el cifrado del protocolo SSL/TLS es inquebrantable.

Para saber si la página web que estamos visitando es segura y por tanto utiliza el protocolo HTTPS, debemos observar que en la barra de navegación a la izquierda aparece HTTPS y no HTTP, pues a veces, aunque el propio navegador indique las páginas seguras mediante la imagen de un candado amarillo a la derecha de la barra de direcciones, éste puede haber sido colocado por un atacante.

El protocolo SSH fue diseñado para dar seguridad al acceso a computadores en forma remota.

- Cumple la misma función que Telnet, pero utilizando el puerto 22.
- Utiliza criptografía, logrando así seguridad en la conexión.
- Requiere que por parte del servidor exista un demonio que mantenga continuamente en el puerto 22 el servicio de comunicación segura.
- La forma de efectuar su trabajo es muy similar a SSL.
    - El cliente envía una señal al servidor pidiéndole comunicación por el puerto 22.
    - El servidor acepta la comunicación, en el caso de poder mantenerla bajo encriptación mediante un algoritmo definido, y le envía la clave pública al cliente para que pueda descifrar los mensajes.
    - El cliente recibe la clave teniendo la posibilidad de guardar la clave para futuras comunicaciones o destruirla después de la sesión actual.

4.- CRIPTOGRAFÍA EN JAVA

Java nos proporciona diferentes **APIs** para la creación de aplicaciones con comunicaciones seguras. Con estas APIs podremos realizar, entre otras las siguientes tareas:

- Encriptación de la información con clave pública y privada.
- Firma digital y su verificación.
- Resúmenes de mensajes (HASH).
- Certificados digitales y validación de certificados.
- Comunicaciones de red seguras con el protocolo SSL.

Además, estas APIs permiten utilizar diferentes algoritmos criptográficos (DES, RSA, MD5, etc.), proporcionados por lo que Java denomina proveedor de servicios criptográficos (PSC).

La arquitectura criptográfica de Java fue introducida en el JDK 1.1, incluyendo bibliotecas de acceso a funciones criptográficas de propósito general, conocidas como **Arquitectura Criptográfica de Java (JCA)**, y otras utilidades criptográficas que complementan a las primeras y que se conocen como **Extension Criptográfica de Java (JCE)**.

A partir de JDK 1.4, JCE viene incluido en el JDK y la distinción entre ambos componentes cada vez es menos evidente, por lo que podemos decir que JCA es el core o corazón del API de seguridad en Java y que JCE es una parte de la JCA.

Los paquetes que conforman la **arquitectura criptográfica JCA y JCE** son:

- Paquete java.security.
- Paquete javax.crypto.

Otras bibliotecas disponibles en el JDK, y que utilizan la arquitectura proveedor de JCA, son las que proporcionan:

- **Comunicaciones seguras.**
  - Extensión Java Sockets Seguros.(JSSE). javax.net.ssl.SSLSocket
  - Servicio General de Seguridad a través de Kerberos. (JGSS).
  - Capa de autenticación y seguridad Java (SASL). javax.security.sasl.Sasl
- **Autenticación y control de Acceso.**
  - Servicio de Autenticación y Autorización. (JAAS). javax.security.auth.login.

4.1.- Arquitectura criptográfica en Java

La **arquitectura criptográfica de Java**, denominada **JCA**, es la pieza principal de la criptografía de Java y está diseñada en torno a los siguientes principios:

- **Independencia de la aplicación y del algoritmo**. Las aplicaciones pueden utilizar servicios criptográficos, como firmas digitales y resúmenes de mensajes sin preocuparse por los detalles de implementación o de los detalles de los algoritmos que forman la base de estos conceptos.
- **Interoperabilidad**. Los diferentes proveedores son compatibles con todas las aplicaciones, esto es, una aplicación concreta no tiene por qué estar ligada a un proveedor específico, ni un proveedor estará ligado a una aplicación específica.
- **Extensibilidad**. La plataforma Java incluye una serie de proveedores integrados que implementan un conjunto básico de servicios de seguridad utilizados ampliamente en la actualidad, pero también permite la instalación de proveedores personalizados que implementan nuevos servicios.

La arquitectura JCA, emplea un modelo basado en el uso de proveedores, de manera que se puede decir que incluye dos tipos de componentes:

- Motores criptográficos: paquete java.security y javax.crypto.

- Proveedores de servicios criptográficos que proporcionan las implementaciones de diferentes algoritmos criptográficos, como por ejemplo SUN.

La extensión criptográfica de Java (JCE), emplea el mismo modelo basado en proveedores que JCA, en este caso está presente mediante el proveedor SunJCE y proporciona una serie de servicios que permiten complementar los servicios de seguridad de JCA, mediante los paquetes javax.crypto, javax.crypto.spec, javax.crypto.interfaces.

Ya te hemos comentado que a partir de JDK 1.4, la distinción entre JCA y JCE es menos evidente, al ir incluido JCE en el propio JDK, de manera que se habla de estos componentes de manera conjunta. Por tanto, podemos decir que **la arquitectura criptográfica de Java incluye los siguientes componentes software**:

- Las **bibliotecas de clases e interfaces** que proporcionan las diferentes funcionalidades de seguridad, como son:
    - java.security. Consiste básicamente en clases abstractas e interfaces que proporcionan el manejo de certificados, claves, resúmenes de mensajes y firmas digitales.
    - javax.crypto. Proporciona las clases e interfaces para realizar operaciones criptográficas como encriptación/desencriptación, generación de claves y acuerdo de claves, así como generación de códigos de autentificación de mensajes (MAC).
    - javax.crypto.spec. Incluye varias clases de especificación de claves y de parámetros de algoritmos.
    - javax.crypto.interfaces. presenta las interfaces de las claves empleadas en los algoritmos de tipo Diffie-Hellman (clases DHKey, DHPrivateKey y DHPublicKey).
- Los **proveedores de servicios criptográficos**, tales como Sun, SunRsaSign, SunJCE, que contienen las implementaciones de diferentes algoritmos criptográficos.

Lo que se conoce como **Arquitectura de Proveedores** es lo que permite que coexistan múltiples implementaciones de algoritmos criptográficos.

4.2.- Proveedores y motores criptográficos

Un **proveedor de servicios criptográficos** no es ni más ni menos que una empresa de seguridad que genera sus propios servicios de seguridad en Java, implementando las clases e interfaces definidos en JCA/JCE y siguiendo el estándar definido en JCA.

Un **motor criptográfico** (engine) es el conjunto de clases e interfaces que debe implementar un proveedor de servicios criptográficos. Algunas de ellas son las siguientes:

- `MessageDigest`. Para resúmenes de mensajes.
- `Signature`. Para firmas digitales.

- `KeyFactory`. Factoría para el manejo de claves seguras.
- `KeyPairGenerator`. Para generar claves públicas y privadas.
- `Cipher`. Para encriptación y desencriptación de información.

Cada una de estas clases incluyen un método getInstance() que devuelve un algoritmo criptográfico de un proveedor dado.

Un proveedor por defecto implementado dentro de JDK es el proveedor de nombre SUN que proporciona:

- Implementación de DSA, MD5 y SHA-1.
- Generación de claves públicas y privadas para el algoritmo DSA.
- Factoría de claves que soportan conversión de claves públicas a privadas.
- Construcción de certificados X.509.

Así por ejemplo:

- `MessageDigest.getInstance("MD5");` Obtiene el algoritmo MD5 del proveedor por defecto de más prioridad.
- `MessageDigest.getInstance("MD5", "ProveedorA");` Obtiene el algoritmo MD5 del proveedor ProveedorA.

Y ¿cómo **podemos utilizar más de un proveedor criptográfico en nuestra aplicación**? Para ello solo tienes que modificar el fichero `java.security` situado en el directorio `lib/security`. El formato de definición de cada uno de los proveedores es el siguiente:

- `security.provider.n=nombre_de_la_clase`.

4.3.- 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.
- La **clase** `SecureRandom` permite generar números aleatorios seguros.
  - El método `setSeed()` permite establecer el valor de la semilla.
  - El constructor `secureRandom()` utiliza la semilla del proveedor SUN.
  - El método `next()` y el `nextBytes()` obtienen el valor de los números generados.

La creación de claves se basa en el tamaño de las mismas, de manera que, si incrementas mucho el tamaño de una clave, el tiempo de cálculo de la misma también se incrementará; y esto puede suponer que la administración de claves no sea lo suficientemente ágil para una determinada aplicación.

El siguiente código java genera una pareja de claves (pública y privada) mediante la clase `KeyPairGenerator` y muestra los valores obtenidos de la `PrivateKey` y la `PublicKey`.

```
 //bibliotecas necesarias
import java.security.KeyPair;
import java.security.KeyPairGenerator;

/**
 *
 * @author IMCG
 */
public class Main {
    //Programa que crea una pareja de claves (pública y privada) y
las muestra
    public static void main(String[] args) {
      //Asigna al objeto claves de tipo keyPair el par de claves
generadas
      //por el método GeneraParejaClave()
      KeyPair claves=GeneraParejaClave();
     //Imprime el valor de las claves generadas en diferentes
formatos
      System.out.println("Algoritmo Kprivada: "
              +claves.getPrivate().getAlgorithm());
      System.out.println("Codificación Kprivada: "
              +claves.getPrivate().getFormat());
      System.out.println("Bytes Kprivada: "
              +claves.getPrivate().toString());
      System.out.println("Algoritmo Kpública: "
              +claves.getPublic().getAlgorithm());
      System.out.println("Codificación Kpública: "
              +claves.getPublic().getFormat());
      System.out.println("Bytes Kpública: "
              +claves.getPublic().toString());
    }
    //Método que genera una clave tipo KeyPair (uan pareja de claves)
     public static KeyPair GeneraParejaClave() {
       KeyPair claves = null;
```

```
        try{
          //Crea el objeto para generar un par de claves mediante RSA
          KeyPairGenerator genera=KeyPairGenerator.getInstance("RSA");
          genera.initialize(512); //asigna tamaño de la clave
          claves=genera.generateKeyPair(); //genera la pareja de
claves
        }catch(Exception e){
          e.printStackTrace();
        }
        return claves;
    }

}
```

4.4.- Resúmenes de mensajes con la clase *MessageDigest*

La clase `MessageDigest` del paquete `java.security` permite la creación de resúmenes de mensajes con el algoritmo y proveedor especificados. Los métodos que debes utilizar para crear un resumen de mensaje son:

- `getInstance()`: obtiene el algoritmo de resumen.
- `update()`: obtiene el resumen.
- `digest()`: completa la obtención del resumen.

En JDK podemos encontrar dos **algoritmos de resumen de mensajes**:

- MD5. Genera una salida de 128 bits de longitud fija.
- SHA-1. Genera una salida de 160 bits.

El siguiente código es un ejemplo sencillo de creación de un resumen de mensaje con SHA-1:

```
import java.security.*;
public class Main {
      public static void main(String[] args) {
            try {
            MessageDigest sha1 = MessageDigest.getInstance("SHA1");
            String texto = "Texto para el mensaje ejemplo SHA1";
            sha1.update(texto.getBytes()); //obtiene el resumen
            byte[] resumen = sha1.digest(); //completa la generación
del resumen
            for (int k = 0; k < resumen.length; k++) { //muestra el
resumen
            System.out.println("(" + resumen[k] + ")");
            }
            } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            }
      }
}
```

4.5.- Firma digital con la clase *Signature de java.security*

La clase `Signature` del paquete `java.security` permite realizar una firma digital, así como hacer su verificación.

Los pasos que debes seguir 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.

EL algoritmo de firma digital DSA viene implementado en el JDK de SUN, es parte del estándar de firmas digitales DSS y con él se pueden utilizar los algoritmos de resumen MD5 y SHA-1. Es el que utilizaremos en nuestro ejemplo.

En este ejemplo tienes un programa que firma digitalmente un texto y verifica su firma digital.

```
//bibliotecas necesarias
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;

public class Main {

  public static void main(String[] args) {
    String texto = "texto de prueba para ser firmado";
    KeyPair clave = generarClaves();
    byte[] textoFirmado = hacerFirma(texto.getBytes(),
            clave.getPrivate());
    if (verificarFirma(texto.getBytes(), clave.getPublic(),
textoFirmado)) {
      System.out.println("Firma realizada y verificada
correctamente");
    } else {
      System.out.println("Firma incorrecta");
    }
```

```
  }

  //método que genera una pareja de claves (pública y privada)
  //que se utilizarán en la firma digital
  public static KeyPair generarClaves() {
    //inicializa el objeto claves, tipo KeyPair, a null
    KeyPair claves = null;
    try {
      //Indica el algoritmo a utilizar en la generación de claves
      KeyPairGenerator generador =
KeyPairGenerator.getInstance("DSA");
      //asigna la pareja de claves generadas al objeto tipo KeyPair,
claves
      claves = generador.genKeyPair();
    } catch (NoSuchAlgorithmException e) {
      e.printStackTrace();
    }
    //retorna un objeto tipo KeyPair
    return claves;
  }
//método que realiza la firma digital del texto o datos y la devuelve

  public static byte[] hacerFirma(byte[] datos, PrivateKey clave) {
    byte[] firmado = null;
    try {
      //crea el objeto tipo Signature con algoritmo DSA
      Signature firma = Signature.getInstance("DSA");
      //inicializa la firma con la clave privada a utilizar
      firma.initSign(clave);
      //obtine el resumen del mensaje
      firma.update(datos);
      //obtiene la firma digital
      firmado = firma.sign();
    } catch (Exception e) {
      e.printStackTrace();
    }
    //devuleve la firma digital
    return firmado;
  }
//Método que verifica la firma digital, devolviendo:
//false, si la firma no es correcta o se produce una excepción
//verdadero, si la firma es correcta
  public static boolean verificarFirma(byte[] texto, PublicKey clave,
          byte[] textoFirmado) {
    try {
      //crea el objeto tipo Signature con algoritmo DSA
      Signature firma = Signature.getInstance("DSA");
      //verifica la clave pública
      firma.initVerify(clave);
      //actualiza el resumen de mensaje
      firma.update(texto);
      //devuelve el resultado de la verificación
      return (firma.verify(textoFirmado));
    } catch (Exception e) {
      e.printStackTrace();
    }
    return false;
  }
}
```

4.6.- Encriptación con la clase *Cipher* del paquete *javax.crypto*

La **clase** `Cipher` **permite realizar encriptación y desencriptación, tanto con clave pública como privada**. Para ello:

- Mediante el método `getInstance()` se indica el algoritmo y proveedor que utilizará el objeto cifrador `Cipher`.
- Mediante el método `init()` se indicará el modo de operación del objeto `Cipher`, por ejemplo encriptar o desencriptar.
- Mediante los métodos `update()` y `doFinal()` se insertarán datos en el objeto cifrador.

Podemos utilizar diferentes modos de operación, entre ellos:

- `ENCRYPT_MODE`. Es el modo encriptación.
- `DECRYPT_MODE`. Es el modo desencriptación.
- `WRAP_MODE`. Convierte la clave en una secuencia de bytes para transmitirla de forma segura.
- `UNWRAP_MODE`. Permite obtener las claves generadas con `WRAP_MODE`.

La **encriptación mediante un objeto** `Cipher` puede ser:

- **De bloque o `Block Cipher`**. El texto a cifrar se divide en bloques de una tamaño fijo de bits, normalmente 64 bits. Cada uno de estos bloques se cifra de manera independiente, y posteriormente se construye todo el texto cifrado. Si el texto a cifrar no es múltiplo de 64 se completa con un relleno o padding.

  Por ejemplo: `Cipher.getInstance("Rijndael/ECB/PKCS5Padding")` indica que:

  - Se utiliza el algoritmo `Rijndael`.
  - El modo es ECB (Electronic Code Book).
  - El relleno es PKCS5 Padding.
- **De flujo o `Stream Cipher`**. El texto se cifra bit a bit, byte a byte o carácter a carácter, en lugar de bloques completos de bits. Resulta muy útil cuando hay que transmitir información cifrada según se va creando, eso es, se cifra sobre la marcha.

Hemos hablado del uso de la encriptación para garantizar la confidencialidad en el envío de mensajes, pero también podemos utilizarla para proteger información almacenada.

¿Cómo podemos cifrar un fichero almacenado en disco? Este es el ejemplo que precisamente vamos a ver a continuación.

```
//Debes crear el archivo c:\cripto\fichero antes de ejecutarlo

import java.security.*; //JCA
import javax.crypto.*; //JCE
import java.io.*; //ficheros

//Programa que encripta y desencripta un fichero
//mediante clave privada o simétrica utilizando el algoritmo DES
public class Main {
```

```
  public static void main(String[] Args) {
    //declara e incializa objeto tipo clave secreta
    SecretKey clave = null;

    //llama a los métodos que encripta/desencripta un fichero
    try {
      //Llama al método que encripta el fichero que se pasa como
parámetro
      clave = cifrarFichero("c:\\cripto\\fichero");
      //Llama la método que desencripta el fichero pasado como primer
parámetro
      descifrarFichero("c:\\cripto\\fichero.cifrado", clave,
              "c:\\cripto\\fichero.descifrado");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  //método que encripta el fichero que se pasa como parámetro
  //devuelve el valor de la clave privada utilizada en encriptación
  //El fichero encriptado lo deja en el archivo de nombre
fichero.cifrado
  //en el mismo directorio
  private static SecretKey cifrarFichero(String file) throws
NoSuchAlgorithmException, NoSuchPaddingException, FileNotFoundException,
IOException, IllegalBlockSizeException, BadPaddingException,
InvalidKeyException {
    FileInputStream fe = null; //fichero de entrada
    FileOutputStream fs = null; //fichero de salida
    int bytesLeidos;

    //1. Crear e inicializar clave
    System.out.println("1.-Genera clave DES");
    //crea un objeto para generar la clave usando algoritmo DES
    KeyGenerator keyGen = KeyGenerator.getInstance("DES");
    keyGen.init(56); //se indica el tamaño de la clave
    SecretKey clave = keyGen.generateKey(); //genera la clave privada

    System.out.println("Clave");
    mostrarBytes(clave.getEncoded()); //muestra la clave
    System.out.println();


    //Se Crea el objeto Cipher para cifrar, utilizando el algoritmo DES
    Cipher cifrador = Cipher.getInstance("DES");
    //Se inicializa el cifrador en modo CIFRADO o ENCRIPTACIÓN
    cifrador.init(Cipher.ENCRYPT_MODE, clave);
    System.out.println("2.- Cifrar con DES el fichero: " + file
            + ", y dejar resultado en " + file + ".cifrado");
//declaración  de objetos
    byte[] buffer = new byte[1000]; //array de bytes
    byte[] bufferCifrado;
    fe = new FileInputStream(file); //objeto fichero de entrada
    fs = new FileOutputStream(file + ".cifrado"); //fichero de salida
    //lee el fichero de 1k en 1k y pasa los fragmentos leidos al
cifrador
    bytesLeidos = fe.read(buffer, 0, 1000);
    while (bytesLeidos != -1) {//mientras no se llegue al final del
fichero
      //pasa texto claro al cifrador y lo cifra, asignándolo a
bufferCifrado
```

```
      bufferCifrado = cifrador.update(buffer, 0, bytesLeidos);
      fs.write(bufferCifrado); //Graba el texto cifrado en fichero
      bytesLeidos = fe.read(buffer, 0, 1000);
    }
    bufferCifrado = cifrador.doFinal(); //Completa el cifrado
    fs.write(bufferCifrado); //Graba el final del texto cifrado, si lo
hay
    //Cierra ficheros
    fe.close();
    fs.close();
    return clave;
  }
  //método que desencripta el fichero pasado como primer parámetro file1
  //pasándole también la clave privada que necesita para desencriptar,
key
  //y deja el fichero desencriptado en el tercer parámetro file2

  private static void descifrarFichero(String file1, SecretKey key,
String file2) throws NoSuchAlgorithmException, NoSuchPaddingException,
FileNotFoundException, IOException, IllegalBlockSizeException,
BadPaddingException, InvalidKeyException {
    FileInputStream fe = null; //fichero de entrada
    FileOutputStream fs = null; //fichero de salida
    int bytesLeidos;
    Cipher cifrador = Cipher.getInstance("DES");
    //3.- Poner cifrador en modo DESCIFRADO o DESENCRIPTACIÓN
    cifrador.init(Cipher.DECRYPT_MODE, key);
    System.out.println("3.- Descifrar con DES el fichero: " + file1
            + ", y dejar en  " + file2);
    fe = new FileInputStream(file1);
    fs = new FileOutputStream(file2);
    byte[] bufferClaro;
    byte[] buffer = new byte[1000]; //array de bytes
    //lee el fichero de 1k en 1k y pasa los fragmentos leidos al
cifrador
    bytesLeidos = fe.read(buffer, 0, 1000);
    while (bytesLeidos != -1) {//mientras no se llegue al final del
fichero
      //pasa texto cifrado al cifrador y lo descifra, asignándolo a
bufferClaro
      bufferClaro = cifrador.update(buffer, 0, bytesLeidos);
      fs.write(bufferClaro); //Graba el texto claro en fichero
      bytesLeidos = fe.read(buffer, 0, 1000);
    }
    bufferClaro = cifrador.doFinal(); //Completa el descifrado
    fs.write(bufferClaro); //Graba el final del texto claro, si lo hay
    //cierra archivos
    fe.close();
    fs.close();
  }

  //método que muestra bytes
  public static void mostrarBytes(byte[] buffer) {
    System.out.write(buffer, 0, buffer.length);
  }
}
```

Otro ejemplo de cifrado, pero esta vez mediante clave pública y utilizando el algoritmo RSA es el siguiente:

```
import java.io.IOException;
import java.security.*;
import javax.crypto.*;

//Encriptar y desencriptar un texto mediante clave pública RSA
public class Main {

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

    System.out.println("Crear clave pública y privada");
    //Crea e inicializa el generador de claves RSA
    KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
    keyGen.initialize(512);//tamaño de la clave
    KeyPair clavesRSA = keyGen.generateKeyPair();
    PrivateKey clavePrivada = clavesRSA.getPrivate();//obtiene clave
privada
    PublicKey clavePublica = clavesRSA.getPublic();//obtiene clave
pública

    //muestra las claves generadas
    System.out.println("clavePublica: " + clavePublica);
    System.out.println("clavePrivada: " + clavePrivada);
    //texto a encriptar o cifrar
    byte[] bufferClaro = "Este es el mensaje secreto\n".getBytes();

    //Crea cifrador RSA
    Cipher cifrador = Cipher.getInstance("RSA");
    //Pone cifrador en modo ENCRIPTACIÓN utilizando la clave pública
    cifrador.init(Cipher.ENCRYPT_MODE, clavePublica);
    System.out.println("Cifrar con clave pública el Texto:");
    mostrarBytes(bufferClaro);

    //obtiene todo el texto cifrado
    byte[] bufferCifrado = cifrador.doFinal(bufferClaro);
    System.out.println("Texto CIFRADO");
    mostrarBytes(bufferCifrado); //muestra texto cifrado
    System.out.println("\n_____________________________________");

    //Pone cifrador en modo DESENCRIPTACIÓN utilizando la clave privada
    cifrador.init(Cipher.DECRYPT_MODE, clavePrivada);
    System.out.println("Descifrar con clave privada");

    //obtiene el texto descifrado
    bufferClaro = cifrador.doFinal(bufferCifrado);
    System.out.println("Texto DESCIFRADO");
    mostrarBytes(bufferClaro);//muestra texto descifrado
    System.out.println("\n_____________________________________");
  }

  public static void mostrarBytes(byte[] buffer) throws IOException {
    System.out.write(buffer);
  }
}
```

5.- SOCKETS SEGUROS EN JAVA (JSSE)

Otra biblioteca disponible e integrada en Java desde el JDK 1.4 es JSSE (Extensión Java Sockets Seguros):

- Utiliza la arquitectura proveedor de JCA.
- Proporciona **comunicaciones seguras mediante el protocolo SSL** (Secure Sockets Layer) soportado a través del paquete javax.net.ssl.

En unidades anteriores, has estudiado y visto ejemplos de cómo la programación de sockets proporciona un mecanismo de muy bajo nivel para la comunicación e intercambio de datos entre dos ordenadores, uno considerado como cliente, que es el que inicia la conexión con el otro, el servidor, que está a la espera de conexiones de clientes.

Un socket **seguro** es un socket basado en el protocolo SSL y por tanto proporcionará en una comunicación autentificación, integridad y confidencialidad de los datos transmitidos.

Los usos más habituales de SSL en Java se basan en las siguientes clases:

- SSLSocket, para programar sockets seguros de cliente.
- SSLServerSocket, para programar sockets seguros de servidor.

Los sockets **seguros** son parecidos en su funcionamiento a los sockets normales, pero con ciertos cambios en su proceso de creación e inicialización.

5.1.- Programar un socket seguro de servidor

Para crear `sockets` seguros de servidor, Java utiliza el patrón de diseño `Factory`, de manera que lo primero que haremos siempre es obtener un objeto `SSLServerSocketFactory`. Este objeto encapsula los detalles de creación, configuración e inicialización del `socket` seguro, como es la autenticación de claves, validación de certificados, etc.

Después crearemos un objeto `SSLServerSocket`. Esta clase soporta todos los métodos estándar de la clase `ServerSocket`, además de métodos específicos para trabajar con aspectos de seguridad.

Podemos resumir los **pasos para programar un socket seguro de servidor** en los siguientes:

- Obtener un objeto `SSLServerSocketFactory`.
- Crear un objeto `SSLServerSocket` indicando el puerto de escucha del servidor.
- Crear un `socket` seguro cliente que esté atento a las posibles conexiones al servidor.
- Crear un canal seguro sobre el `socket` abierto.

En el siguiente segmento de código, puedes ver los pasos indicados anteriormente, donde el `socket` seguro del servidor escucha por el puerto 5000.

```
//Declara objeto tipo Factory para crear socket SSL servidor
SSLServerSocketFactory facto=
(SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
//Crea un socket servidor seguro
SSLServerSocket socketServidorSsl=
(SSLServerSocket)facto.createServerSocket(5000);
SSLSocket socketSsl= (SSLSocket)socketServidorSsl.accept();
//crea canal seguro sobre el socket abierto
BufferedReader entrada=new BufferedReader(
new InputStreamReader(socketSsl.getInputStream()))
```

5.2.- Programar un socket seguro de cliente

Para crear un `socket` seguro cliente se puede utilizar un objeto `SSLSocketFactory`, aunque no siempre es necesario, pues también puede crearse en el momento de aceptar una conexión del servidor en obligado o in-bound connection.

Después se crearemos un objeto `SSLSocket`. Esta clase soporta todos los métodos estándar de la clase `Socket`, además de métodos específicos para trabajar con `sockets` seguros.

En resumen, los pasos para programar un `socket` seguro cliente pueden ser:

- Obtener un objeto `SSLSocketFactory`.
- Crear un objeto `SSLSocket` indicando el nombre del servidor y puerto de escucha.
- Crear un canal seguro de comunicación con el servidor.

En el siguiente segmento de código puedes ver los pasos indicados para crear un `socket` cliente seguro, que se conectará al servidor de nombre `localhost` y por el puerto 5000:

```
//Declara un objeto tipo Factory para crear sockets SSL
SSLSocketFactory facty=
(SSLSocketFactory)SSLSocketFactory.getDefault();
//Crea un socket seguro
SSLSocket socketSsl=
(SSLSocket)facty.createSocket("localhost", 5000);
//Consola que lee la entrada del usuario
BufferedReader entrada=new BufferedReader (
new InputStreamReader(System.in));
//Canal de comunicación con el servidor
BufferedWriter salida= new BufferedWriter(
new OutputStreamWriter(socketSsl.getOutputStream())
```

Puesto que SSL utiliza certificados digitales para la autentificación, necesitamos crear el certificado para nuestro servidor. JSSE puede utilizar los certificados creados por la herramienta keytool de Java. De esta forma, la creación de los certificados la podemos hacer con la ejecución de keytool y con los parámetros pertinentes.

Aquí tienes un ejemplo de una aplicación echo que incluye seguridad entre servidor y clientes. El servidor crea una conexión sobre un socket servidor seguro que atenderá conexiones desde clientes que se identifiquen con un certificado válido. El certificado se genera con la herramienta keytool.

Servidor

```
import java.io.*;
import javax.net.ssl.*;

public class Servidor {
    public static void main( String[] args ) throws IOException {

// Indicamos el certificado a usar, previamente generado, si
// no está en el directorio del proyecto habrá que poner la ruta
      System.setProperty("javax.net.ssl.keyStore", "StoreSSL");
      System.setProperty("javax.net.ssl.keyStorePassword",
"12345678");
      //La contraseña se estableció al crear el certificado con
keytool

      // Obtenemos el objeto de tipo Factory para crear sockets SSL
        SSLServerSocketFactory fact =

(SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
        // Utilizamos el objeto para crear un socket servidor seguro
        SSLServerSocket socketServidorSsl =
            (SSLServerSocket)fact.createServerSocket(9999);
        SSLSocket socketSsl = (SSLSocket)socketServidorSsl.accept();

        // Creamos un canal de entrada sobre el socket seguro que
        // hemos abierto
        BufferedReader entrada = new BufferedReader(
            new InputStreamReader(socketSsl.getInputStream()));

        String linea = null;
        System.out.println( "Esperando..." );
        // Presentamos todas las líneas que vayan llegan entrando en
        // el canal a través del socket
        while( (linea = entrada.readLine()) != null ) {
            System.out.println("Recibido: " + linea );
            System.out.flush();
        }
        entrada.close();
        socketSsl.close();
        socketServidorSsl.close();
    }
}
```

Cliente

```
import java.io.*;
import javax.net.ssl.*;

public class Cliente {
    public static void main( String[] args ) throws IOException {

      // Indicamos el certificado a usar, previamente importado, si
      // no está en el directorio del proyecto habrá que poner la ruta
      System.setProperty("javax.net.ssl.trustStore", "TrustSSL");
      System.setProperty("javax.net.ssl.trustStorePassword", "87654321");
      //La contraseña se estableció al importar el certificado con keytool

      // Obtenemos el objeto de tipo Factory para crear sockets SSL
        SSLSocketFactory fact =
(SSLSocketFactory)SSLSocketFactory.getDefault();
        // Utilizamos el objeto para crear un socket seguro
        SSLSocket socketSsl = (SSLSocket)fact.createSocket(
"localhost",9999 );

        // Consola desde la que leemos la entrada del usuario
        BufferedReader entrada = new BufferedReader(
            new InputStreamReader(System.in));
        // Canal de comunicación con el servidor de eco
        BufferedWriter salida = new BufferedWriter(
            new OutputStreamWriter(socketSsl.getOutputStream()));

        String linea = null;
        System.out.println( "Listo..." );
        // Vamos enviando las líneas al servidor
        while( (linea = entrada.readLine()) != null ) {
            salida.write( linea+'\n' );
            salida.flush();
        }
        entrada.close();
        salida.close();
        socketSsl.close();
    }
}
```

Para crear un certificado es necesario utilizar la herramienta **keytool**, proporcionada por la plataforma Java 2.

A continuación, se reproducen los comandos utilizados en el servidor y en el cliente para generar un certificado, exportar su clave pública a un archivo de certificado e importar dicho certificado en el cliente a partir del archivo de certificado:

1. Se crea el certificado asignándole un alias *claveSsl,* usando el algoritmo *RSA,* y almacenándolo en el almacén de claves (keystore) *StoreSSL*. Nos pedirá una serie de datos para identificar al creador del certificado y contraseñas para proteger el almacén y la clave privada

   ```
   % keytool -genkey -alias claveSsl -keyalg RSA -keystore StoreSSL
   ```

2. El siguiente comando exporta el certificado a un archivo llamado *claveSSL.crt*:

   ```
   % keytool -export -alias claveSsl -keystore StoreSSL -rfc -file
   claveSSL.crt
   ```

3. Finalmente, es necesario incorporar en el cliente el certificado al nuevo almacenamiento para permitir realizar la validación; lo cual se hace con el comando siguiente:

```
% keytool -import -alias claveSsl -file claveSSL.crt -keystore
TrustSSL
```

En el código del servidor se usaron las siguientes líneas para hacer uso del certificado:

```
System.setProperty("javax.net.ssl.keyStore", "StoreSSL");

System.setProperty("javax.net.ssl.keyStorePassword",     "12345678");
//La contraseña se estableció al crear el certificado con keytool
```

En el código del cliente se usaron las siguientes líneas para hacer uso del certificado:

```
System.setProperty("javax.net.ssl.trustStore", "TrustSSL");

System.setProperty("javax.net.ssl.trustStorePassword",    "87654321");
//La contraseña se estableció al importar el certificado con keytool
```