MEDAC
Instituto Oficial de Formación Profesional

PROGRAMACIÓN DE SERVICIOS Y PROCESOS
TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Comunicaciones seguras

15

ÍNDICE

OBJETIVOS

Conocer los protocolos más importantes de comunicaciones seguras.

Profundizar sobre el protocolo SSL/TLS y sus propiedades.

Realizar programas con datos cifrados, con transmisiones en red seguras, y utilizando sockets seguros.

/ 1. Introducción y contextualización práctica

En esta unidad pondremos de manifiesto por qué en una comunicación deberemos encriptar las comunicaciones para hacerlas seguras, utilizando una serie de protocolos seguros.

Vamos a estudiar cuáles son esos protocolos seguros, y haremos hincapié en uno de los más importantes, el protocolo SSL/TLS.

También seguiremos ahondando sobre el concepto de encriptación, para continuar protegiendo la información de forma segura.

Por último, experimentaremos sobre cómo podremos realizar una comunicación en red de forma segura, utilizando para ello los *sockets* seguros.

Escucha el siguiente audio en el que planteamos el caso práctico que iremos resolviendo a lo largo de esta unidad.

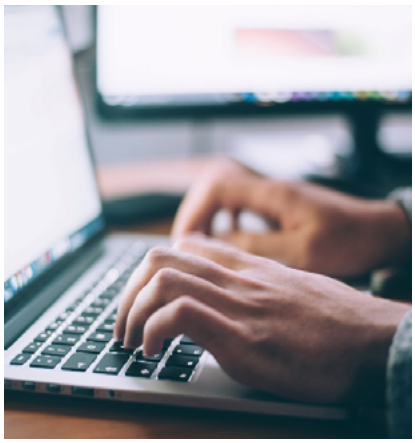

Fig. 1. Comunicación segura

Audio Intro. "La seguridad de las comunicaciones"

https://bit.ly/331MIud

/ 2. Protocolos seguros de comunicaciones

La protección de la información cuando se desarrollan aplicaciones que tienen comunicaciones en red es algo que debemos proporcionar, ya que las **comunicaciones mediante una red pueden ser fácilmente interceptadas**, y, por lo tanto, pueden ser manipuladas por personas indeseadas.

Cuando combinamos el poder de la criptografía con las comunicaciones en red, estaríamos creando lo que se conoce como **protocolos seguros de comunicación**, o también llamados **protocolos criptográficos, o de encriptación**.

Existen multitud de este tipo de protocolos, pero nos centraremos en los dos más comunes:

1. **Protocolo SSL:** Este protocolo proporciona la posibilidad de tener una **comunicación segura en el modelo cliente/servidor**, protegiéndolo de ataques en la red, como puede ser el problema de "*man in the middle*" u hombre en el medio, que consiste en que un tercero se dedique a esnifar el tráfico de la red de comunicaciones, pudiendo acceder a información confidencial.

2. **Protocolo TLS**: Este protocolo surgió como una evolución del protocolo SSL, proporcionando la posibilidad de utilizar muchos más **algoritmos criptográficos para codificar la información enviada en las comunicaciones.**

Los protocolos SSL y TLS son unos protocolos criptográficos que podemos encontrar entre las capas de aplicación y de transporte del modelo TCP/IP, gracias a esto, vamos a poder utilizarlos para realizar cifrados de información en protocolos como Telnet, IMAP, FTP, HTTP...

Siempre que un protocolo de encriptación como SSL o TLS sea ejecutado sobre un protocolo de comunicación, obtendremos la versión segura del mismo:

- **FTPS:** Versión segura del protocolo FTP.
- **HTTPS:** Versión segura del protocolo HTTP.
- **SSH:** Versión segura del protocolo Telnet.

Según esto, podemos afirmar que, por ejemplo, el protocolo FTPS no es que sea más importante que el protocolo FTP, pero al estar ejecutándose en él un protocolo de criptografía, ya es un protocolo seguro.

CAPA	Objetos de transmisión
APLICACIÓN	Mensajes
TRANSPORTE	Paquetes
RED (INTERNET)	Datagramas
ENLACE	Tramas
FÍSICA	Bits

Fig. 2. Modelo TCP/IP

/ 3. Características SSL/TLS

El **protocolo SSL** (*Secure Sockets Layer*) fue creado por la empresa Netscape en un afán de hacer seguras las **comunicaciones entre los navegadores web y los servidores**, aunque se podía, y se puede, utilizar en cualquier aplicación con esquema cliente/servidor.

El protocolo SSL nos va a proporcionar las propiedades que hacen seguras a las comunicaciones. Estamos hablando de:

- Autenticación.
- Confidencialidad.
- Integridad.

El funcionamiento del protocolo SSL consiste en que antes de poder tener una comunicación segura entre cliente y servidor, deben de "negociarse" una serie de condiciones o parámetros para dicha comunicación, esto se conoce como **apretón de manos o *handshake*,** conocido como el ***SSL/TLS Handshake Protocol*.**

También existe la versión del llamado ***SSL/TLS Record Protocol*,** mediante el cual se van a especificar de qué forma se van a encapsular los datos que serán transmitidos, pudiéndose incluso negociar los datos de la propia negociación previa.

Las **fases** que se utilizan en el protocolo SSL son las siguientes:

1. En la llamada **fase inicial** se negocian los algoritmos criptográficos que se van a utilizar en la comunicación.
2. La siguiente **fase** es la de **autenticación**, en la que se intercambiarán las claves y se autenticarán las partes mediante certificados de criptografía asimétrica. En esta fase es donde se van a crear las claves necesarias para realizar la transmisión de información.
3. En la última **fase** se hará una **verificación** de qué el canal es seguro para la comunicación.
4. En este punto ya comenzaría la **comunicación segura** de información.

Si por cualquier motivo fallase la negociación, la comunicación no llegaría a establecerse.

Podemos utilizar los siguientes algoritmos criptográficos para ser utilizados con SSL/TLS:

- **Algoritmos de clave simétrica**: IDEA, DES...
- **Algoritmos de clave pública**: RSA.
- **Certificados digitales**: RSA.
- **Resúmenes**: MD5, SHA...

Audio 1. "Razones del éxito de SSLT/TLS"
https://bit.ly/3pDf7AB

/ 4. Caso práctico 1: "Máxima seguridad"

Planteamiento: Pilar y José están repasando todos los protocolos seguros que van a necesitar a lo largo de la unidad, y son muchos y sofisticados, ya que la seguridad en la información es algo muy serio.

Durante el tiempo de estudio de todos estos protocolos, nuestros amigos se plantean implementar un sistema que ofrezca la máxima seguridad posible, y se hacen una pregunta, "¿con tantos protocolos seguros como existen, ¿cuáles de ellos serán los más seguros?, ¿los podremos combinar todos entre sí para ofrecer la máxima seguridad?"

Nudo: ¿Qué piensas al respecto? ¿Cómo crees que podría hacerse una combinación de protocolos para más seguridad? ¿Acaso es necesaria esta combinación?

Desenlace: Es normal que cuando empezamos a estudiar todos los protocolos que existen para seguridad en las comunicaciones, podamos estar un poco perdidos como nuestros amigos.

La idea de los protocolos es encapsular la información de la mejor forma posible para dar una mayor seguridad a los datos, así que, si éstos siguen esta filosofía, no sería muy recomendable el uso de multitud protocolos seguros, ya que por sí solos suelen ser bastante efectivos, y lo único que estaríamos consiguiendo es ralentizar la comunicación con todas las operaciones de encriptado extra.

Esto no quiere decir que no se pueda llevar a cabo dicha combinación de protocolos para obtener la máxima seguridad. Pueden existir determinados entornos para los que la información sea extremadamente sensible, por lo que, en estos casos, se podría llevar a cabo una combinación de varios protocolos, aunque sea en detrimento del tiempo que se necesitaría para llevar a cabo las comunicaciones de los datos.

Por otra parte, existen opciones sencillas de combinación de protocolos que no conllevarían mucha ralentización en las comunicaciones, y aportarían un plus de seguridad a nuestros proyectos. Un ejemplo de esto podría ser el uso de comunicaciones seguras vía SSL/TLS y los datos que se transmitan en dichas comunicaciones que puedan estar encriptados con protocolos de seguridad como puede ser un cifrado con un certificado digital.

Fig. 3. Máxima seguridad para combatir el crimen digital

/ 5. Encriptación de datos con *Cipher*

El lenguaje de programación Java nos proporciona la clase *Cipher*, mediante la cual vamos a poder realizar **codificaciones de datos**.

Uno de los **algoritmos de encriptado** que vamos a poder utilizar es el **cifrado AES,** no obstante, se admiten muchos modos de operación más.

El algoritmo de cifrado AES, *Advanced Encryption Standard* por sus siglas en inglés, es un cifrado de esquema por bloques que se comenzó a utilizar en Estados Unidos. AES posee un tamaño de bloque fijo de 128 bits y tamaños de clave de 128, 192 o 256 bits. La mayoría de los cálculos del algoritmo AES se hacen en un campo finito determinado.

Para poder utilizar AES deberemos descargar e integrar la **biblioteca de *commons-codec* de Apache,** la cual podemos descargar desde el siguiente enlace:

http://commons.apache.org/proper/commons-codec/download_codec.cgi

Descargamos los binarios y nos quedamos con el fichero *commons-codec-x.yy.jar*. Siendo *x* e *yy* las versiones actuales del paquete.

Una vez descargada la integramos en nuestro proyecto NetBeans.

Para encriptar un mensaje con una clave mediante el algoritmo AES podemos usar el código que podemos ver en la figura 4.

```
public static void main(String[] args)
{
    String key = "92AE31A79FEEB2A3";    // llave
    String iv = "0123456789ABCDEF";
    String mensaje = "Hola mundo";

    try
    {
        // Proceso de encriptado con AES
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        SecretKeySpec skeySpec = new SecretKeySpec(
            key.getBytes(), "AES");
        IvParameterSpec ivParameterSpec = new
        IvParameterSpec(iv.getBytes());
        cipher.init(Cipher.ENCRYPT_MODE, skeySpec, ivParameterSpec);
        byte[] encrypted = cipher.doFinal(mensaje.getBytes());
        System.out.println("El mensaje encriptado es: " + new
        String(encodeBase64(encrypted)));
    }
    catch(InvalidAlgorithmParameterException | InvalidKeyException |
    NoSuchAlgorithmException | BadPaddingException |
    IllegalBlockSizeException | NoSuchPaddingException error)
    {
        System.out.println("Error");
    }
}
```

Fig. 4. Cifrado AES

Vídeo 1. "Clases para implementar una firma digital en Java"
https://bit.ly/36N3n60

/ 6. Certificados para *sockets* seguros

El lenguaje de programación Java nos permite crear una versión segura de los *sockets* que utilizamos en unidades anteriores.

Si recordamos, utilizamos los *sockets* para establecer una comunicación en red entre las dos partes del modelo cliente/servidor, pero los *sockets* que utilizamos anteriormente no eran seguros, ya que no cifraban la información.

Para poder utilizar la versión segura de los *sockets*, primeramente, deberemos **crear los certificados que nos ayudarán a encriptar la información.**

Para crear el certificado del servidor necesitamos abrir una **ventana de comandos** (cmd de windows o bash de linux) **y ejecutar** el siguiente comando (Chudiang, 2016):

```
keytool -genkey -keyalg RSA -alias serverKay -keystore serverKey.jks -storepass servpass
```

Cuando ejecutemos el comando, el sistema nos irá solicitando una serie de datos, como nuestro nombre, apellidos, etc. Un certificado va a ir asociado a alguna persona u organización, es por este motivo por lo que en este punto nos solicita esos datos.

Cuando tengamos el **certificado del servidor**, debemos generar el **certificado para el cliente.** Como el certificado del servidor está dentro de un almacén, tenemos que sacarlo de ahí a un fichero. El comando para generar el certificado del cliente que debemos usar es:

```
keytool -export -keystore serverKey.jks -alias serverKay -file ServerPublicKey.cer
```

Repetimos todo el proceso para generar los ficheros del cliente. Creamos el certificado del cliente en un almacén de certificados *clientKey.jks*:

```
keytool -genkey -keyalg RSA -alias clientKay -keystore clientKey.jks -storepass clientpass
```

Para finalizar metemos la **clave pública del cliente** en los certificados de confianza del servidor:

```
keytool -import -v -trustcacerts -alias clientKay -file ClientPublicKey.cer -keystore
serverTrustedCerts.jks -keypass servpass -storepass servpass
```

Fig. 5. Seguridad de encriptación

/ 7. Sockets seguros I: Servidor

Vamos a ver cómo podemos implementar un servidor seguro en Java. La forma fácil de crear los ***sockets* SSL** es usar las factorías de *socket* SSL por defecto que nos proporciona java. Para el lado del servidor, el código sería (Chudiang, 2016):

```
SSLServerSocketFactory serverFactory = (SSLServerSocketFactory)
```

```
SSLServerSocketFactory.getDefault();
```

```
ServerSocket serverSocket = serverFactory.createServerSocket(puerto);
```

La única pregunta que podemos hacernos en este punto es ¿cómo podemos indicar dónde se encuentran los almacenes de certificados y certificados de confianza que generamos en el paso anterior? Esto lo vamos a poder hacer mediante las **propiedades de *System*,** o bien con opciones del parámetro -D al arrancar nuestra aplicación java. **Las propiedades a fijar son:**

- **javax.net.ssl.keyStore:** Con el que indicamos el almacén donde está el certificado que nos identifica.
- **javax.net.ssl.keyStorePassword:** Con el que indicamos la clave para acceder a dicho almacén y para acceder al certificado dentro del almacén.

- **javax.net.ssl.trustStore:** Con el que indicamos el almacén donde están los certificados en los que se confía.
- **javax.net.ssl.trustStorePassword:** Con el que indicamos la clave para acceder a dicho almacén y a los certificados dentro del almacén.

Si decidimos hacerlo mediante *System.setProperty(),* el código quedaría así:

```
System.setProperty("javax.net.ssl.keyStore", "src/main/certificados/servidor/serverKey.jks");
```

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

```
System.setProperty("javax.net.ssl.trustStore", "src/main/certificados/servidor/
serverTrustedCerts.jks");
```

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

Fig. 6. Seguridad en la comunicación

/ 8. Caso práctico 2: "Realizando una auditoría"

Planteamiento: Nuestros amigos Pilar y José acaban de recibir otro encargo, y esta vez es uno más urgente que de costumbre, les comenta su jefe de departamento.

El motivo de la urgencia es porque una empresa ha detectado que se ha producido una filtración de información en su aplicación durante una comunicación en red. El encargo consiste en hacer una auditoría para comprobar la seguridad de su aplicación.

Nudo: ¿Qué puntos crees que deberían mirar nuestros amigos para comprobar la seguridad de las comunicaciones de la aplicación?

Desenlace: El hecho que se haya producido una filtración de información en una comunicación, por muy pequeña que sea, puede significar que ha habido un atacante esnifando el tráfico de la aplicación.

Nuestros amigos deberían empezar a comprobar si los datos que utiliza la aplicación están cifrados y en caso de que no lo estén deberán implementar un sistema de cifrado de los mismos, por ejemplo, mediante un algoritmo de criptografía asimétrica, para así poder decodificar fácilmente la información cuando llegue a su destino.

Otro punto muy importante en el que deberían detenerse, ya que hay comunicaciones en red en la aplicación, será el de comprobar si los *sockets* que se utilizan para implementar dichas comunicaciones son seguros o no.

El hecho de usar *sockets* seguros puede significar la diferencia entre que un atacante pueda o no obtener la información que se está transmitiendo.

Como punto final, todas las comunicaciones deberían hacerse usando los sockets seguros, pero enviando la información codificada, para así tener un doble nivel de seguridad.

```
// Obtenemos la información para enviar
info = obtenerInfo()

// Ciframos la información
infosegura = cifrar(info, clave)

// Enviamos la información segura por
// un socket seguro
socketseguro.enviar(infosegura)
```

Fig. 7. Pasos para enviar información segura

/ 9. Sockets seguros II: Cliente

Vamos a ver cómo podemos implementar un cliente seguro en Java. Al igual que con el servidor, en el cliente la forma fácil de crear los *sockets* SSL es usar las factorías de *socket* SSL por defecto que nos proporciona java. El código en este caso sería:

```
SSLSocketFactory clientFactory = (SSLSocketFactory)
```

```
SSLSocketFactory.getDefault();
```

```
Socket cliente = clientFactory.createSocket(servidorseguro, puerto);
```

Donde mediante la variable **servidorseguro y puerto** indicamos cuáles serán la dirección IP y el puerto, donde está el servidor seguro que hemos creado anteriormente.

Para trabajar con los clientes a partir de este punto, se deberá obtener un *socket* de la forma habitual.

¿Qué inconveniente podemos encontrar en este mecanismo? Hay que remarcar que **todas las variables de configuración** que hemos visto anteriormente en la parte de creación del servidor seguro van a **afectar a todo el programa Java**.

A partir de ese momento, todos los *sockets* que abramos tendrán el mismo certificado y confiarán en los mismos certificados. Si quisiésemos poder establecer varios *sockets* con distintos certificados y distintos certificados de confianza, necesitamos una configuración más compleja.

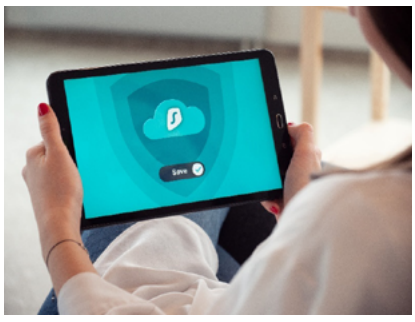

Fig. 8. Estableciendo una comunicación segura

Una vez que ya tengamos creados nuestro servidor y nuestros clientes seguros, la forma de trabajar va a ser exactamente la misma que la que utilizamos en unidades anteriores en comunicaciones en red. A continuación, veremos un ejemplo de aplicación de sockets seguros:

Vídeo 2. "Ejemplo de aplicación de sockets seguros"

https://bit.ly/36N3n60

/ 10. Resumen y resolución del caso práctico de la unidad

A lo largo de esta unidad hemos comprobado que, en una transmisión de información, deberemos siempre **encriptar las comunicaciones** para así poder hacerlas seguras, utilizando una serie de **protocolos seguros.**

Hemos visto cuáles son esos protocolos seguros, y sobre todo hemos hecho hincapié en uno de los más importantes, el protocolo **SSL/TLS**, que es el responsable de que las comunicaciones sean seguras.

También hemos aprendido cómo podemos encriptar información de forma segura, utilizando la clase ***Cipher*** para ello.

Por último, hemos practicado sobre cómo podemos realizar una comunicación en red de forma segura, utilizando para ello los ***sockets* seguros**, tanto *sockets* para crear el servidor, como para crear el cliente, garantizando la seguridad de la comunicación en ambos extremos.

Resolución del caso práctico de la unidad

Ciertamente el mundo de la seguridad en las comunicaciones es apasionante, y así lo están descubriendo nuestros amigos Pilar y José.

El hecho de asegurarnos o tratar de asegurarnos que nuestras comunicaciones son seguras es mucho más importante de lo que puede parecer en un principio.

En el caso del problema que se plantean nuestros amigos, podemos creer que basta con que no nos aparezca el típico candado amarillo en la barra de direcciones web cuando navegamos, pero tenemos que tener cuidado, ya que podemos estar siendo víctimas de un atacante que oculta esa información, así que, lo más seguro es comprobar que en nuestra barra de direcciones web utilizamos el protocolo HTTPS, y con eso ya tendremos una navegación segura.

Fig. 9. Navegando de forma segura

/ 11. Bibliografía

Colaboradores de Wikipedia. (2019, 17 julio). Seguridad de las comunicaciones. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Seguridad_de_las_comunicaciones

Colaboradores de Wikipedia. (2020a, mayo 21). Función hash. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Funci%C3%B3n_hash

Colaboradores de Wikipedia. (2020a, mayo 24). Seguridad de la capa de transporte. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Seguridad_de_la_capa_de_transporte

Colaboradores de Wikipedia. (2020, 19 agosto). Advanced Encryption Standard. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Advanced_Encryption_Standard

Luis Enrique Juarez. (2013, 14 junio). Web Services usando Java y Netbeans. YouTube. https://www.youtube.com/watch?v=BPk-vup0_0M

Socket SSL con Java - ChuWiki. (s. f.). chuwiki. Recuperado 2 de septiembre de 2020, de http://chuwiki.chuidiang.org/index.php?title=Socket_SSL_con_Java