PROGRAMACIÓN DE SERVICIOS Y PROCESOS

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Comunicaciones seguras
Protocolo SSL/TLS
Sockets seguros

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**.

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:

   "*Secure Sockets Layer*": 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 **esniffar el tráfico** de la red de comunicaciones, pudiendo **acceder** a información confidencial. (esniffar en red es interceptar y examinar).

2. Protocolo TLS:

   "*Transport Layer Security*" o Seguridad de la Capa de Transporte: 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.

Modelo TCP/IP

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

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. Fase **inicial**: se negocian los **algoritmos** criptográficos que se van a utilizar en la comunicación.

2. Fase de **autenticación**: 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**...

Razones del éxito de SLT/TLS

El éxito del protocolo SSL/TLS se debe fundamentalmente a la expansión que ha tenido el **comercio electrónico** en Internet, aunque también es utilizado para poder **crear redes privadas virtuales** o **VPN**.

Consejos para máxima seguridad

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**.

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:

Cifrado AES

```
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/PKCSSPadding");
		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.printin("El mensaje encriptado es: " +
            new String (encodeBaseé4 (encrypted) ));
        }
        catch (InvalidAlgorithmParameterException | InvalidKeyException |
        NoSuchAlgorithmException | BadPaddingException |
        IllegalBlockSizeException | NoSuchPaddingException error)
        {
        System.out.printin("Error");
        }
    }
```

Clases para implementar una firma digital en Java

El lenguaje de programación Java nos proporciona la clase **Signature**, del paquete java.security, que nos va a permitir realizar una implementación de firma digital, además de verificarla.

Vamos a ver los pasos necesarios para poder realizar este proceso.

1. Generamos las claves publicas y privadas mediante la clase **KeyPairGenerator**. Utilizaremos la clase **PrivateKey** para firmar y la clase **PublicKey** para verificar la firma.
2. Realizaremos la firma digital mediante la clase **Signature**, utilizando además un algoritmo de clave asimétrica, como puede ser el **DSA**. Utilizaremos los métodos initSign() para generar la firma, **update()** para crear el resumen del mensaje y **sign()** que terminara de crear la firma digital.
3. Verificaremos la firma mediante la clave publica. Usaremos el método **initVerify()**, al cual le pasaremos la clave publica, luego con el método **update()** actualizaremos el resumen del mensaje para comprobar si coincide con el enviado y con el método **verify()** realizaremos la verificación de la firma.

```
import java.security.*;

public class DigitalSignatureExample {
    public static void main(String[] args) {
        try {
```

```
            // Paso 1: Generar las claves pública y privada
            KeyPairGenerator keyPairGen =
                  KeyPairGenerator.getInstance("DSA");
            keyPairGen.initialize(1024);
            KeyPair keyPair = keyPairGen.generateKeyPair();
            PrivateKey privateKey = keyPair.getPrivate();
            PublicKey publicKey = keyPair.getPublic();

            // Paso 2: Realizar la firma digital
            Signature signature = Signature.getInstance("SHA256withDSA");
            signature.initSign(privateKey);
            String message = "Mensaje a firmar";
            signature.update(message.getBytes());
            byte[] digitalSignature = signature.sign();
            System.out.println("Firma digital generada correctamente.");

            // Paso 3: Verificar la firma
            signature.initVerify(publicKey);
            signature.update(message.getBytes());
            boolean verified = signature.verify(digitalSignature);
            if (verified) {
                System.out.println("La firma digital es válida.");
            } else {
                System.out.println("La firma digital no es válida.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Este código genera un par de claves pública y privada, firma un mensaje utilizando la clave privada y luego verifica la firma utilizando la clave pública.

Asegúrate de manejar adecuadamente las excepciones en un entorno de producción. Además, ten en cuenta que el algoritmo DSA utilizado aquí es solo un ejemplo.

En aplicaciones reales, es posible que desees utilizar algoritmos más robustos, como RSA o ECDSA.

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

Sockets seguros

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. ¡¡¡OJO!!
- 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 (en los que se confía). ¡¡¡OJO!!!

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”);
```

Sockets seguros

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.

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:

Ejemplo de aplicación de sockets seguros

Vamos a ver un ejemplo de Socket seguros en Java. Para este ejemplo hemos creado dos clases: 'SSLCliente', que representará un cliente seguro, y 'SSLServidor', que representará un servidor seguro.

Comenzamos por el servidor seguro. En este servidor hemos creado una **variable** del tipo '**SSLServerSocket**' que será el socket seguro para implementar nuestro servidor. En primer lugar, en el constructor de la clase le hemos **pasado el puerto** al que nos conectaremos.

Lo primero que debemos hacer es crear donde están los certificados para nuestro servidor mediante estas instrucciones.

Una vez hecho eso, deberemos obtener las claves de confianza que estarán también en la misma carpeta de los certificados en nuestro caso.

Una vez hecho eso, configuraremos nuestro contexto seguro y nos crearemos un server socket con el puerto que hemos indicado. Una vez hecho esto, nuestro servidor seguro estará creado con las claves de confianza que hemos configurado.

Hemos creado un método 'start' en el que lanzaremos una **hebra** que estará escuchando clientes seguros. Una vez que conecte con un cliente, obtendrá sus buffer de lectura para poder escribir y leer.

SSLservidorSeguro.java

```
public class SSLservidor {

    private final String SERVERPASS = "servpass";
    private final SSLServerSocket serverSocket;
    private final int puerto;

    public SSLservidor(int puerto) throws FileNotFoundException,
            KeyStoreException, IOException, NoSuchAlgorithmException,
            CertificateException, UnrecoverableKeyException,
            KeyManagementException
    {
        this.puerto = puerto;

        // Indico los certificados seguros del servidor
        KeyStore keyStore = KeyStore.getInstance("JKS");
        keyStore.load(new FileInputStream(
                "src" + File.separator
                + "certificados" + File.separator
                + "servidor" + File.separator
                + "serverKey.jks"), SERVERPASS.toCharArray());
```

```
    KeyManagerFactory kmf = KeyManagerFactory.
              getInstance(KeyManagerFactory.
                          getDefaultAlgorithm());
    kmf.init(keyStore, SERVERPASS.toCharArray());

    KeyStore trustedStore = KeyStore.getInstance("JKS");
    trustedStore.load(new FileInputStream(
            "src" + File.separator
            + "certificados" + File.separator
            + "servidor" + File.separator
            + "serverTrustedCerts.jks"), SERVERPASS
            .toCharArray());
    TrustManagerFactory tmf =
              TrustManagerFactory.
              getInstance(TrustManagerFactory.
                          getDefaultAlgorithm());

    tmf.init(trustedStore);
    SSLContext sc = SSLContext.getInstance("TLS");
    TrustManager[] trustManagers = tmf.getTrustManagers();
    KeyManager[] keyManagers = kmf.getKeyManagers();
    sc.init(keyManagers, trustManagers, null);

    // Creo el socket seguro del servidor
    SSLServerSocketFactory ssf = sc.getServerSocketFactory();
    serverSocket =
              (SSLServerSocket) ssf.createServerSocket(puerto);
}

public void start() {
    System.out.println("Servidor escuchando en el puerto " +
puerto);

    new Thread() {
        @Override
        public void run() {
            try {
                Socket aClient = serverSocket.accept();
                System.out.println("Cliente aceptado");
                aClient.setSoLinger(true, 1000);
                BufferedReader input = new BufferedReader(
                          new InputStreamReader(
                                      aClient.getInputStream()));
                String recibido = input.readLine();
                System.out.println("Recibido " + recibido);
```

```
                    PrintWriter output = new PrintWriter(
                              aClient.getOutputStream());
                    output.println("Hola, " + recibido);
                    output.flush();
                    aClient.close();
                }
                catch (IOException e) {
                    System.out.println("Error servidor → " +
                e.toString());
                }
            }
        }.start();
    }

}
```

En el cliente tenemos un proceso muy similar.

Aquí hemos creado una variable del tipo '**SSLSocket**' que será nuestro socket seguro.

En el constructor de nuestra clase del cliente seguro tenemos la **dirección del servidor** al que nos conectaremos y el **puerto** al que nos conectaremos.

Ahora, de igual forma que hicimos en el servidor, debemos obtener dónde están los certificados y las claves de confianza. En este caso, para nuestro cliente seguro.

Una vez hecho eso, creamos un cliente mediante el método 'createSocket'.

También hemos creado un método 'start' que será el que **inicie** la funcionalidad del **cliente**. En este método, conectaremos con un servidor seguro, obtendremos sus flujos para poder comunicarnos con él y le enviaremos un mensaje y recibiremos un mensaje.

SSLcliente.java

```
public class SSLcliente {

    private final String CLIENTPASS = "clientpass";
    private final SSLSocket client;

    public SSLcliente(
            String direccionservidor, int puerto) throws
            KeyStoreException, FileNotFoundException, IOException,
            NoSuchAlgorithmException, CertificateException,
            UnrecoverableKeyException, KeyManagementException
```

```
{
    // Indico los certificados seguros del cliente
    KeyStore keyStore = KeyStore.getInstance("JKS");
    keyStore.load(new FileInputStream(
            "src" + File.separator
            + "certificados" + File.separator
            + "cliente" + File.separator
            + "clientKey.jks"),
            CLIENTPASS.toCharArray());

    KeyManagerFactory kmf =
              KeyManagerFactory.getInstance(
                          KeyManagerFactory.getDefaultAlgorithm());
    kmf.init(keyStore, CLIENTPASS.toCharArray());
    KeyStore trustedStore = KeyStore.getInstance("JKS");
    trustedStore.load(new FileInputStream(
            "src" + File.separator
            + "certificados" + File.separator
            + "cliente" + File.separator
            + "clientTrustedCerts.jks"),
              CLIENTPASS.toCharArray());
    TrustManagerFactory tmf =
              TrustManagerFactory.getInstance(
                          TrustManagerFactory.getDefaultAlgorithm());

    tmf.init(trustedStore);
    SSLContext sc = SSLContext.getInstance("TLS");
    TrustManager[] trustManagers = tmf.getTrustManagers();
    KeyManager[] keyManagers = kmf.getKeyManagers();
    sc.init(keyManagers, trustManagers, null);

    // Creo el socket seguro del cliente
    SSLSocketFactory ssf = sc.getSocketFactory();
    client = (SSLSocket) ssf.createSocket(direccionservidor,
            puerto);
    client.startHandshake();
}

public void start() {
    System.out.println("Inicio cliente");
    new Thread() {
        @Override
        public void run() {
            try {
                Random aleatorio = new Random();
```

```
                    PrintWriter output = new PrintWriter(
                              client.getOutputStream());
                    output.println("Cliente: " +
                              aleatorio.nextInt(100));
                    output.flush();
                    BufferedReader input =
                              new BufferedReader(
                                        new InputStreamReader(

          client.getInputStream()));
                    String received = input.readLine();
                    System.out.println("Recibido: " +
                    received);
                    client.close();
               }
               catch (IOException e) {
                    System.out.println("Error cliente → " +
               e.toString());
               }
          }
        }.start();
    }
}
```

Para poder probarlo, deberemos tener un método 'main' en el que indiquemos el **puerto** en el que nos vamos a conectar, por ejemplo, el 5557.

Nos creamos un servidor y luego ya podremos crear tantos clientes como queramos, siendo esta comunicación una comunicación segura.

clase main

```
public static void main(String[] args) {
      int puerto = 5557;
      try
      {
          // Creo el servidor
          new SSLservidor(puerto).start();
          // Creo un cliente
          new SSLcliente("localhost", puerto).start();
      }
      catch (IOException | KeyManagementException | KeyStoreException
              | NoSuchAlgorithmException | UnrecoverableKeyException
              | CertificateException error)
      {
          System.out.println("Error --> " + error.toString());
      }
  }
```

Forma correcta de realizar una auditoría

Si se ha 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.

Deberíamos 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.

Ejemplo de acceso seguro a app web

El trabajo consiste en el desarrollo de una **aplicación web**, a la que únicamente se permite el **acceso de forma segura**, es decir, mediante la identificación de los usuarios.

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.

Reflexión

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.