Práctica 3
Comunicación segura con SSL

Eloy Anguiano

Oscar Delgado

Carlos García

Antonio Calleja

Elena Gómez

Fecha de entrega optativa: 20 de marzo a las 9:00.
Fecha de entrega obligatoria: lunes 8 de mayo a las 9:00.

1 Introducción a SSL

SSL (*Secure Sockets Layer*) es un protocolo de nivel de aplicación diseñado para permitir conexiones seguras sobre TCP, proporcionando confidencialidad e integridad a la comunicación. Asumiendo que el canal de transmisión es inseguro, la información se envía cifrada haciendo uso de algoritmos criptográficos. Es posible además utilizar SSL de forma transparente para otros protocolos de aplicación de nivel superior, que no notan ninguna diferencia respecto a la utilización de canales convencionales de comunicación no seguros.

En SSL los mensajes se transmiten utilizando cifrado de clave simétrica. El cliente y el servidor comparten una misma clave secreta que utilizan para cifrar y descifrar los datos. Al iniciar la comunicación, se acuerda y se intercambia dicha clave utilizando un protocolo de *handshake* basado en cifrado de clave pública o asimétrica. En dicho protocolo cada parte dispone de dos claves: una pública que se distribuye libremente y una privada que ha de mantenerse en secreto, de manera que los mensajes cifrados con la clave pública únicamente pueden ser descifrados utilizando la clave privada correspondiente y viceversa. De esta forma, el cliente y el servidor pueden acordar de manera segura una clave sin conocerse previamente, y sin riesgo de que ésta sea interceptada por terceros.

El mecanismo anterior, sin embargo, no impide a un atacante suplantar la identidad del servidor y engañar al cliente con una clave pública falsa. En la práctica, las claves públicas se intercambian junto con campos de identificación por medio de *certificados*. Cuando se inicia la comunicación, el cliente verifica dichos campos y utiliza la clave pública proporcionada en el certificado para cifrar un mensaje de prueba. Para que la conexión tenga éxito, el servidor ha de probar su identidad utilizando su clave privada para descifrar el mensaje y enviarlo de vuelta al cliente. De esta forma, un atacante debe disponer de la clave privada asociada al certificado enviado por el servidor para suplantarlo.

A pesar de lo anterior, un intruso aún podría interceptar, falsificar y enviar de vuelta al cliente un certificado que parezca legítimo, efectuando un ataque denominado *man-in-the-middle*, en el que se coloca de intermediario durante toda la comunicación. Por este motivo existen autoridades de certificación (*Certificate Authority, CA*, un ejemplo es *Verisign*). Estas son organizaciones externas que verifican los certificados y garantizan que las identidades asociadas a ellos son auténticas. Tras realizar exhaustivas comprobaciones, la CA valida un certificado mediante su *firma digital*. Esto es un resumen del certificado del servidor (obtenido por ejemplo mediante MD5[1]) cifrado con su propia clave privada. Si el cliente dispone de una lista de autoridades en las que confía, puede verificar que el certificado recibido es realmente del servidor descifrando el resumen con la clave pública del correspondiente CA y comprobar que es auténtico[2].

El esquema de funcionamiento aparece descrito en la figura inferior e involucra los siguientes pasos a modo de resumen:

1.– El servidor genera una clave privada y un certificado que solicita que sea firmado por una CA.

2.– El cliente se conecta con el servidor mediante una conexión TCP y envía un mensaje `Hello`. Este mensaje

[1] `http://es.wikipedia.org/wiki/MD5`

[2] Recordar que lo que ha sido cifrado con la clave privada solo puede ser descifrado con la clave pública y viceversa

contiene una lista del conjunto de cifrados soportados y la versión del protocolo SSL más alta permitida.

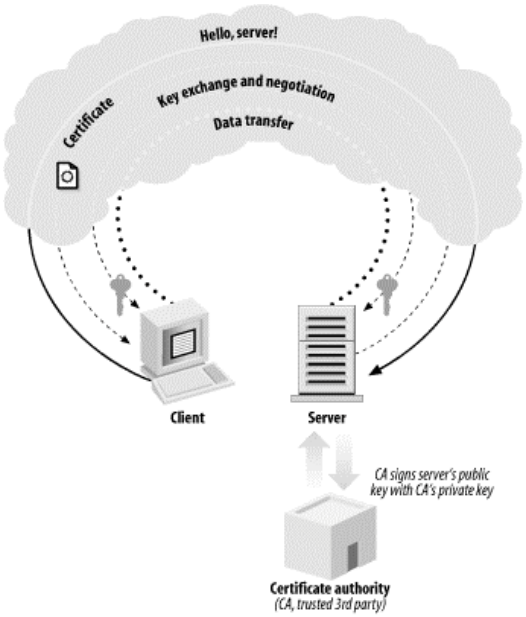
3.– El cliente recibe un registro `ServerHello`. En él, el servidor elige los parámetros de conexión a partir de las opciones ofertadas con anterioridad por el cliente y envía su certificado.

4.– Una vez recibida y validada la información previa por parte del cliente, éste crea una clave secreta para ser usada en el cifrado simétrico de datos que protegerá la comunicación. Esta clave es cifrada con la clave pública del servidor.

5.– El servidor usa su clave privada para descifrar la clave enviada por el cliente.

6.– A partir de este momento la comunicación se lleva a cabo utilizando cifrado simétrico mediante la clave intercambiada.

Para añadir un nivel mayor de seguridad es posible que el servidor también solicite al cliente que se autentifique. En este caso, el cliente deberá enviar también un certificado valido que haya sido firmado por una CA al servidor, y éste será verificado. Este será el esquema que utilizaremos en esta práctica.



2 OpenSSL

OpenSSL es una implementación *open source* del protocolo SSL que incluye tanto una API para desarrollo como una potente herramienta por línea de comandos. En esta sección se describen brevemente los pasos que ha de llevar a cabo una aplicación para iniciar la comunicación segura mediante SSL. Esta librería está disponible para su descarga en `http://www.openssl.org/` y en sistemas Linux puede ser instalada con el comando `sudo apt-get install libssl0.9.8`, y `apt-get install libssl-dev`, para el caso particular de la versión 0.9.8. Cualquier versión relativamente moderna será suficiente para llevar a cabo la práctica descrita en este documento.

Como punto a destacar, toda la documentación relativa a la interfaz de programación facilitada en la implementación OpenSSL está disponible online en la página `http://www.openssl.org/` y a través de las páginas de manual de `openssl(1)`, una vez instalados los correspondientes paquetes. La información facilitada es por lo tanto escasa con el objetivo de que ésta sea consultada a través de estos medios.

2.1 Creando una aplicación segura mediante SSL

El primer paso que ha de llevar a cabo cualquier aplicación que vaya a hacer uso de la capa SSL es llamar a la función *SSL_load_error_strings*. Esta función se encarga de /cargar mensajes de error que serán usados para informar al usuario de la aplicación en caso de producirse alguno. Cuando se produzca un error, se puede mostrar llamando a la función *ERR_print_errors_fp*. Posteriormente se llamará a la función *SSL_library_init*. Ésta se encarga de inicializar la librería SSL y registrar los métodos de cifrado soportados.

Inicializando el contexto

Tras estos dos pasos se llamará a la función *SSL_CTX_new* que crea un nuevo contexto para la utilización de la capa SSL y lo inicializa a los valores por defecto. Como argumento se le pasará un método de conexión. En nuestro caso será el valor devuelto por la función *SSLv23_method* que añade soporte para las versión SSL 2 y 3 tanto para clientes como servidores. El valor devuelto será guardado para ser utilizado en las siguientes llamadas, que se encargarán de inicializar correctamente el contexto.

Habrá que llamar a la función *SSL_CTX_load_verify_locations* que recibirá, además del contexto, la ruta al certificado de la CA que será utilizado para validar los certificados recibidos por la aplicación. Es importante resaltar que esta función solo ha de llamarse en caso de querer añadir entidades CA propias fuera de las ya preestablecidas. Posteriormente se llamará a la función *SSL_CTX_set_default_verify_paths* para utilizar los certificados de las entidades CA bien conocidas. Esto hará que nuestra aplicación pueda conectarse con servidores públicos que usen certificados firmados por estas CAs.

El siguiente paso es especificar el certificado que utilizará nuestra aplicación. Para ello se llamará a la función *SSL_CTX_use_certificate_chain_file*. También será necesario especificar la clave privada de nuestra aplicación llamando a *SSL_CTX_use_PrivateKey_file*. Como argumentos se suministrará la ruta al fichero que contiene la clave privada y se especificará que el formato del fichero es SSL_FILETYPE_PEM.

Por último se llamará a la función *SSL_CTX_set_verify*, que fijará el modo de funcionamiento de nuestra aplicación. Este modo será el mismo tanto para clientes como para servidores. Habrá que garantizar que se verifica la autenticidad del otro extremo del canal de comunicación y que se produce un fallo si el otro extremo no envía el correspondiente certificado. En nuestro caso no utilizaremos función de *callback*.

Obteniendo un canal de comunicación seguro

Tras inicializar el contexto, el siguiente paso que llevará a cabo una aplicación será obtener un canal de comunicación sobre el que utilizar la capa SSL. Esto se hará por ejemplo utilizando los sockets que hemos visto en prácticas anteriores. Una vez se haya obtenido dicho canal, representado mediante el descriptor de socket de una conexión TCP abierta, se deberá crear una estructura SSL que se utilizará para asegurar el canal. Para ello se llamará a la función *SSL_new* y se le pasará como argumento el contexto previamente inicializado. Un puntero a la estructura creada será guardado para añadir seguridad al canal. Este puntero se pasará en las siguientes llamadas.

El siguiente paso será asociar la estructura SSL creada el canal de comunicación. Para ello se llamará a la función *SSL_set_fd* y se le pasará como argumento el puntero a la estructura SSL y el descriptor de socket de la conexión TCP. Tras este paso el funcionamiento de clientes y servidores diferirá ligeramente.

En el caso de que nuestra aplicación sea un servidor deberá llamar a la función *SSL_accept* con el puntero a la estructura SSL. Esta función bloqueará al servidor y lo pondrá en espera hasta recibir el handshake por parte del cliente. En el caso de que nuestra aplicación sea un cliente deberá llamar a la función *SSL_connect* que inicializará el handshake con el servidor. Si ambas funciones no devuelven error significará que el handshake se habrá llevado a cabo sin errores. Sin embargo, esto no implica que ya podamos confiar en el otro extremo.

El siguiente paso será comprobar que el certificado del otro extremo es correcto, tanto para el cliente como para el servidor. Para ello se llamará primero a la función *SSL_get_peer_certificate* para comprobar que de hecho se ha recibido un certificado. Parece increíble pero este paso es necesario ya que si el servidor no envía certificado alguno, el cliente lo dará por válido automáticamente. Tras comprobar que de hecho se ha recibido un certificado y éste ha sido liberado, se llamará a *SSL_get_verify_result* que intentará validar el certificado con algún CA. Sólo si el certificado es válido el canal se considerará seguro.

Enviando y recibiendo mensajes

Una vez el canal de comunicación se ha asegurado es posible realizar el envío y la lectura de datos a través de él. Para ello se utilizarán las funciones *SSL_write* y *SSL_read*. Estas funciones son equivalentes a las funciones send y recv que hemos visto en prácticas anteriores.

Cerrando el canal seguro

Una vez la aplicación no vaya a realizar más envíos o recepción de mensajes deberá cerrar el canal de comunicación y liberar las estructuras creadas de forma dinámica. Para ello habrá que llamar a *SSL_shutdown*, *SSL_free* y *SSL_CTX_free*. Finalmente se cerrará el descriptor de socket que se estaba utilizando para enviar y recibir mensajes.

2.2 Creación de los certificados y claves necesarias

En esta sección se describe el uso de la herramienta `openssl(1)` para generar los certificados y las claves necesarias para establecer una comunicación segura. En esta práctica simularemos una CA que firmará y validará los certificados del servidor y de los clientes.

Creación del certificado raíz para la CA

En primer lugar crearemos un certificado raíz para la CA que se encargará de firmar todos los certificados, de forma que el cliente pueda verificar la identidad del servidor y viceversa. El primer paso es generar la clave privada RSA que la CA utilizará para firmar los demás certificados. Para ello empleamos el comando `genrsa` de `openssl`:

```
openssl genrsa -out rootkey.pem 2048
```

En este caso la clave generada no se guardará cifrada en disco. El último parámetro hace referencia a la longitud de la clave, 2048 bits en este caso. Como toda clave privada, el fichero resultante ha de mantenerse

siempre en secreto. Una vez disponemos de la misma, creamos un certificado auto-firmado con el comando `req`:

```
openssl req -new -x509 -key rootkey.pem -out rootcert.pem
```

El comando `req` sirve para crear solicitudes de certificado. Las CA investigan y, si procede, firman estas solicitudes para generar así certificados válidos. En el caso del certificado raíz, la misma CA valida su propio certificado con la opción `-x509`. Son las propias funciones `SSL_accept()` y `SSL_connect()` las encargadas de enviar los certificados al producirse el *handshake*. Tanto el certificado como la clave pueden combinarse en un único fichero que, por supuesto, ha de mantenerse en secreto.

```
cat rootcert.pem rootkey.pem > root.pem
```

Por último, comprobamos que los campos con información del emisor y del propietario del certificado coinciden.

```
openssl x509 -subject -issuer -noout -in root.pem
```

Creación del certificado del cliente

Para obtener un certificado asociado al cliente es nuevamente necesario generar una clave privada específica para ese cliente y, a partir de ésta, una solicitud de firma de certificado. A diferencia de la CA, el cliente y el servidor no firman sus propios certificados, puesto que cada uno sólo confía en la autoridad certificadora para autenticar al otro extremo. En este caso, será necesario llamar al comando `openssl req` para generar una solicitud propiamente dicha en vez de un certificado auto-firmado. Como opciones se especificarán la clave privada del cliente y el nombre del fichero generado para la solicitud. Estas opciones se pueden consultar en la página del manual del comando *req*.

La solicitud de certificado se hace llegar a la CA para que genere el certificado del cliente incluyendo su firma digital. Para ello deberemos pasar el fichero con la solicitud al comando `openssl x509`, junto con el certificado y la clave privada de la CA para que sea usada en la firma. Especificaremos que el fichero de entrada es una solicitud y no un certificado completo, que debe crearse el fichero con el número de serie de la CA, y también el nombre del fichero de salida. Se debe consultar el man del commando `x509` o la documentación *online* para identificar las opciones correspondientes.

Finalmente, combinamos el certificado generado para el cliente, la clave privada del cliente y el certificado raíz de la CA en un único fichero `client.pem` de forma similar a como se ha hecho anteriormente con el comando cat. Comprobamos que ahora el emisor y el propietario del certificado no coinciden:

```
openssl x509 -subject -issuer -noout -in client.pem
```

Nota: Es importante que los datos facilitados en la petición de firma de certificado del cliente sean distintos de los usados para crear el certificado de la CA. Si no, los certificados no se considerarán válidos por la capa SSL.

Nota: Para el autocorrector de la práctica, los certificados creados deben tener determinados valores en sus atributos. Concretamente, el atributo *Common Name* del certificado de la CA debe ser *Redes2 CA*, *G-CCCC-NN-P1-client* para el del cliente, y *G-CCCC-NN-P1-server* para el del servidor. Así, para una pareja que asista a la clase 2363 y sea la número 8, el *Common Name* de los certificados de cliente y servidor debe ser *G-2363-08-P3-client* y *G-2363-08-P3-server*, respectivamente. Más información en el manual del corrector C3PO.

Creación del certificado del servidor

El proceso de creación del certificado correspondiente al servidor es totalmente análogo al del cliente: creación de una clave privada RSA, generación de una solicitud de firma de certificado, firma de la solicitud por parte de la CA, y concatenación de los ficheros. Nuevamente, es posible comprobar los datos del certificado a través del comando `openssl x509`.

El *makefile* de esta práctica deberá incluir una regla `certificados` que utilice los distintos comandos de `openssl` para generar todos los certificados en una carpeta llamada `cert`.

3 Extensión de la librería para soportar canales seguros

El objetivo de esta práctica es añadir funcionalidades adicionales a la librería con la que se ha venido trabajando en las prácticas 1 y 2. En particular, se añadirán una serie de funciones que permitan gestionar la comunicación mediante canales seguros. Las funciones a añadir serán las descritas a continuación. Los argumentos y los valores devueltos se dejan a elección de los alumnos. Sin embargo, se valorará negativamente un diseño incorrecto de las mismas.

- *inicializar_nivel_SSL*: Esta función se encargará de realizar todas las llamadas necesarias para que la aplicación pueda usar la capa segura SSL.
- *fijar_contexto_SSL*: Esta función se encargará de inicializar correctamente el contexto que será utilizado para la creación de canales seguros mediante SSL. Deberá recibir información sobre las rutas a los certificados y claves con los que vaya a trabajar la aplicación.
- *conectar_canal_seguro_SSL*: Dado un contexto SSL y un descriptor de socket esta función se encargará de obtener un canal seguro SSL iniciando el proceso de handshake con el otro extremo.
- *aceptar_canal_seguro_SSL*: Dado un contexto SSL y un descriptor de socket esta función se encargará de bloquear la aplicación, que se quedará esperando hasta recibir un handshake por parte del cliente.
- *evaluar_post_connectar_SSL*: Esta función comprobará una vez realizado el handshake que el canal de comunicación se puede considerar seguro.
- *enviar_datos_SSL*: Esta función será el equivalente a la función de envío de mensajes que se realizó en la práctica 1, pero será utilizada para enviar datos a través del canal seguro. Es importante que sea genérica y pueda ser utilizada independientemente de los datos que se vayan a enviar.
- *recibir_datos_SSL*: Esta función será el equivalente a la función de lectura de mensajes que se realizó en la práctica 1, pero será utilizada para enviar datos a través del canal seguro. Es importante que sea genérica y pueda ser utilizada independientemente de los datos que se vayan a recibir.
- *cerrar_canal_SSL*: Esta función liberará todos los recursos y cerrará el canal de comunicación seguro creado previamente.

Todas las funciones deberán estar documentadas y tener una página man correspondiente tal y como se ha realizado en las prácticas anteriores.

Utilizando estas funciones se deberá codificar un servidor de *echo* que sólo acepte conexiones seguras cuyo certificado haya sido firmado por una autoridad CA (en este caso la autoridad CA simulada en la sección anterior). **Tanto el cliente del servicio como el servidor requerirán el certificado del otro extremo para verificar la conexión**. El servidor se limiatará a devolver los datos enviados por el cliente y el cliente deberá imprimir por pantalla los datos recibidos. Los certificados a utilizar se pasarán como argumentos a cada programa, tanto cliente como servidor. Se deberá comprobar que si el certificado utilizado no está firmado por una autoridad CA competente la conexión no será válida.

3.1 Modificación de IRC para soportar canales seguros

Utilizando las funciones descritas anteriormente se debe aportar la funcionalidad necesaria para que el servidor IRC de la práctica 1 acepte también conexiones seguras en el puerto 6697, y para que funcione la opción SSL en el cliente IRC de la práctica 2.

4 Implementación y Evaluación

Los elementos que deben implementarse en esta práctica son los siguientes:

- (*Hasta 2 ptos.*) Creación de los certificados digitales de la CA, cliente y servidor.
- (*Hasta 2'5 ptos.*) Implementación de la funcionalidad SSL en una librería estática propia. Uso de la librería anterior en un modelo simple de cliente/servidor de "eco" seguro.
- (*Hasta 2'5 ptos.*) Extensión del cliente y servidor IRC de las prácticas anteriores para que puedan utilizar SSL en su conexión.

Los 3 puntos restantes se reparten, al 50%, entre el *estilo de codificación* y la *documentación*, para la que se pide una breve memoria en PDF, código con comentarios adecuados, así como las páginas man de las funciones SSL pedidas.

5 Bibliografía recomendada

Una explicación más detallada de OpenSSL, aunque no muy actualizada, puede encontrarse en *Network security with OpenSSL* de J. Viega, M. Messier y P. Chandra (O'Reilly). Incluye una descripción exhaustiva de todas las opciones de la herramienta por línea de comandos `openssl`. Otra opción es *SSL and TLS: Designing and Building Secure Systems* de Eric Rescorla, publicado por Addison-Wesley. Ambos están disponibles en la biblioteca. La primera referencia está disponible online en el enlace:
`http://proquest.safaribooksonline.com/book/networking/security/059600270x`