# Práctica 3
# Comunicación segura mediante openSSl

## Introducción

El objetivo de esta tercera practica es implementar una capa de seguridad en el cliente y en el servidor. Para ello implementaremos un modulo de alto nivel a partir de las funciones de openSSL y posteriormente desarrollaremos un cliente y un servidor echo que tengan comunicación segura. Una vez comprobemos mediante c3po su buen funcionamiento, podremos implementar la capa de seguridad tanto en el servidor de la practica 1 como en el cliente de la practica 2.

## Creación de los certificados

Siguiendo las indicaciones de Moodle hemos creado una regla en el makefile que permite la creación de todos los certificados. Este regla se llama “certificados”. Tras ejecutarla, se crean directamente los certificados en la carpeta certs.

Para realizarla no solo basto con ver los ejemplos, tuvimos que entenderlos y adaptarlos a los resiquitos solicitados.

## Diseño

### Implementación del modulo SSL de alto nivel.

Aquí dejamos una breve descripción del procedimiento seguido por las funciones. Todo esta comentado con mas detalle en el código.

• inicializar_nivel_SSL: tras cargar los posibles errores devueltos por el SSL, inicializa la librería y por ultimo fija el contexto con el certificado de la CA y el del Cliente/Servidor.

• fijar_contexto_SSL:  crea un nuevo contexto y lo inicializa con el certificado de la CA y el del cliente/servidor.

• conectar_canal_seguro_SSL: Crea una nueva conexión ssl con el contexto y el socket pasados como argumento y tras realizar el handsake mediante SSL_connect devuelve la estructura SSL.

• 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: Comprueba que la estructura SSL pasada como argumento constituye un canal seguro.

• enviar_datos_SSL:

parámetros de entrada:
SSL *ssl
char *buffer
int nbytes
La función llama simplemente llama a SSL_write(ssl, (void*)buffer, nbytes) y devuelve el resultado.

• recibir_datos_SSL:
parámetros de entrada:
SSL *ssl
char *buffer
int nbytes
La función llama simplemente llama a SSL_read(ssl, (void*)buffer, nbytes) y devuelve el resultado.

• cerrar_canal_SSL: La función libera la estructura SSL y el Contexto pasados como argumentos.

**Creación de un servidor echo con SSL.**

La implementación del servidor echo se realiza siguiendo la siguiente secuencia de llamadas:

Primero inicializamos el nivel SSL con el certificado de la CA y el del Servidor que se crean al ejecutar make certificados.

A continuación abrimos un socket en el puerto 6667 (realizando bind y accept). Este socket lo hemos cogido dado que no se especificaba otro y el puerto destinado a echo se encuentra por debajo del 1024 y por tanto necesita permisos de super-usuario.

Tras esto, llamamos a **aceptar_canal_seguro_SSL** con los retornos de los dos pasos anteriores. Y comprobamos que la conexión este bien establecida mediante **evaluar_post_connectar_SSL.**

Una vez aquí, el servidor echo solo necesita recibir con SSL (mediante **recibir_datos_SSL**) y reenviarlos con **enviar_datos_SSL**.

Tal y como se especifica el servidor deja de hacer echo cuando recibe exit. Tras esto simplemente cierra el canal SSL y los sockets del cliente y el servidor.

**Creación de un cliente echo con SSL.**

La implementación del cliente sigue la misma estructura. Los pasos son los siguentes:

Primero inicializamos el nivel SSL con el certificado de la CA y el del Cliente que se crean al ejecutar make certificados.

A continuación conectamos el cliente al socket 6667. Hemos elegido este puerto por un mero criterio de familiariadad dado que el puerto de echo se encuentra por debajo de 1024.

Tras esto, llamamos a **conectar_canal_seguro_SSL** con los retornos de los dos pasos anteriores. Y comprobamos que la conexión este bien establecida mediante **evaluar_post_connectar_SSL.**

Una vez aquí, el servidor echo solo necesita enviar con SSL (mediante **enviar_datos_SSL**) los datos escritos por teclado y recibirlos con **recibir_datos_SSL**.

Tal y como se especifica el cliente deja de enviar datos tras enviar “exit”. Tras esto simplemente cierra el canal SSL y el socket de comunicación con el servidor.

**Implementación del servidor_IRC y cliente_IRC con envio seguro.**

Para la implementación de la capa SSL deben de implementarse los “TODOS” que dejamos en el servidor y en el cliente.

En el caso del servidor implementamos el flag de SSL(--port, --ssl,  -s).

Cuando recibimos - -port X, inicializamos el servidor SSL como se explica a continuacion en el puerto X.
Cuando recibimos - -ssl o -s, inicializamos un nivel SSL, preparamos el socket 6669 y lo preparamos todo para que el servidor, mediante un hilo, acepte conexiones seguras. Para ellos, en bucle cada vez que recibimos una nueva conexión segura  aceptamos el canal seguro mediante **aceptar_canal_seguro_SSL ,** y tas comprobar el buen estado de la conexión (con **evaluar_post_connectar_SSL**) permitimos que el cliente sea atendido por el servidor( y pueda loguearse).

En el caso del cliente implementamos flags de - -port que permite elegir el puerto de conexión y la bandera -ssl, en este caso se realiza el proceso que ya hemos visto varias veces antes, inicializamos el nivel SSL y abrimos el puerto correspondiente(6669 por defecto o el descrito en - -port).
A continuación conectamos el cliente a localhost y dicho puerto y habilitamos el canal seguro con **conectar_canal_seguro_SSL**  y **evaluar_post_connectar_SSL.**

Todo este proceso habilita a los modulos a utilizar SSL pero ademas de esto es necesario que el envío sea mediante SSL. Para ello nos hemos definidos dos funciones **sendData** y **recvData** que permiten enviar datos tanto con la capa SSL como sin ella. Esto no permite sustituir las llamadas de send y receive permitiendo que el mismo código se encargue de las conexiones SSL como las no seguras.

**NOTA:** El servidor al ser ejecutado en modo seguro no esta deamonizado. Esta decisión la hemos tomado al ver que el proceso de daemonizar mueve el proceso bajo init. Esto hacia inviable cagar los certificados dado que los teniamos por direccionamiento relativo.

## Conclusiones técnicas

Técnicamente ha sido la mas fácil de las 3, dado que el enunciado y los ejemplos proporcionados eran de gran ayuda. Sin embargo destacamos que en la practica era necesario entender perfectamente los conceptos que se estaban aplicando, dado que si te saltabas algún paso no funcionaba el cifrado.

## Conclusiones personales

La practica nos ha resultado muy interesante. Sobre todo por que ahora mismo hay un buena tendencia a cifrar las comunicaciones y esta bien que aprendamos a realizar algo tan importante.

Además consideramos que es una practica algo mas cercana a la teoría. Esto lo consideramos altamente positivo ya que la practica nos ha ayudado a entender la teoría y la teoría nos sirve para ver  por que las cosas funcionan como lo hacen en la práctica.