2.9. Transport Layer Security (TLS)

Transport Layer Security (TLS), o *Secure Sockets Layer* (SSL) como se llamaban las primeras versiones del protocolo, es el protocolo seguro más importante de Internet. Es un protocolo cliente-servidor que funciona encima de TCP (mientras que DTLS [RFC6347] funciona sobre UDP), garantiza la confidencialidad e integridad de las comunicaciones que la usan, y permite autenticar mediante certificados digitales al servidor (y opcionalmente al cliente), por lo que también evita ataques de *Adversary-in-the-Middle* (AitM) si se utiliza una lista de autoridades de certificación confiable.

TLS ha tenido múltiples versiones (SSL 2.0, SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2 y TLS 1.3), que han ido resolviendo los problemas de seguridad de las versiones anteriores, y añadiendo algoritmos criptográficos cada vez más seguros. Actualmente las últimas dos versiones de TLS (v1.2 [RFC5246] y v1.3 [RFC8446]) se consideran seguras, y la única forma de descifrar el tráfico intercambiado es mediante la colaboración de alguno de los extremos de la comunicación.

TLS es un protocolo bastante complejo, que de hecho tiene varios protocolos o capas internas. La *Record Layer* es la capa inferior de TLS y se encarga de delimitar los mensajes que envían los diferentes protocolos de TLS en estructuras de tipo TLV (Tipo-Longitud-Valor), y que también incluyen la versión de TLS en uso. Esta capa es la que se encarga de cifrar y garantizar la integridad de los datos mediante códigos de autenticación (HMAC). Para ello utiliza un conjunto de claves de cifrado y de firmado, que son diferentes en cada sentido y se negocian de nuevo en cada sesión TLS.

Por encima de la *Record Layer* se encuentran los cuatro protocolos de TLS: el *Change Cipher Spec Protocol*, el *Alert Protocol*, el *Handshake Protocol*, y el *Application Data Protocol*. Los dos primeros son muy sencillos, puesto que se limitan a un único mensaje: que sirve para indicar que a partir de ese momento se van a cifrar todos los datos que envíe la *Record Layer* (*Change Cipher Spec*), y para notificar problemas

en la sesión TLS (*Alert*), lo que suele implicar la finalización de la misma. El *Application Data Protocol* no es más que el protocolo de nivel de aplicación que envía datos mediante TLS. Así que el protocolo más importante de TLS es el *Handshake Protocol*, que es el que se emplea para negociar todas las claves que usa la *Record Layer* y que se ejecuta tan pronto como se completa el *3-way handshake* de la conexión TCP subyacente.

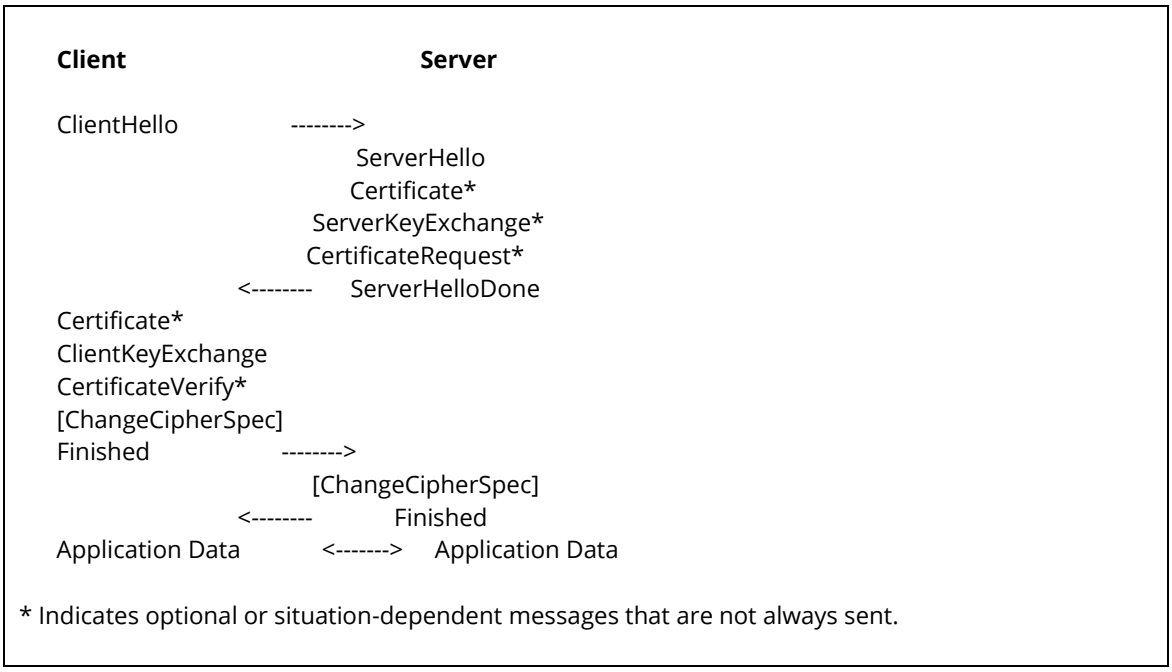

Figura 2.17 - Establecimiento (*handshake*) de una sesión TLSv1.2 [RFC5246]

La **Figura 2.17** muestra todos los mensajes que se pueden llegar a enviar en un *handshake* completo de una sesión TLS v1.2:

- ***ClientHello***: Este mensaje contiene la versión de TLS más alta que implementa el cliente (e.g. v1.2), así como la lista de todos los *ciphersuites* y extensiones que soporta. Un ***ciphersuite*** TLS (e.g. TLS_DHE_RSA_WITH_AES_128_GCM_SHA256) es un conjunto de algoritmos criptográficos que se emplean para negociar las claves (e.g. DHE), firmar digitalmente (e.g. RSA), cifrar (e.g. AES_128_GCM) y como función hash (e.g. SHA256) de las firmas digitales y HMAC. Las extensiones TLS permiten añadir funcionalidades opcionales al protocolo, tales como el uso de algoritmos de curva elíptica para el intercambio de claves o de firma digital. En particular, hay dos extensiones muy importantes para la captura y análisis de sesiones TLS: la extensión ***Server Name Indication*** **(SNI)** y la extensión ***Aplication Layer Protocol Negotation*** **(ALPN)**, que indican, respectivamente, el nombre DNS del servidor TLS con el que se quiere conectar el cliente (por si hay varios servidores virtuales compartiendo la misma dirección IP), y los protocolos de

nivel de aplicación soportados por el cliente. La importancia de estas extensiones radica en que el mensaje *ClientHello* no está cifrado, por lo que todos sus campos se envían en texto claro, y es posible conocer el servidor TLS con el que se quiere comunicar un cliente TLS y el protocolo de aplicación superior, incluso cuando todos los datos intercambiados por el nivel de aplicación estén cifrados. El conjunto de *ciphersuites* y extensiones soportadas también permite intentar identificar la librería criptográfica que emplea el cliente (e.g. OpenSSL 1.1.1).

- ***ServerHello***: Cuando el servidor TLS recibe el mensaje *ClientHello* busca los *ciphersuites* y extensiones del cliente que también soporte el servidor, elige las más seguras, y envía este mensaje para indicar al cliente qué versión de TLS, *ciphersuite* y extensiones se van a emplear en esa sesión TLS.
- ***Certificate***: Este mensaje contiene el certificado X.509 del servidor TLS, incluyendo los certificados de todas las autoridades de certificación (CAs) intermedias hasta llegar (sin incluir) a una de las autoridades de certificación raíz de las que se fía el cliente. De forma que éste pueda comprobar la validez de toda la cadena de certificados, y por tanto fiarse de la identidad del servidor (porque está validada por una CA confiable).
- ***CertificateRequest***: Además de autenticar al servidor, TLS permite, de manera opcional, autenticar al cliente mediante otro Certificado. Este mensaje sirve para solicitar al cliente que envíe al servidor su certificado en un mensaje *Certificate*, indicando las autoridades de certificación (CAs) de las que se fía el servidor.
- ***ServerKeyExchange***: Las primeras versiones de SSL y TLS utilizaban la clave RSA pública del certificado del servidor para implementar el intercambio de claves. El cliente simplemente generaba de manera aleatoria un secreto maestro (denominado ***Pre-Master Secret Key***), del que se derivan el resto de claves de la sesión SSL/TLS, y lo cifraba con la clave pública RSA del servidor. De esta forma además se comprobaba que el servidor tuviese la clave privada asociada, porque esa era la única forma de obtener el secreto maestro y por tanto las claves de sesión necesarias para descifrar el tráfico. El problema de este mecanismo de intercambio de claves es que, si en el futuro alguien consigue comprometer la clave privada del servidor, sería capaz de descifrar todo el tráfico que cualquier cliente se ha intercambiado con el servidor utilizando ese certificado. Así que en las últimas versiones de TLS (y obligatoriamente en TLSv1.3) la generación del *Pre-Master Secret Key* se puede realizar mediante un protocolo de intercambio de claves Diffie-Hellman Efímero (DHE) o basado en curva elíptica (ECDH). Este mensaje envía los parámetros de DHE o ECDH del servidor al cliente, que deben estar firmados con la clave privada del servidor para demostrar la posesión del certificado (el cliente comprueba la firma con la clave pública del certificado).

- ***ServerHelloDone***: Una vez que el servidor termina de enviar sus mensajes (*ServerHello*, *Certificate* y opcionalmente *CertificateRequest* y/o *ServerKeyExchange*), utiliza este mensaje para indicar al cliente que es su turno.
- ***ClientKeyExchange***: Este mensaje completa el intercambio de claves con el servidor, bien enviando el *Pre-Master Secret Key* cifrado con la clave pública que aparece en el certificado del Servidor, o bien enviando los parámetros DHE o ECDHE del cliente.
- ***Certificate*** y ***CertificateVerify***: Cuando el servidor solicita al cliente un certificado con el mensaje *CertificateRequest*, el cliente debe enviar su certificado en un mensaje *Certificate*, que debe estar firmado por alguna de las CAs indicadas por el servidor o incluir una cadena de certificados hasta alguna de ellas. El mensaje *CertificateVerify* está firmado digitalmente con la clave privada del cliente, para demostrar así su posesión.
- ***ChangeCipherSpec***: En cuanto el cliente genera el *Pre-Master Secret Key* (*ClientKeyExchange*) o lo negocia mediante DHE o ECDHE (*ServerKeyExchange*), puede generar todas las claves que necesita la *Record Layer* para cifrar y autenticar sus mensajes. Así que el cliente utiliza el mensaje *ChangeCipherSpec* (aunque formalmente no forma parte del *Handshake Protocol*) para indicar que, a partir de ese momento, todos los mensajes que enviará el cliente están cifrados con las claves negociadas. Del mismo modo, en cuanto el Servidor recibe el *ClientKeyExchange* del cliente, es capaz de obtener el *Pre-Master Secret* y de ahí todas las claves de cifrado y autenticación de la sesión, de modo que también utiliza el mensaje *ChangeCipherSpec* para indicar que a partir de ese momento todos sus mensajes también van a estar cifrados con esas claves.
- ***Finished***: Estos son los primeros mensajes de la sesión TLS que están cifrados y autenticados. Eso quiere decir que todos los mensajes anteriores se transmiten en claro y podrían haber sido modificados por un atacante (e.g. para negociar una versión de SSL vulnerable). El mensaje *Finished* se utiliza para enviar una suma de comprobación de todos los mensajes del protocolo *Handshake* que se han recibido (que no puede ser modificada ya que el mensaje *Finished* está autenticado) de forma que el otro extremo compruebe que coincide exactamente con los mensajes que ha enviado él y por tanto que nadie los ha modificado.

Una vez que ambos extremos han comprobado sus mensajes *Finished*, se establece la sesión TLS y puede empezar el intercambio seguro de datos de nivel de aplicación (e.g. HTTP).

El problema de la negociación TLS es que es bastante largo (2x RTT), requiere transmitir uno o más certificados (que pueden ocupar varios Kilobytes) y el uso de

criptografía asimétrica para cifrar, firmar y/o negociar claves. Así que TLS también define un mecanismo de negociación de sesión TLS rápido, denominado *Session Resumption*.

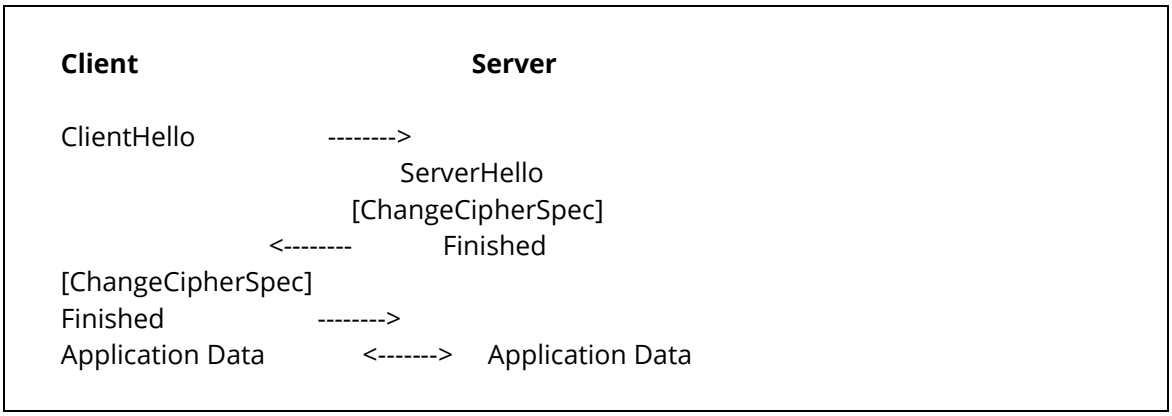

Figura 2.18 – Resumen de una sesión TLSv1.2 [RFC5246]

La idea del TLS *Session Resumption* es reutilizar las claves simétricas ya negociadas en una sesión TLS anterior, de forma que se pueda reducir mucho el tiempo y la computación necesaria para el completar el protocolo de *Handshake*. Existen un par de formas para implementar el *Session Resumption*. En la primera, tanto el cliente como el servidor almacenan las claves negociadas anteriormente y las asocian con un *SessionID* que establece el servidor en su mensaje *ServerHello*. De forma que cuando el cliente vuelva a conectarse con el servidor, envía ese *SessionID* en su mensaje *ClientHello* para indicar que desea resumir esa sesión TLS. Si el servidor todavía se acuerda de las claves de esa sesión, responde con un mensaje *ServerHello* con el mismo valor de *SessionID* y, como no es necesario autenticarse frente al cliente (porque ya lo hizo en la sesión anterior) ni negociar claves de sesión, envía inmediatamente el mensaje *ChangeCipherSpec,* y empieza a utilizar las claves anteriores para cifrar y firmar el mensaje *Finished* que sirve para comprobar que los mensajes *ClientHello* y *ServerHello* no han sido alterados. El cliente hace exactamente lo mismo (incluso si el servidor requiere la autenticación del cliente mediante certificado), por lo que el *Handshake* de la sesión resumida se limita a 1.5 RTTs y no hay que enviar ningún certificado ni utilizar ninguna operación de criptografía asimétrica. Si el servidor no se acuerda de las claves del *SessionID* solicitado por el cliente, simplemente genera un nuevo *SessionID* y se ejecuta un *Handshake* TLS completo.

La única pega de este mecanismo de resumen mediante *SessionID* es que requiere que los servidores almacenen las claves de sesión de todos sus clientes, y sobre todo que sean capaces de recuperarlas cuando el cliente lo solicite. Lo cual es un problema en granjas de servidores, donde el cliente puede contactar con un servidor diferente. Por esa razón se creó un nuevo mensaje de servidor, denominado *NewSessionTicket* [RFC5077], que actúa como una especie de *Cookie*

HTTP para almacenar información cifrada en el cliente, y una extensión para los mensajes *ClientHello* y *ServerHello* para indicar que ambos extremos soportan este mecanismo. El Mensaje *NewSessionTicket* lo envía el servidor al cliente justo antes de su mensaje *ChangeCipherSpec* (**Figura 2.19.A**) y contiene las claves de sesión que acaba de negociar con el cliente (i.e. *ClientKey Exchange*), cifradas con una clave simétrica que solo conocen él y resto de servidores de la granja.

De esta forma no hace falta que el servidor se acuerde de las claves de sesión de ese cliente, porque la próxima vez que el cliente se conecte al servidor y quiera resumir la sesión (**Figura 2.19.B**), solo tiene que incluir el valor del *NewSessionTicket* que recibió del servidor en la extensión *SessionTicket* de su *ClientHello*. Así que el Servidor puede descifrar las claves de sesión con su clave secreta y resumir la sesión como si se utilizase el mecanismo de *SessionID* básico, aunque puede volver a enviar un mensaje *NewSessionTicket* para renovar el estado del cliente.

A) **Client** **Server**

ClientHello
(SessionTicket extension) -------->
ServerHello
(empty SessionTicket extension)
Certificate*
ServerKeyExchange*
CertificateRequest*
<-------- ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished -------->
NewSessionTicket
[ChangeCipherSpec]
<-------- Finished
Application Data <-------> Application Data

B) **Client** **Server**

ClientHello
(SessionTicket extension) -------->
ServerHello
(empty SessionTicket extension)
NewSessionTicket
[ChangeCipherSpec]
<-------- Finished

```
    [ChangeCipherSpec]
    Finished                -------->
    Application Data        <-------->  Application Data
```

Figura 2.19 – A) Negociación y B) Resumen de una sesión TLSv1.2 con *SessionTicket* [RFC5077]

Como TLS trabaja por encima de TCP, es capaz de proteger los datos de aplicación, pero no la conexión TCP subyacente. Por lo tanto, un atacante que sea capaz de observar los números de secuencia de la conexión TCP, puede cerrarla prematuramente falsificando un TCP FIN o un RST, lo que truncaría la sesión TLS. Para evitar este ataque, y señalizar que la sesión TLS se ha completado correctamente antes de cerrar la sesión TLS, las implementaciones de TLS generan un mensaje de *Alert* cifrado con el código de error *Close Notify*, justo antes de cerrar la conexión subyacente. Así que, aunque sea contraintuitivo, las sesiones TLS que terminan correctamente deben incluir un mensaje *Alert* justo antes de cerrar la conexión TCP subyacente (que por otro lado es lo que ocurriría si se detecta un error grave en la sesión TLS).