Universidad de Buenos Aires

Facultades de Ciencias Económicas,
Ciencias Exactas y Naturales e Ingeniería

Carrera de Especialización en Seguridad Informática

TRABAJO FINAL DE ESPECIALIZACION

Tema:

"La era de TLS 1.3"

Autor: Lic. Carlos Quiroga

Tutor: Mg. Lic. Juan Devincenzi

Marzo de 2022 - Cohorte 2020

Declaración Jurada

Por medio de la presente, el autor manifiesta conocer y aceptar el Reglamento de Trabajos Finales vigente y se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual.

FIRMADO

Carlos Alberto Quiroga Juncos

DNI 18.317.942

Resumen

El protocolo criptográfico denominado TLS *(Transport Layer Security)*, y anteriormente conocido como SSL *(Secure Sockets Layer)*, es hoy el mecanismo mediante el cual es posible realizar comunicaciones seguras a través de una red, principalmente Internet.

Su aplicación primordial sobre la red Internet, incluye a todas las transacciones comerciales y de información sensible que ocurren a diario en el planeta, en modo cliente / servidor.

Su destacada función no solo se ciñe a la navegación segura. Otras aplicaciones abarcan al correo electrónico, la mensajería, la voz sobre IP, y a las comunicaciones entre servicios *web*.

Debido a la complejidad que supone su actualización, en el presente año 2022 conviven en el ciberespacio varias versiones de TLS. Y la más reciente y segura todavía no se encuentra generalmente adoptada.

Este trabajo tiene como finalidad el abordar esta problemática presentando una guía clara y actualizada sobre la evolución de su funcionamiento, desde la concepción original como SSL 3.0, hasta el nuevo estándar vigente TLS 1.3 de 2018, y presentar conclusiones sobre el estado actual de esta migración aún en curso.

Palabras Clave:

Protocolo criptográfico, SSL, TLS, HTTPS, 443.

Contenido

Introducción 1

1. Protocolo SSL 3.0 3

1.1 Visión General 4

1.1.1 Sesiones y Conexiones 6

1.1.2 La Máquina de Estados 6

1.1.3 Conjuntos de Cifrado 8

1.1.4 Intercambio y Generación de Claves 9

1.2 Subprotocolo *Record* 12

1.2.1 Fragmentación 13

1.2.2 Compresión 14

1.2.3 Autentificación 14

1.2.4 Cifrado 15

1.3 Subprotocolo *Handshake* 16

1.3.1 Reanudación y Renegociación 19

1.3.2 Los Mensajes 21

1.4 Subprotocolo *Change Cipher Spec* 46

1.5 Subprotocolo *Alert* 47

1.6 Subprotocolo *Application Data* 50

2. Protocolo TLS 1.2 53

2.1 Sesiones y Conexiones 54

2.2 Generación de Claves 56

2.3 Los Registros en Línea 58

2.4 Conjuntos de Cifrado 59

2.2.1 Cambios en TLS 1.0 59

2.2.2 Cambios en TLS 1.1 61

2.2.3 Cambios en TLS 1.2 62

2.5 Extensiones 67

2.6 Más Extensiones 75

2.7 Otros Cambios 88

2.7.1 Certificados 88

2.7.2 Mensajes de Alerta 89

2.7.3 Varios 89

3.Protocolo TLS 1.3 91
3.1 Principales Cambios 93
3.2 Conjuntos de Cifrado 94
3.3 *Performance* 96
3.3.1 Modo 1-RTT 96
3.3.2 *Key Schedule* 99
3.3.3 Reanudación 101
3.3.4 Modo 0-RTT 104
3.4 Reseña Final 107
3.4.1 Conjuntos de Cifrado 107
3.4.1 0-RTT 108
3.4.2 *Forward Secrecy* 108
3.4.3 Cifrado del *Handshake* 109
3.4.4 HDKF 110
3.4.5 Máquina de Estados 110
3.4.6 Curvas Elípticas y Algoritmos de Firma 111
3.4.7 Mejoras Criptográficas 114
3.4.8 Pre-Shared Key 116
3.4.9 Otros Cambios 117
4. Conclusiones 120
5. Bibliografía 129

Agradecimientos

A mi querida Madre, la Dra. Antonia Juncos Brizuela.

Introducción

El año 2018 ha observado la ocurrencia de un importante punto de inflexión, la publicación de la versión TLS 1.3, luego de 10 años de vigencia de su versión anterior, TLS 1.2, como puede verse en la figura.

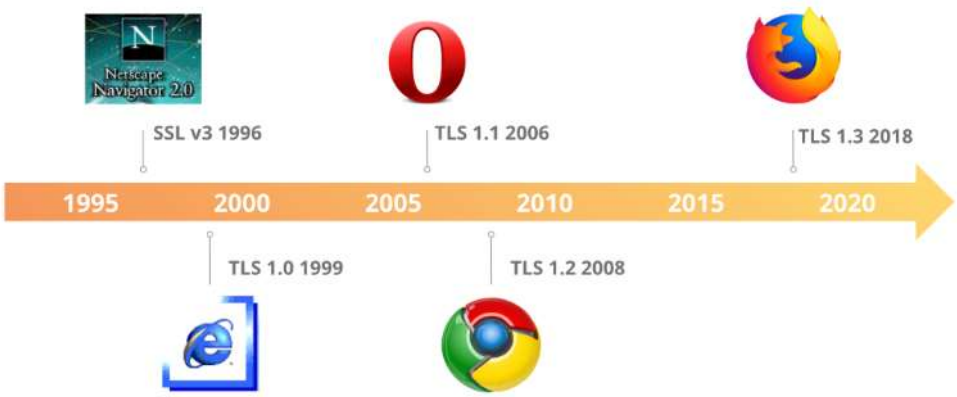

Figura I.1 – Línea de tiempo para SSL/TLS. Fuente [I1].

Esta nueva variante, que algunos estiman razonable llamar “TLS 2.0”, introduce cambios y mejoras fundamentales en su diseño, y todos ellos presentan un desafío muy contundente en cuanto a su implementación.

El siguiente gráfico, obtenido a la fecha de presentación de este trabajo en Marzo de 2022, pone en evidencia la problemática actual. El origen de los datos para el mismo proviene del estudio mensual “*SSL Pulse*” desarrollado por la empresa Qualys [I2]. En él se monitorean alrededor de 150 mil servidores utilizando la lista Alexa de los más populares del mundo.

En el mismo puede notarse que si bien TLS 1.3 se encuentra aprobado como el nuevo estándar hace ya casi cuatro años, una gran cantidad de los servidores sigue operando en base a la versión anterior, existiendo incluso algunos con implementaciones más antiguas.

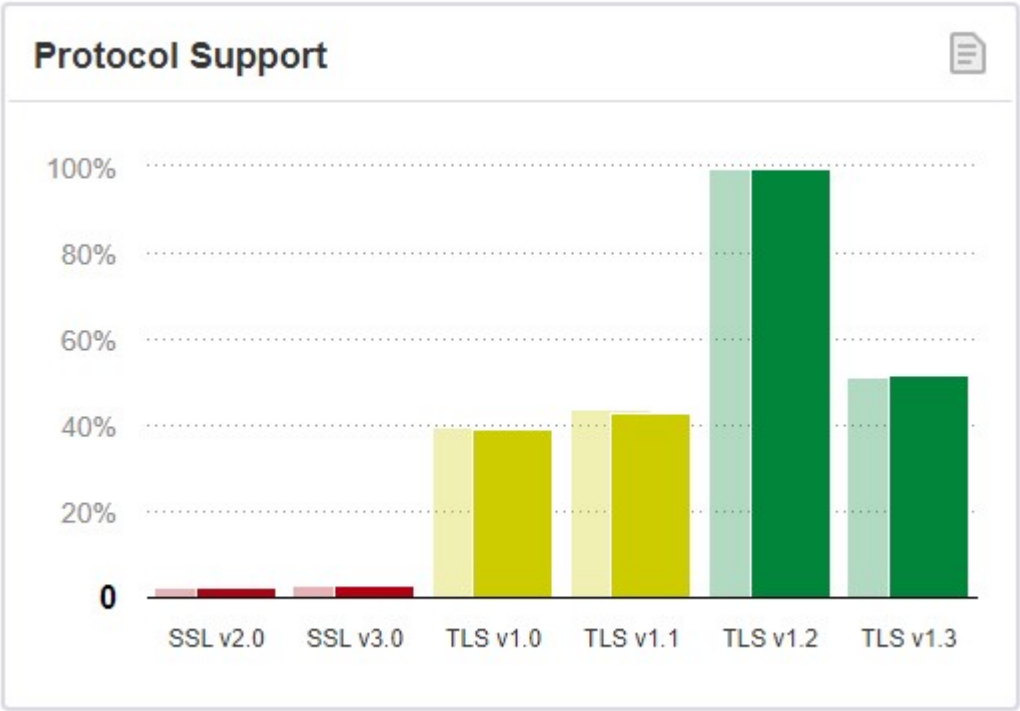

Figura I.2 – Grado de adopción de TLS al 4 de Enero de 2022 . Fuente [I2].

Es por ello por lo que un profesional de TI en general y sobre todo aquel dedicado a la Seguridad de la Información debe estar atento a esta situación, y conocer con claridad las bases establecidas por SSL, los cambios introducidos en las distintas versiones de TLS y en particular, las importantes diferencias que presenta la última, TLS 1.3.

El objetivo de este trabajo es confeccionar una guía clara y actualizada sobre el funcionamiento de las tres versiones relevantes del protocolo, para finalmente concluir sobre las razones técnicas que justifican la demora en su actualización a través de la industria.

El alcance del mismo comprende al diseño original de SSL que sigue siendo su base conceptual y es fundamental para su comprensión, los cambios introducidos hasta la versión más utilizada hoy que es TLS 1.2, y el nuevo diseño de la versión hoy vigente, a partir de la cual ha comenzado lo que estimo adecuado llamar “La Era de TLS 1.3”.

1. Protocolo SSL 3.0

SSL *(Secure Sockets Layer)* es el nombre del protocolo criptográfico desarrollado y propuesto por la empresa *Netscape Communications*.

Su versión 1.0 nunca se publicó por evidenciar serias fallas de seguridad. La versión 2.0 sí fue publicada en 1995, pero también presentó inconvenientes que dieron lugar a un rediseño completo en su versión más conocida, la 3.0 de 1996.

Dicha publicación ha sido convertida en un estándar público de carácter histórico por la IETF *(Internet Engineering Task Force)*, el RFC (*Request For Comments)* 6101 [1] .

En ese documento, se reconoce como autores a Phil Karlton y Alan Freier de Netscape, junto con Paul Kocher. Sin embargo, es importante destacar el liderazgo de Taher El Gamal, como científico en jefe de la empresa, a quien se lo conoce como “el padre de SSL”.

Si bien el nombre SSL sigue siendo utilizado en el mercado, todas sus versiones son hoy obsoletas, y el protocolo ha sido reemplazado por el denominado TLS *(Transport Layer Security)*, desarrollado por la IETF y que estudiaremos en los dos capítulos siguientes.

En este, nos ocuparemos de forma minuciosa en describir el diseño, la finalidad y los componentes conceptuales de SSL. Aún cuando esta versión del protocolo ya no se encuentra vigente, la estructura de su sucesor TLS se basa en ella y en algunos casos es idéntica. La simpleza de esta versión original, permite ver con mayor claridad también las razones que motivaron los cambios y mejoras introducidos por las posteriores.

Hemos tomado aquí como referencia a la RFC 6101, y al excelente libro *SSL Theory and Practice* de Rolf Oppliger [2] del cual se adaptaron algunas figuras. No nos ocuparemos de las vulnerabilidades, que enunciaremos en el capítulo 3.

1.1 Visión General

SSL fue diseñado con el objetivo de proteger criptográficamente las comunicaciones entre dos partes (*peers*) conectadas a través de una red, en modo cliente-servidor. Para lograrlo, ofrece los siguientes servicios de seguridad:

- Autentificación de las partes, y del origen de los datos.
- Confidencialidad.
- Integridad.

El protocolo se ubica para ese fin, en una capa intermedia y adicional entre la de Aplicaciones y la de Transporte en el modelo TCP/IP, como lo muestra la siguiente figura.

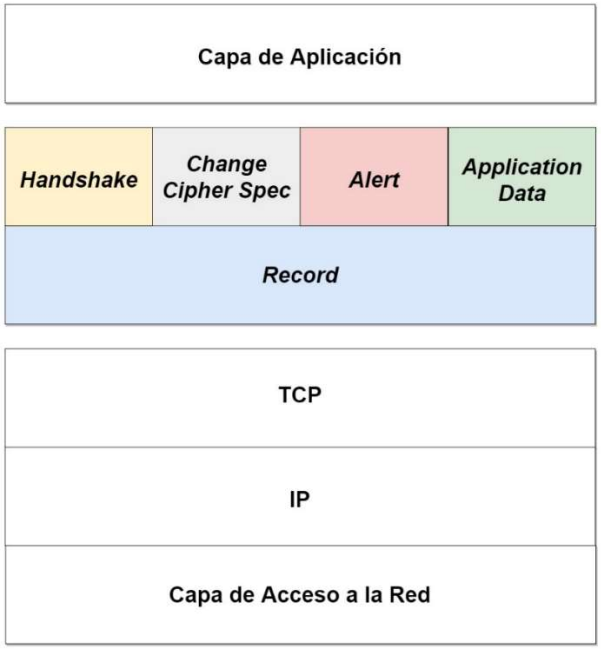

Figura 1.1 - Ubicación de SSL en el modelo TCP/IP.

SSL no ofrece el servicio de no repudio, aún cuando utiliza criptografía de clave pública (PKI) y firmas digitales.

En el gráfico puede observarse que en realidad, SSL se compone a su vez de dos capas, y varios “subprotocolos”. La capa inferior contiene únicamente al subprotocolo ***Record***, mientras que la superior engloba a cuatro subprotocolos denominados ***Handshake***, ***Alert***, ***Change Cipher Spec*** y ***Application Data***.

Este ingenioso diseño, supone el siguiente esquema de funcionamiento. Inicialmente, las dos partes negocian ciertos parámetros criptográficos en un proceso que se denomina *handshake*, utilizando el subprotocolo de ese mismo nombre, y basado en un concepto similar al “*three-way-handshake*” del protocolo de red TCP.

En cualquier momento de esa negociación, o de cualquier etapa en las comunicaciones, el subprotocolo ***Alert*** permite señalar errores ya sea para su corrección o para cancelar todo el proceso.

Finalizada la negociación de parámetros, se utiliza el subprotocolo ***Change Cipher Spec*** como señal para activar la protección criptográfica y comenzar a transmitir datos.

Desde el *handshake* hasta el final, el protocolo ***Record*** participa de todas las comunicaciones. Este subprotocolo se encarga de fragmentar, comprimir, autentificar, cifrar y encapsular los mensajes de cualquiera de los subprotocolos de la capa superior.

Comprender el mecanismo general requiere la revisión de varios conceptos y componentes adicionales, como veremos a continuación.

1.1.1 Sesiones y Conexiones

El protocolo SSL fue creado con la idea de poder ahorrar el esfuerzo computacional que implica la negociación inicial, el *handshake* que pronto veremos en detalle, en las sucesivas conexiones entre dos partes.

En principio, el propio diseño de SSL incluye los conceptos de “Sesiones” y de “Conexiones”.

Una sesión es básicamente una asociación entre las dos partes, la cual incluye la información sobre los parámetros que se negocian durante el apretón de manos inicial.

Las conexiones por su parte, son las instancias de comunicación mediante las cuales efectivamente se transmiten los datos.

Lo importante es comprender que entre dos partes, pueden iniciarse varias conexiones activas relacionadas con una sesión dada. Además, pueden coexistir sesiones simultáneas, lo que es poco frecuente.

1.1.2 La Máquina de Estados

Cada extremo mantiene un registro de información de estado sobre los parámetros vigentes desde el inicio de la ejecución hasta el final de las comunicaciones. Es por ello por lo que se dice que SSL es un protocolo *stateful*.

En realidad, tanto el Cliente como el Servidor mantienen su propio registro de ciertos parámetros relacionados con la sesión, así como otros parámetros relacionados con cada conexión. Estos registros se van alimentando y sincronizando durante la negociación inicial realizada vía el subprotocolo ***Handshake***.

Para las sesiones, la información de estado mantenida por cada participante se muestra en la tabla a continuación.

Nombre Original	Descripción
`session identifier`	**Identificador de Sesión** Una secuencia arbitraria de tamaño máximo 32 bytes, establecida por el Servidor, que identifica a cada sesión activa o reanudable.
`peer certificate`	**Certificado de la otra parte** El certificado versión X.509v3 del otro extremo, si es que está disponible.
`compression method`	**Algoritmo de Compresión**
`cipher spec`	**Parámetros Criptográficos** Algoritmos de cifrado y de MAC, junto con parámetros varios adicionales.
`master secret`	**Clave Maestra** Clave de 48 bytes, compartida por ambas partes.
`is resumable`	**Es Reanudable** Una bandera que indica si la sesión puede utilizarse para establecer nuevas conexiones.

Tabla 1.1 - Parámetros de estado para una sesión.

Y para las conexiones, que como ya mencionamos pueden existir varias para una sesión, el registro es el siguiente:

Nombre Original	Descripción
`server random` `client random`	**Números Aleatorios del Cliente y del Servidor** Secuencias de bytes establecidas por cada parte en cada conexión.
`server write MAC key`	**Clave MAC del Servidor** Clave que utiliza el Servidor para efectuar operaciones MAC sobre los datos que envía
`client write MAC key`	**Clave MAC del Cliente** Clave que utiliza el Cliente para efectuar operaciones MAC sobre los datos que envía
`server write key`	**Clave de Cifrado del Servidor** Con la que el Cliente descifra los datos recibidos.
`client write key`	**Clave de Cifrado del Cliente** Con la que el Servidor descifra los datos recibidos.
`initialization vectors`	**Vectores de Inicialización** Utilizados en el caso en que se implemente el modo CBC para el cifrado en bloques.
`sequence numbers`	**Números de Secuencia** Se mantiene la numeración de todos los mensajes intercambiados entre ambas partes.

Tabla 1.2 - Parámetros de estado para una conexión.

1.1.3 Conjuntos de Cifrado

Un *Cipher Suite* es un conjunto de algoritmos utilizados para proteger una conexión de red. El término fue introducido con la versión SSL 3.0.

Todo el proceso de negociación de parámetros y generación de claves que se realiza durante el *handshake*, requiere de varios algoritmos criptográficos que se utilizan con distintos fines.

Al principio de esa negociación, el Cliente propone al Servidor mediante una lista ordenada por preferencia, una serie de conjuntos de cifrado de los cuales el Servidor selecciona uno.

Todos los *Cipher Suites* respetan un esquema de nomenclatura, que además de identificarlos permite conocer qué algoritmos incluye. Un ejemplo es el siguiente:

SSL_DH_RSA_WITH_DES_CBC_SHA

Donde SSL es el protocolo, DH es el algoritmo de intercambio de clave asimétrico (*Key Exchange*), RSA es el método de autentificación durante el *handshake*, DES es el de cifrado simétrico, CBC su modo de operación y SHA el de *hashing*. Discutiremos los detalles más adelante.

La dupla compuesta por el método de cifrado simétrico y el de *hash* se denomina “especificación de cifrado” (*Cipher Spec*). Esta dupla pasa a ser uno de los datos mantenidos en el registro de sesión de la máquina de estados.

A partir del conjunto de cifrado seleccionado por el Servidor, se utiliza cada algoritmo para completar la parte de la negociación que le compete, y calcular las claves simétricas que se necesitan para todo el proceso. La clave denominada “maestra” es un parámetro de sesión, mientras que las otras que se derivan de ella, forman parte de los parámetros de conexión.

Cabe destacar que la especificación RFC 6101 incluye 31 conjuntos de cifrado, y que uno de ellos es que se usa inicialmente proveyendo protección nula.

1.1.4 Intercambio y Generación de Claves

El objetivo del subprotocolo ***Handshake*** es el de negociar entre las partes una clave simétrica, que se utilizará tanto para autentificar los mensajes, como para luego protegerlos a través del cifrado.

Para lograrlo, emplea criptografía asimétrica en un proceso que se denomina “intercambio de clave” (*Key Exchange*). El resultado es una clave preliminar de 48 *bytes* denominada `pre_master_secret`.

En SSL 3.0 se pueden usar tres algoritmos para el intercambio de esta clave. Uno es RSA, el otro es Diffie-Hellman (DH), y el tercero FORTEZZA [3] que no trataremos en este trabajo.

En algunos casos, se combina el intercambio de clave con la autentificación de las partes, mediante certificados digitales. En general, es solo el Servidor quien provee un certificado. También existe una modalidad “anónima”, en donde los mismos no se incluyen.

Si el método de intercambio de clave es RSA, el Cliente genera una `pre_master_secret` y la cifra con la clave pública del Servidor. Ella puede ser fija y surgida de su certificado enviado previamente al Cliente, o efímera, generada para esta conexión en particular. El Servidor recibe la clave simétrica preliminar y la descifra con su clave privada.

Para el caso en que se utilice DH, ambos generan la `pre_master_secret` a partir del resultado de los cálculos que supone ese algoritmo. Hay tres versiones de este tipo de intercambio en SSL 3.0.

La primera se denomina Diffie-Hellman estático, o simplemente DH. Sus parámetros son fijos y surgen generalmente del certificado del Servidor, aunque puede haber también un certificado del Cliente.

De no haberlo, entonces el Cliente genera sus parámetros durante el *handshake*.

La segunda es conocida como Diffie-Hellman Efímero, o DHE. En este caso, cada parte genera sus parámetros en cada negociación, y los envía al otro extremo en un mensaje del *handshake* que describiremos más adelante.

Dado que en esta modalidad los parámetros no surgen de un certificado, su autentificación se logra a través de firmas digitales. Lo que se hace es utilizar la clave privada del certificado habilitada para la firma en modalidad RSA o DSS (*Digital Signature Standard*).

En tercer lugar, se encuentra el método DH anónimo (*DH.anon*). Como su nombre lo indica, no hay autentificación de las partes, y el intercambio de clave se realiza sin esta comprobación.

De los dos métodos, RSA o DH, la variante DHE es la más segura dado que provee la propiedad de "secreto hacia adelante" (*Forward Secrecy*), tema que abordaremos en el capítulo 3.

Una vez obtenida la `pre_master_secret`, cada parte construye la clave simétrica definitiva denominada `master_secret`, de la siguiente forma:

```
master_secret = MD5(pre_master_secret + SHA('A' +
    pre_master_secret + ClientHello.random +
ServerHello.random)) + MD5(pre_master_secret + SHA('BB' +
    pre_master_secret + ClientHello.random +
ServerHello.random)) + MD5(pre_master_secret + SHA('CCC'
    + pre_master_secret + ClientHello.random +
            ServerHello.random))
```

Y esta clave pasa a formar parte de los parámetros de sesión. La fórmula toma como argumentos dos valores aleatorios `random` que cada parte genera y transmite al inicio del *handshake*, la clave preliminar, y las cadenas de texto fijas "A", "BB" y "CCC" expresadas en hexadecimal.

Nótese que se concatenan tres *hashing* anidados, con MD5 y SHA-1, lo que constituye una función de generación pseudoaleatoria PRF (*Pseudo Random Function*) única y reforzada para SSL. Como el tamaño de salida de MD5 es 16 *bytes*, se obtiene una clave de 48 *bytes*.

Los parámetros criptográficos del estado de conexión, se derivan de la clave maestra, que sirve como fuente de entropía. La clave preliminar por su parte, ya puede ser eliminada de la memoria en forma segura.

Con un criterio similar al utilizado para generar la `master_secret`, se construye un bloque de longitud arbitraria denominado `key_block`:

```
key_block = MD5(master_secret + SHA('A' + master_secret +
       ServerHello.random + ClientHello.random)) +
     MD5(master_secret + SHA('BB' + master_secret +
       ServerHello.random + ClientHello.random)) +
    MD5(master_secret + SHA('CCC' + master_secret +
   ServerHello.random + ClientHello.random)) + [...];
```

Aquí la iteración continúa hasta que logra un tamaño tal, que permita generar las siguientes claves que llamaremos “derivadas”:

```
client_write_MAC_secret
server_write_MAC_secret
    client_write_key
    server_write_key
     client_write_IV
     server_write_IV
```

Donde las dos primeras sirven para autentificar cada mensaje, las dos segundas son las que corresponden al cifrado, y las dos restantes son los vectores de inicialización para el modo CBC, por lo que son opcionales. Estos seis parámetros corresponden al estado de conexión.

1.2 Subprotocolo *Record*

La capa inferior del protocolo SSL denominada ***Record*** es la "locomotora" de todo el mecanismo. Esta capa participa de todas las comunicaciones, procesando y encapsulando a los mensajes de la capa superior, transmitiéndolos luego hacia TCP. Por supuesto, también realiza estas operaciones en el sentido inverso.

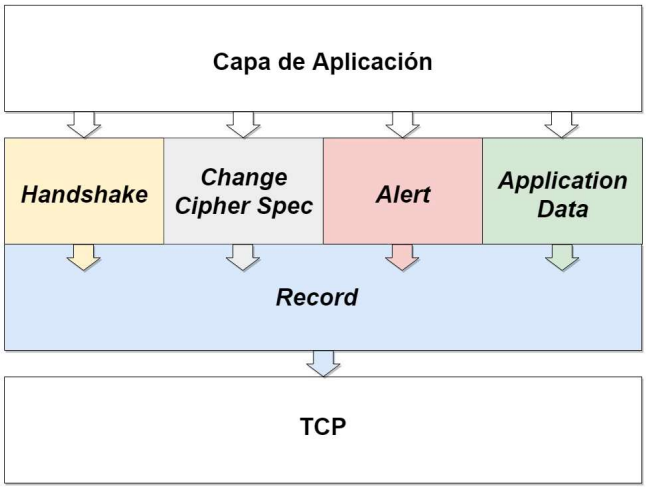

Figura 1.2 - Subprotocolo ***Record***.

El procesamiento que realiza el subprotocolo ***Record*** implica la fragmentación de cada mensaje superior en partes manejables que se tratan individualmente, su compresión que es opcional, la autentificación y el cifrado. Como último paso, se realiza el encapsulamiento.

Los cinco pasos mencionados se reflejan en la siguiente figura, y se describen con mayor detalle a continuación, cada uno en su propio apartado.

Es importante tomar en cuenta que para su operación, el subprotocolo ***Record*** se alimenta de los parámetros activos de la máquina de estados, es decir, de ciertos parámetros relacionados con la sesión, y de otros pertenecientes a la conexión. Los primeros, surgen a su vez del conjunto de cifrados y el método de compresión negociados durante el *handshake*.

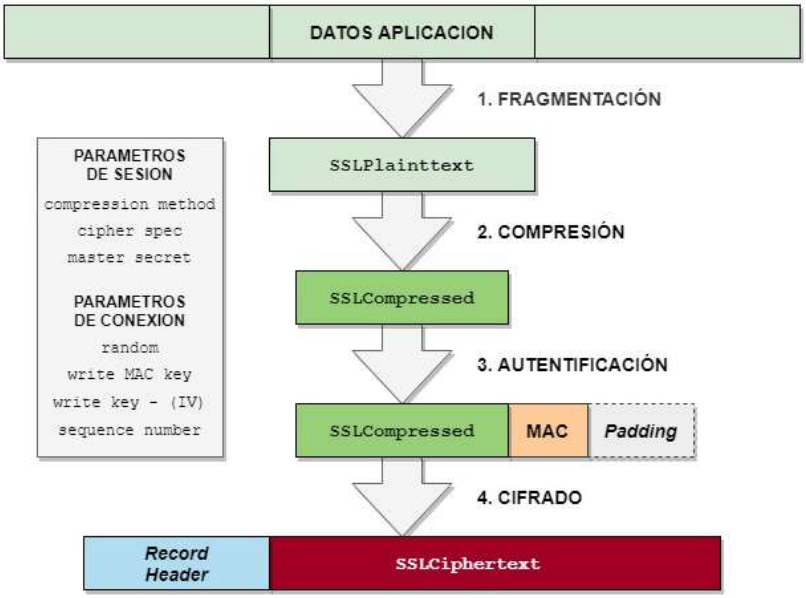

Figura 1.3 Procesamiento y encapsulado del subprotocolo ***Record***.

Los segundos, se combinan entre sí para lograr la autentificación y el cifrado. Una aclaración importante es que al inicio, cuando todavía no se completó el *handshake*, el conjunto de cifrado activo es uno especial, sin compresión, sin cifrado y sin autentificación:

SSL_NULL_WITH_NULL_NULL

Otra aclaración esta vez sobre la figura, es que el encabezado (*record header*) que en ella se agrega al final, en realidad se va actualizando a través de cada paso, desde el comienzo. Ya veremos su contenido.

1.2.1 Fragmentación

En este primer paso los datos que provienen de alguno de los cuatro subprotocolos superiores, se subdividen en bloques de hasta 2^{14} *bytes*, que se procesarán como se mencionó anteriormente, en forma individual.

La estructura de datos resultante, es denominada `SSLPlaintext`, como se ve en la figura 1.3.

1.2.2 Compresión

El segundo paso es opcional, y si se decide efectuar la compresión, esta debe suceder previo al cifrado, dado que a la inversa no tendría sentido.

En todos los casos, se utiliza el algoritmo de compresión especificado en el parámetro `compression method` de la máquina de estado, el cual se negocia al principio del *handshake*. Sin embargo, el único algoritmo definido en SSL 3.0 es el valor nulo, por lo que en realidad no se comprime.

Haya o no compresión, la estructura de datos anterior `SSLPlaintext` se transforma en una nueva denominada `SSLCompressed`. Su tamaño puede ser de hasta 2^{14} *bytes* más 1024.

1.2.3 Autentificación

El objetivo de este paso es el de comprobar la integridad del mensaje y que el mismo sea auténtico. Para ello, se utiliza un algoritmo MAC (*Message Authentication Code*) específico del protocolo SSL, y similar al propuesto en la RFC 2104 [4]. Su construcción es la siguiente:

```
hash(MAC_write_secret + pad_2 + hash(MAC_write_secret +
        pad_1 + seq_num + SSLCompressed.type +
    SSLCompressed.length + SSLCompressed.fragment));
```

Se puede observar que en la fórmula, se realizan dos operaciones de `hash` anidadas. El algoritmo utilizado, es parte del parámetro de sesión de la máquina de estado denominado `cipher spec` que a su vez es parte del conjunto de cifrado que se negocia durante el *handshake*.

La autentificación en la función SSL MAC se concreta a través de la clave `MAC_write_secret` que es un parámetro de la conexión en la máquina de estado, derivado del intercambio de claves inicial.

Cada operación de `hash` utiliza como relleno los argumentos `pad_1` y `pad_2`, que consisten en un caracter especial repetido un cierto número de veces, según si se utiliza el algoritmo MD5 o SHA-1.

Se agrega al cómputo, el argumento `seq_num` que correponde al número de secuencia asignado a cada mensaje transmitido, lo que conforma una protección adicional ante eliminaciones, alteraciones o agregados. Este parámetro es parte de los elementos de estado de conexión.

Finalmente, el algoritmo SSL MAC comprueba el tipo de mensaje con el argumento `SSLCompressed.type`, su longitud `.length` y el propio contenido `SSLCompressed.fragment`.

Esta etapa en el procesamiento no genera una estructura de datos nueva sino que agrega un componente para la próxima, que es el cálculo del MAC, de color amarillo en la figura.

1.2.4 Cifrado

El paso número cuatro tiene como finalidad el proteger la confidencialidad de la estructura `SSLCompressed` junto con su MAC y depende del tipo de algoritmo de cifrado que se utilice.

Si se trata de un cifrado por flujo (*stream cipher*) la tarea es sencilla. El único algoritmo de este tipo definido en SSL es RC4, con claves de 40 o 128 *bits* de tamaño.

.La otra opción es el cifrado por bloque (*block cipher*). En este caso, es necesario ajustar el tamaño del conjunto `SSLCompressed` más el MAC que es la información a cifrar, al tamaño del bloque del algoritmo, por lo que es posible que se necesite aplicar un relleno (*padding*).

Además, algunos modos de cifrado por bloque tales como CBC requieren del uso de un vector de inicialización (IV - *Initialization Vector*). Este parámetro para el primer bloque surge del estado de conexión, y para los siguientes se utiliza el bloque anterior.

La clave para el cifrado es otro parámetro del estado de conexión, derivado de la clave maestra que se negocia en el *handshake*. En la figura se ve como `write key`.

La estructura de datos final, autentificada y protegida, con o sin relleno según el caso se denomina `SSLCiphertext` y se la ve en color rojo en la figura. Puede contener más de un mensaje del mismo tipo.

Una vez terminado el procesamiento, se agrega el encabezado. Este consta de tres campos que son:

`type` – Un *byte* para el código del subprotocolo superior que envía el mensaje: 20 para ***Change Cipher Spec,*** 21 para ***Alert***, 22 para ***Handshake*** y 23 para ***Application Data***.

`version` – Dos *bytes* para la versión de SSL en uso, que en este caso es 3.0. Notar que la versión se compone de un valor mayor 3 y uno menor 0, expresados uno en cada *byte*.

`length` – Dos *bytes* para el tamaño en *bytes* de la estructura de datos final que se transmite, a la cual se denomina fragmento. Si bien esos dos *bytes* tienen valor máximo 2^{16}-1, la RFC establece como máximo 2^{14}-1.

Por lo tanto, el contenido completo de una estructura ***Record***, comprende cuatro campos. El encabezado con los tres descritos arriba y el `fragment` con los datos protegidos.

1.3 Subprotocolo *Handshake*

La capa superior de SSL incluye cuatro subprotocolos, de los cuales ***Handshake*** es sin duda el más importante y complejo. Su función es primordial y consiste en negociar los parámetros necesarios para la transmisión segura de los datos.

Esa negociación que sucede al inicio de las comunicaciones, incluye la posibilidad de que una o las dos partes se autentifiquen mediante certificados digitales. Generalmente esto sucede del lado del Servidor.

El “apretón de manos” completo ocurre en cuatro viajes (*trips*), tal como muestra la figura 1.4. Cada viaje incluye uno o varios mensajes que deben seguir un orden estricto, donde si no se lo respeta se aborta la operación.

Los mensajes contienen uno o más parámetros que son los esenciales para completar el proceso.

Estos cuatro viajes, descritos en forma resumida, son los siguientes:

- El primero consiste en un único mensaje CLIENTHELLO enviado por el Cliente.
- El segundo incluye como mínimo dos mensajes y hasta cinco en total, enviados por el Servidor:
    1. Un mensaje SERVERHELLO en respuesta al del Cliente.
    2. El certificado digital del Servidor si es necesario, que en general lo es. El mensaje se llama CERTIFICATE.
    3. También opcional, un mensaje SERVERKEYEXCHANGE, con parámetros necesarios adicionales.
    4. Si es requerido que el Cliente también presente su certificado, entonces se envía un mensaje CERTIFICATE REQUEST.
    5. Un mensaje que finaliza este segundo viaje, denominado SERVERHELLODONE.

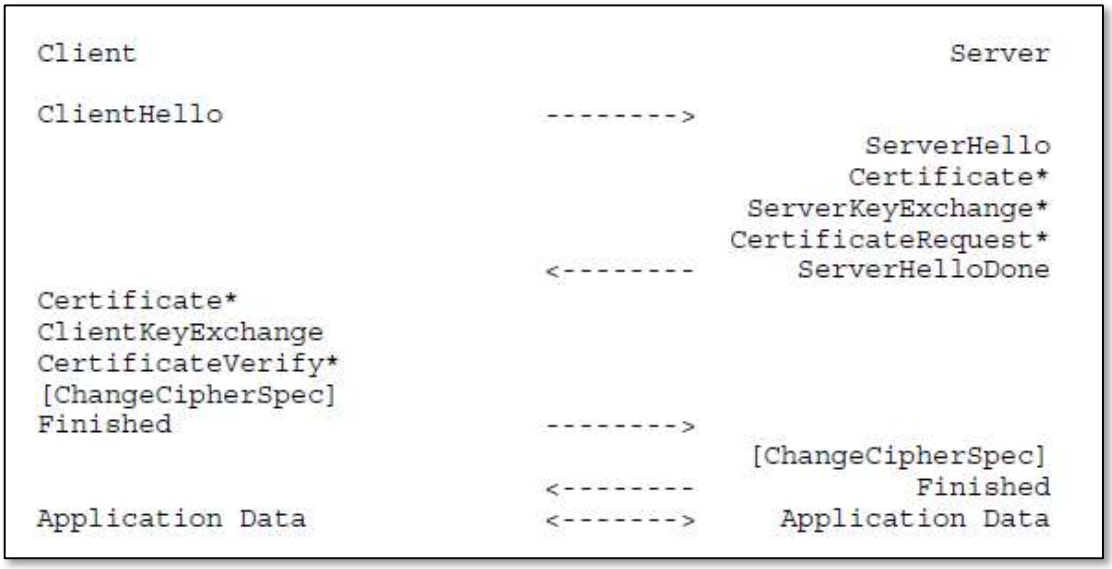

Figura 1.4 - el *handshake* completo.

- El viaje número tres consiste en dos a cinco mensajes enviados nuevamente por el Cliente:

    1. En respuesta al pedido del Servidor si eso sucedió, un mensaje CERTIFICATE.

    2. Este mensaje es el paso más importante de la negociación, y depende del algoritmo de intercambio de claves elegido. Su nombre es CLIENTKEYEXCHANGE.

    3. Si el Cliente tuvo que enviar su certificado en el paso 1, entonces también debe enviar una verificación del mismo. Se trata de un mensaje firmado digitalmente con la clave pública de su certificado. El nombre del mensaje es CERTIFICATEVERIFY.

    4. El cuarto mensaje denominado CHANGECIPHERSPEC no pertenece al subprotocolo ***Handshake***. Se incluye aquí solo para respetar el orden expuesto en la figura.

    5. FINISHED es el nombre del último mensaje de este grupo, el cual finaliza la negociación por parte del Cliente.

- La cuarta etapa se compone de dos mensajes finales enviados por el Servidor:

    1. Su propio mensaje CHANGECIPHERSPEC.

    2. También un mensaje FINISHED, que ahora marca el final de la negociación por parte del Servidor.

En esta introducción hemos realizado una descripción general del funcionamiento de este subprotocolo ***Handshake***. A continuación, revisaremos primero los conceptos de reanudación y renegociación, para luego analizar cada uno de los mensajes en detalle.

1.3.1 Reanudación y Renegociación

Ya mencionamos anteriormente que el proceso de negociación inicial denominado *handshake* es costoso en términos computacionales. Esto implica sobre todo, la posibilidad de que su duración sea extensa.

En la primera conexión entre un Cliente y un Servidor, no queda otro camino que sufrir ese retraso. Pero en las sucesivas, hay dos opciones que son la reanudación (*resumption*) y la renegociación (*renegotiation*) de sesiones.

Una renegociación consiste en ejecutar nuevamente todo el “apretón de manos”, y establecer una nueva sesión con su correspondiente costo. Esto puede ser solicitado en cualquier momento por el Cliente, enviando un nuevo mensaje CLIENTHELLO, y también por el Servidor, en este caso enviando un mensaje conocido como HELLOREQUEST. Este último caso es muy poco frecuente. Hay varios motivos por el que se puede optar por renegociar, tal como la renovación de los parámetros tanto de sesión como de conexión, o el llegar al límite de los números de secuencia de los mensajes.

La segunda opción, la reanudación de sesión, es la más eficiente, e implica un ahorro de esfuerzo que se ve plasmado en una versión resumida del proceso de negociación que se ve en la siguiente figura.

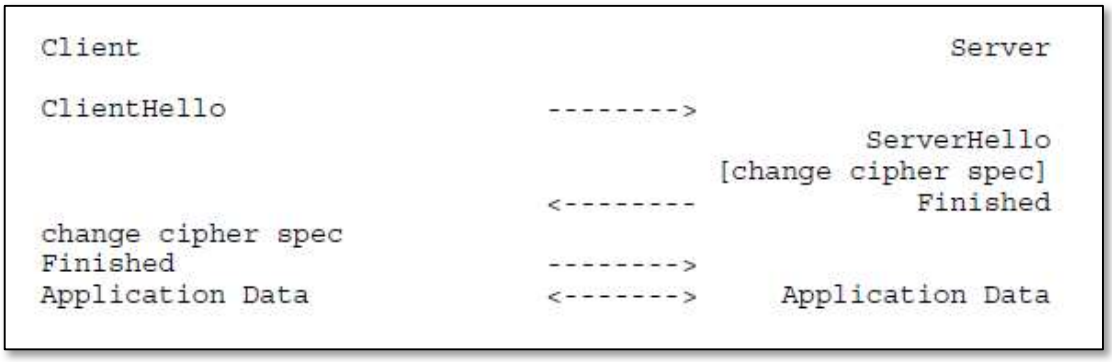

Figura 1.5 - El *handshake* para la reanudación.

Como se puede apreciar, los pasos de la reanudación y los cálculos asociados son muchos menos. El Cliente solicita reanudar un ID de sesión ya existente, y el Servidor busca en su *caché* de sesiones si todavía tiene los datos de la misma.

Hay que recordar que el ID es uno de los datos que componen el estado de la sesión en la máquina de estados del lado del Servidor, y su nombre es `session identifier`.

Si el Servidor encuentra ese ID responde con un mensaje SERVERHELLO que lo confirma, envía luego la señal CHANGECIPHERSPEC actualizando su máquina de estado y completa la confirmación con un mensaje FINISHED.

El Cliente envía entonces su propio CHANGECIPHERSPEC, también actualiza su máquina de estado, envía luego un FINISHED, y ya se puede comenzar a intercambiar datos, con las protecciones adecuadas.

1.3.2 Los Mensajes

El subprotocolo ***Handshake*** incluye hasta 12 mensajes que son intercambiados entre el Cliente y el Servidor en la versión completa de su ejecución. Cada uno de estos mensajes, es protegido y encapsulado por el subprotocolo ***Record***.

Cabe recordar que en un fragmento del subprotocolo ***Record*** puede ubicarse más de un mensaje ***Handshake***.

MENSAJE HELLOREQUEST

Comenzamos describiendo este simple mensaje opcional, que el Servidor puede enviar al Cliente para solicitarle una renegociación, solo para respetar el orden establecido por el protocolo.

Este mensaje se utiliza muy poco en la práctica, y sirve para casos especiales en que el Servidor necesita que se realice el proceso de *handshake* desde cero. Por ejemplo, en el caso en que una sesión haya persistido por mucho tiempo, y por razones de seguridad convenga renovar los parámetros criptográficos tales como las claves.

Record Header	
`type`	22 (0x16). Código de mensaje ***Handshake***.
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	2 *bytes*. Tamaño del fragmento completo=0.
Handshake Header	
`type`	0 (0x00). Código del mensaje HELLOREQUEST.
`length`	3 *bytes*. Tamaño del mensaje=0.

Tabla 1.3 - Campos de un mensaje HELLOREQUEST.

En la tabla podemos observar la estructura del mensaje y los campos involucrados, incluyendo el encabezado añadido por el subprotocolo ***Record***.

Su estructura es muy simple, y el mensaje de contenido vacío viaja solo en un fragmento individual. Se trata de un aviso, que debe ser enviado una sola vez, e ignorado por el Cliente si ya existe un apretón de manos en curso.

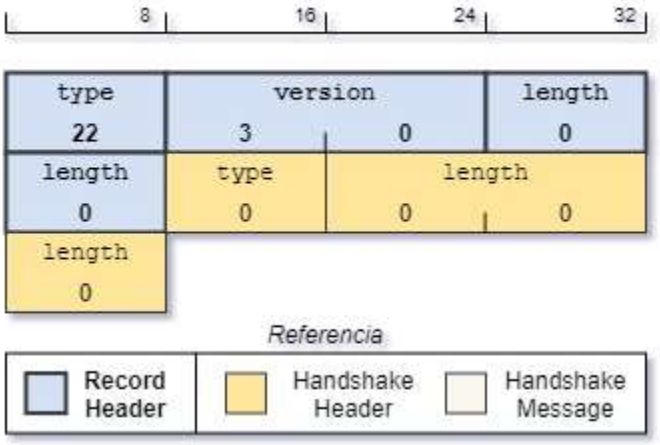

Figura 1.6 – Mensaje HELLOREQUEST.

MENSAJE CLIENTHELLO

Si bien este mensaje no es realmente el primero previsto por el protocolo como vimos en el título anterior, sí lo es en la práctica y aparece iniciando la negociación en la mayoría de los diagramas publicados en la bibliografía y otras fuentes sobre SSL que describen al *handshake*.

CLIENTHELLO es uno de los mensajes más importantes del proceso, y su contenido incluye varios parámetros de sesión que ya comienzan a alimentar la máquina de estado de ambas partes. El mensaje incluye los siguientes campos, que se detallan en la tabla 1.4, y que vale la pena analizar individualmente.

Comenzando por el campo `client_version`, podemos decir que este dato que ya está en el *Record Header* se reitera por razones de compatibilidad con versiones anteriores.

El campo `random` es en realidad una estructura que se compone de dos partes. La primera de tamaño 4 *bytes*, corresponde a la fecha actual expresada en formato UNIX estándar. Los siguientes 28 *bytes*, contienen el verdadero valor aleatorio que debería ser generado adecuadamente. Este valor compuesto sirve como fuente de entropía para los cálculos de clave que hemos descrito en las secciones anteriores.

Record Header	
`type`	22 (0x16). Código de mensaje ***Handshake***.
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	2 *bytes*. Tamaño del fragmento completo.
Handshake Header	
`type`	1 (0x01). Código del mensaje CLIENTHELLO.
`length`	3 *bytes*. Tamaño del mensaje.
CLIENTHELLO	
`client_version`	2 *bytes*. Es la mayor versión de SSL que soporta el Cliente.
`random`	32 *bytes*. Número aleatorio del Cliente.
`session_id`	32 *bytes* máximo. ID de sesión que se desea reanudar, o cero si es una nueva.
`cipher_suites`	Variable. Lista de conjuntos de cifrado que soporta el Cliente.
`compression_methods`	Variable. Lista de métodos de compresión que soporta el Cliente.

Tabla 1.4 - Campos de un mensaje CLIENTHELLO.

Con respecto a `session_id`, también se trata de una estructura, donde el primer *byte* indica su tamaño que puede alcanzar los 32 *bytes*. Este dato es muy importante, ya que define si el Cliente desea reanudar una sesión anterior, renovar los parámetros de una conexión existente, o simplemente mantener varias conexiones simultáneas sin la necesidad de realizar en cada caso el *handshake* completo. Si no se está reanudando, entonces el valor del primer *byte* es cero y el resto es nulo.

Una vez más el mensaje CLIENTHELLO presenta una estructura destinada a proponer al Servidor una lista de conjuntos de cifrado que el Cliente dice soportar, ordenada por su preferencia.

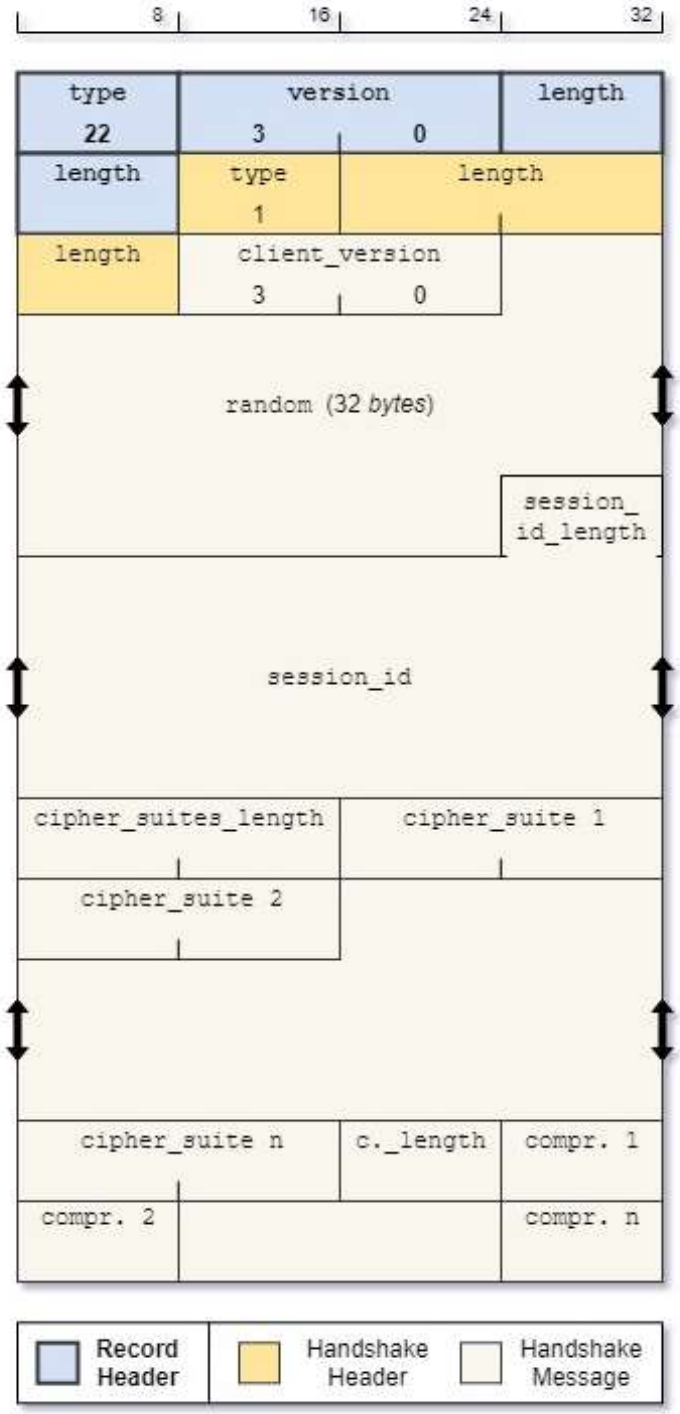

Figura 1.7 - Mensaje CLIENTHELLO.

La misma se denomina `cipher_suites`, y se compone de dos primeros *bytes* que enuncian su tamaño, y luego de una lista de longitud variable de conjuntos de cifrado enumerados en dos *bytes* cada uno, que especifican su código.

El mismo esquema se repite en el campo `compression_methods`. Sin embargo, vale aclarar que en SSL 3.0, solo se define la compresión nula, por lo que se acostumbra a establecer su tamaño en uno, y el byte siguiente en cero, lo que corresponde a no utilizar compresión.

Cabe comentar que CLIENTHELLO es el único mensaje del *handshake* en el que se permite incluir datos adicionales posteriores a este campo que especifica el método de compresión. En esta versión, se ignoran esos datos, pero en TLS toman gran importancia porque así es como se implementan las "extensiones" del protocolo.

MENSAJE SERVERHELLO

Luego de haber recibido un mensaje CLIENTHELLO, el Servidor lo verifica, lo procesa y si todo es correcto responde con este mensaje denominado SERVERHELLO.

La estructura de este mensaje de respuesta es muy similar a la del anterior, salvo que en el caso de los *Cipher Suites* se selecciona uno en particular de los sugeridos por el Cliente, y lo mismo ocurre con el algoritmo de compresión.

La siguiente tabla muestra los campos que incluye un mensaje SERVERHELLO, con su propio encabezado ***Handshake*** y encapsulado por el protocolo ***Record***:

Es recomendable en este punto realizar algunos comentarios respecto de la lista de campos que componen este mensaje.

El campo `random` que genera el servidor, es creado de idéntica forma que el que envía el Cliente, pero es independiente del mismo y tiene otro valor aleatorio distinto. De hecho, en la bibliografía y otras fuentes se lo refiere como "*Server.Random*", aunque el nombre oficial en la especificación es ese.

Record Header	
`type`	22 (0x16). Código de mensaje ***Handshake***.
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	2 *bytes*. Tamaño del fragmento completo.
Handshake Header	
`type`	2 (0x02). Código del mensaje SERVERHELLO.
`length`	3 *bytes*. Tamaño del mensaje.
SERVERHELLO	
`server_version`	Versión de SSL confirmada por el servidor. Su tamaño es 2 *bytes*: 3.0 (0x03, 0x00).
`random`	Un valor aleatorio generado por el Servidor, independiente del enviado por el Cliente. Su tamaño es 32 *bytes*.
`session_id`	El número de sesión que se desea reanudar. Si vale cero, entonces será una nueva. De tamaño variable, máximo 32 *bytes*.
`cipher_suite`	Código del conjunto de cifrado seleccionado por el Servidor. Tamaño: 2 *bytes*.
`compression_method`	Código del compresión seleccionado por el Servidor. Tamaño: 1 *byte*.

Tabla 1.4 - Campos de un mensaje SERVERHELLO.

El campo `session_id` de tamaño máximo 32 *bytes*, incluye un primer *byte* que indica su longitud. El resto del contenido del mismo es el propio valor variable que corresponde al número de sesión, establecido por el Servidor para permitir la reanudación en esta ocasión si el Cliente envió el ID de una sesión anterior, o en la próxima si la sesión es nueva.

En su respuesta, el Servidor está confirmando los parámetros criptográficos sugeridos por el Cliente. Es por ello por lo que selecciona un solo conjunto de cifrado en el campo `cipher_suite`, y un solo algoritmo de compresión `compression_method`.

Cabe notar también que a esta altura de la negociación, ambas partes ya han intercambiado datos importantes que alimentan sus respectivas máquinas de estado, en sus parámetros de sesión.

En particular, ya se han enviado los respectivos `random`, han establecido el `session_id` optando por crear una nueva sesión o reanudar una existente, y han acordado en utilizar un `cipher_suite` y un

`compression_method`. También han coincidido en usar SSL versión 3.0, confirmándolo en `server_version`.

Importante es señalar que hasta aquí, la negociación ocurre sin protección criptográfica, es decir, sin cifrar ni comprimir, y sin autentificación del mensaje. El conjunto de cifrados activo es el que no realiza cambios sobre el texto plano, SSL_NULL_WITH_NULL_NULL.

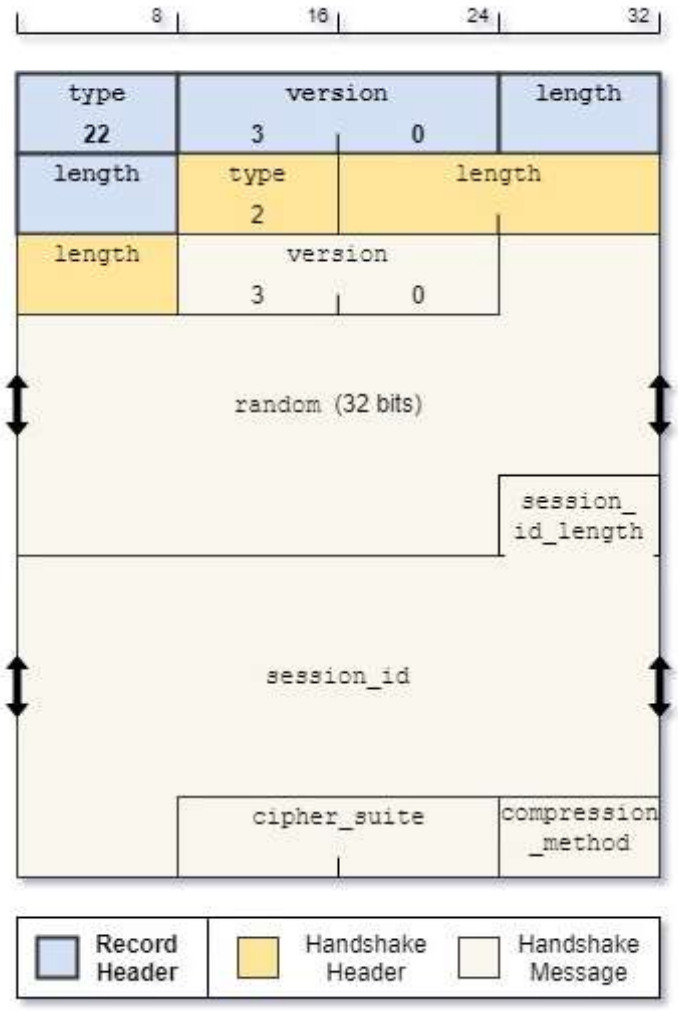

Figura 1.8 - Estructura del mensaje SERVERHELLO.

MENSAJE CERTIFICATE

Con este mensaje llega el momento en el que el Servidor debe autentificarse, enviando su certificado de clave pública. Si bien el mensaje es opcional, dado que existe la posibilidad de hacer la negociación en forma anónima, lo más común es que al menos el Servidor se identifique.

La tabla a continuación expone la estructura de un mensaje CERTIFICATE con todos sus campos, su encabezado ***Handshake*** y el encapsulado ***Record***:

Record Header	
`type`	22 (0x16). Código de mensaje ***Handshake***.
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	2 *bytes*. Tamaño del fragmento completo.
Handshake Header	
`type`	11 (0x0B). Código del mensaje CERTIFICATE.
`length`	3 *bytes*. Tamaño del mensaje.
CERTIFICATE	
`certificate_list`	Cadena completa de certificados del Servidor, de abajo hacia arriba.

Tabla 1.5 - Campos de un mensaje CERTIFICATE.

Este mensaje se compone de un solo campo, denominado `certificate_list` el cual tiene su propia estructura interna. Como indica la tabla, contiene la cadena completa de certificados del Servidor.

Ello implica que el campo mencionado incluye a cada uno de los certificados de esa cadena (*certificate chain*) desde el propio del Servidor, pasando por todas las autoridades certificantes CA (*certificate authorities*) en orden ascendente hasta el certificado raíz.

Dado que los distintos certificados tienen su propia longitud, es preciso delimitarlos individualmente y también registrar el tamaño de toda la cadena.

El encabezado del ***Handshake*** prevé un campo `length` de 3 *bytes*, por lo que su tamaño máximo es de 2^24-1, es decir, que puede ser grande.

Dentro del campo `certificate_list`, cada certificado de la cadena está a su vez precedido por 3 *bytes* que indican su longitud particular. Todo este detalle puede observarse en la figura a continuación.

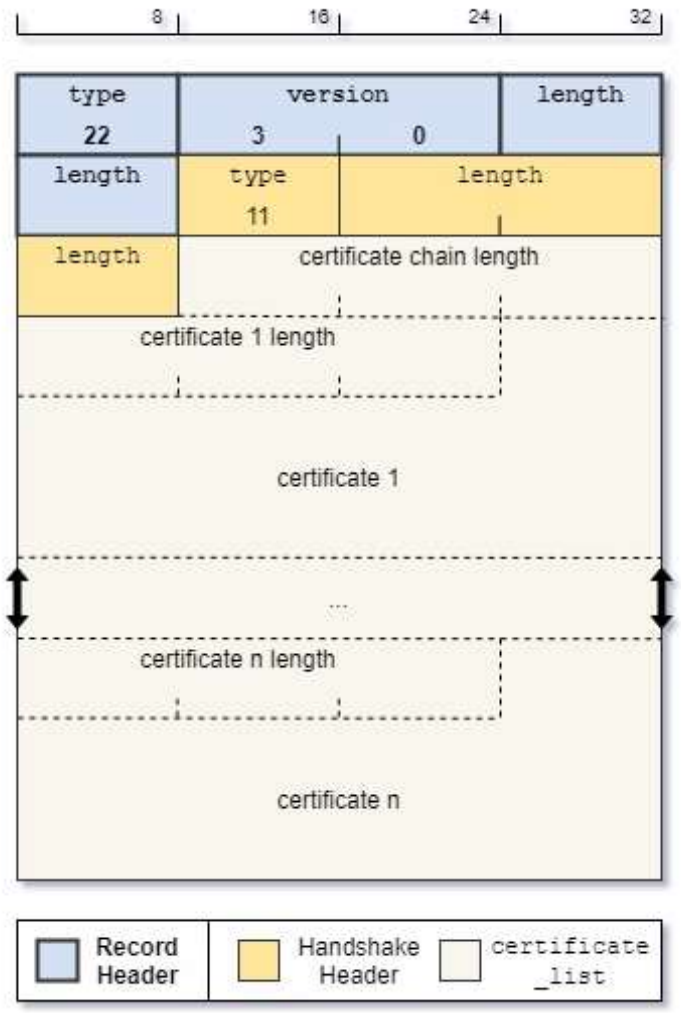

Figura 1.9 - Mensaje CERTIFICATE.

El mensaje CERTIFICATE, si se envía, viaja en la misma segunda etapa que su antecesor SERVERHELLO. Y es importante tener presente que el tipo de certificado enviado debe coincidir con el algoritmo de intercambio de claves negociado previamente. El formato utilizado es en general el X.509.

Mensaje SERVERKEYEXCHANGE

El motivo por el cual el Servidor envía su certificado tal como vimos en el título anterior, es en primer lugar para autentificarse. Sin embargo, además de ello, en ciertos casos el certificado sirve como fuente de parámetros para el intercambio de claves. Podemos entonces considerar la siguiente clasificación de casos:

1. Aquéllos en los que no es necesario este mensaje, y con el certificado del Servidor es suficiente.
2. Los que hacen necesario tanto el certificado del Servidor, como parámetros adicionales en este mensaje.
3. Los que requieren de los parámetros pero no del certificado.

Esta relativa complejidad, surge a partir del conjunto de cifrado negociado anteriormente. Hay que recordar que el *Cipher Suite*, puede incluir tanto un algoritmo de intercambio de claves como un método de autentificación.

La combinación entonces de las distintas alternativas, es la que da origen a los diferentes casos que describiremos en adelante.

El primer caso, corresponde a los que usan o bien un certificado RSA válido para intercambios de clave, o aquéllos que contienen los parámetros de Diffie Hellmann (DH) fijos firmados digitalmente por la autoridad que emite el certificado.

Para ambos, el mensaje SERVERKEYEXCHANGE no es necesario, y con el certificado del Servidor es suficiente para que el Cliente pueda hacer sus cálculos.

Si se utiliza RSA como método de intercambio de claves, entonces el Cliente puede extraer la clave pública directamente del certificado del Servidor y utilizarla luego para cifrar la `premaster_secret` y enviársela.

En cambio si se opta por Diffie-Hellman con parámetros fijos para intercambio de claves, entonces el Cliente obtiene los parámetros del certificado del Servidor, realiza ese intercambio de claves y utiliza el resultado como la propia `premaster_secret`.

Record Header	
`type`	22 (0x16). Código de mensaje ***Handshake***.
`Version`	3.0 (0x03,0x00). Versión de SSL.
`length`	2 *bytes*. Tamaño del fragmento completo.
Handshake Header	
`type`	12 (0x0C). Mensaje SERVERKEYEXCHANGE.
`length`	3 *bytes*. Tamaño del mensaje.
SERVERKEYEXCHANGE	
`ServerDHParams` `ServerRSAParams`	Parámetros de Diffie-Hellman o de RSA.
`Signature`	Opcional, la firma digital del campo anterior.

Tabla 1.6 - Campos de un mensaje SERVERKEYEXCHANGE.

La tabla muestra el contenido de un mensaje SERVERKEYEXCHANGE para los casos 2 (certificado más parámetros) y 3 (solo parámetros), en donde el mismo es necesario. En ellos, según el método de intercambio de claves acordado, se enviarán o bien los parámetros de DHE y el campo tiene como nombre `ServerDHParams` o los de RSA, y entonces el campo será `ServerRSAParams`.

Ambas versiones del campo que contiene los parámetros, poseen una estructura interna que puede verse en las siguientes figuras. Esa estructura es diferente para DHE y para RSA.

Si se utiliza Diffie-Hellman Efímero (DHE) para intercambio de claves entonces se incluye al mensaje SERVERKEYEXCHANGE y éste contiene primero a los párámetros de Diffie-Hellman, y luego a una firma digital para

estos parámetros, que depende del tipo de certificado enviado por el Servidor.

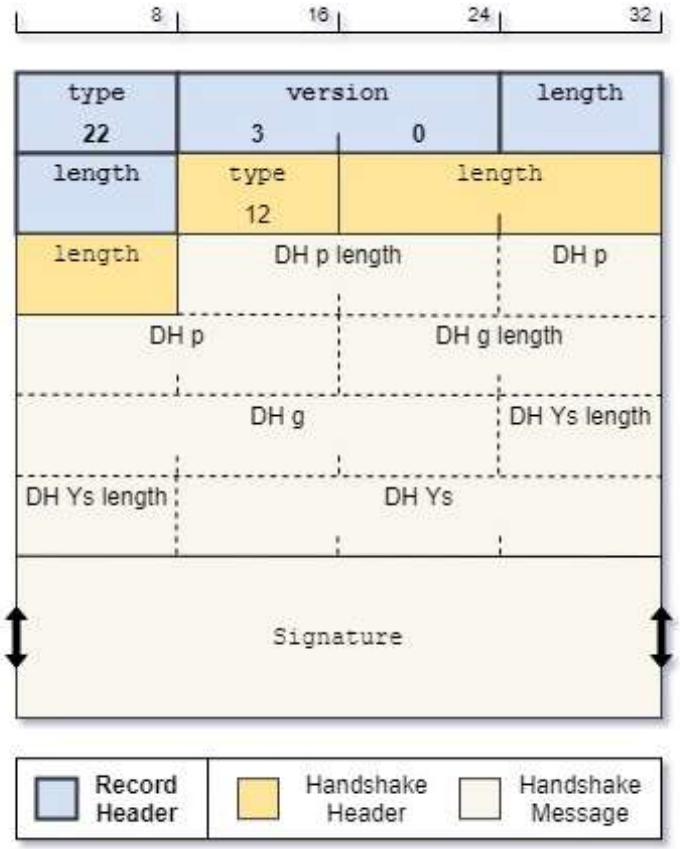

Figura 1.10 - Campos de un mensaje SERVERKEYEXCHANGE para DHE.

Como se puede apreciar en la figura, se incluyen para DHE los factores p (número primo), g (el generador) y por último Ys (el exponente público). Para cada uno de ellos, se anteponen 2 *bytes* que indican su tamaño, dado que son variables. Todos estos datos componen el campo `ServerDHParams`.

Finalmente para DHE combinado con el certificado del Servidor, se agrega el campo `Signature` cuyo formato depende del tipo de certificado enviado.

El otro caso en el que además del certificado del Servidor hacen falta datos adicionales, es cuando se negocia un método de intercambio de claves RSA pero el certificado está solamente habilitado para la firma digital.

Como el Cliente no puede usar la clave pública de ese tipo de certificados para cifrar la `premaster_secret`, el Servidor debe generar en el momento un par de claves pública y privada y enviarlas en un mensaje SERVERKEYEXCHANGE.

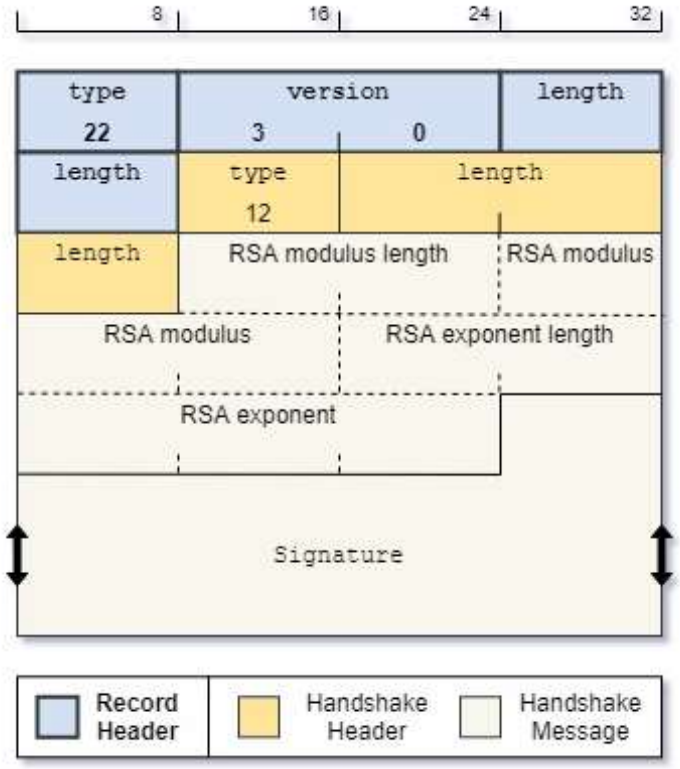

Figura 1.11 - Campos de un mensaje SERVERKEYEXCHANGE para RSA.

La figura superior expone este caso particular, en donde se ve que el Servidor envía al Cliente los parámetros módulo y exponente característicos de RSA, generados expresamente para este intercambio. Y esto requiere también de la firma digital de esos factores, que se agregan en el campo `Signature`.

El contenido del campo `Signature`, a su vez, depende del algoritmo de firma indicado en el certificado del Servidor.

Si el certificado del Servidor es para firma RSA, entonces se concatenan dos cálculos de *hash* individuales, uno con el algoritmo MD5 y otro con SHA-1:

```
md5_hash: MD5(ClientHello.random + ServerHello.random +
                    ServerParams);
sha_hash: SHA(ClientHello.random + ServerHello.random +
                    ServerParams);
```

Y se aplica la firma digital sobre esa concatenación. Por otra parte, si el certificado del Servidor es para firma DSA (Digital Signature Algorithm), solamente se utiliza el segundo cálculo, utilizando solo SHA-1.

Notemos que en las dos fórmulas, se utilizan los valores `random` de ambas partes más los `ServerParams` como protección contra la reutilización tanto de firmas como de parámetros anteriores.

El tercer caso corresponde a los intercambios realizados en forma anónima, en donde no es necesario el certificado del Servidor. Tanto para Diffie-Hellmann como para RSA en modo anónimo, simplemente se envían los parámetros de cada método de intercambio de claves como se ve en las dos figuras de este punto, pero sin el campo `Signature` final.

Corresponde realizar aquí una aclaración de cierre para este mensaje. Hemos dejado sin tratar expresamente, a los conjuntos de cifrado que incluyen a los métodos de intercambio de claves “FORTEZZA” y “RSA_EXPORT”.

Ambos esquemas son hoy obsoletos y además, no son de interés para los objetivos del presente trabajo.

Mensaje CERTIFICATEREQUEST

Un Servidor que funciona en modo no anónimo, puede enviar opcionalmente al Cliente un pedido de su certificado a través de este mensaje. Los servidores anónimos no deben utilizarlo.

Lo que el Servidor envía en este caso es una lista de tipos de certificado que soporta, más otra lista con los nombres DN (*Distinguished Name*) de las CA (*Certificate Authority*) que acepta.

Record Header	
`type`	22 (0x16). Código de mensaje ***Handshake***.
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	2 *bytes*. Tamaño del fragmento completo.
Handshake Header	
`type`	13 (0x0D). Mensaje CERTIFICATEREQUEST.
`length`	3 *bytes*. Tamaño del mensaje.
CERTIFICATEREQUEST	
`ClientCertificateType`	Tipos de certificado que soporta el Servidor.
`DistinguishedName`	Nombres de las autoridades certificantes que el Servidor acepta.

Tabla 1.7 - Estructura del mensaje CERTIFICATEREQUEST.

Como se ve en la tabla, el mensaje consiste en enviar dos campos, `ClientCertificateType` y `DistinguishedName`. Ambos tienen su propia estructura interna que se refleja en la figura siguiente.

El primero, incluye un primer *byte* que indica la cantidad de tipos de certificados aceptables, y luego sus códigos.

El segundo, consiste en 2 *bytes* para el tamaño de toda la lista de nombres DN de las autoridades certificantes, y para cada nombre 2 *bytes* previos que exponen su longitud particular. Esta lista puede ser grande.

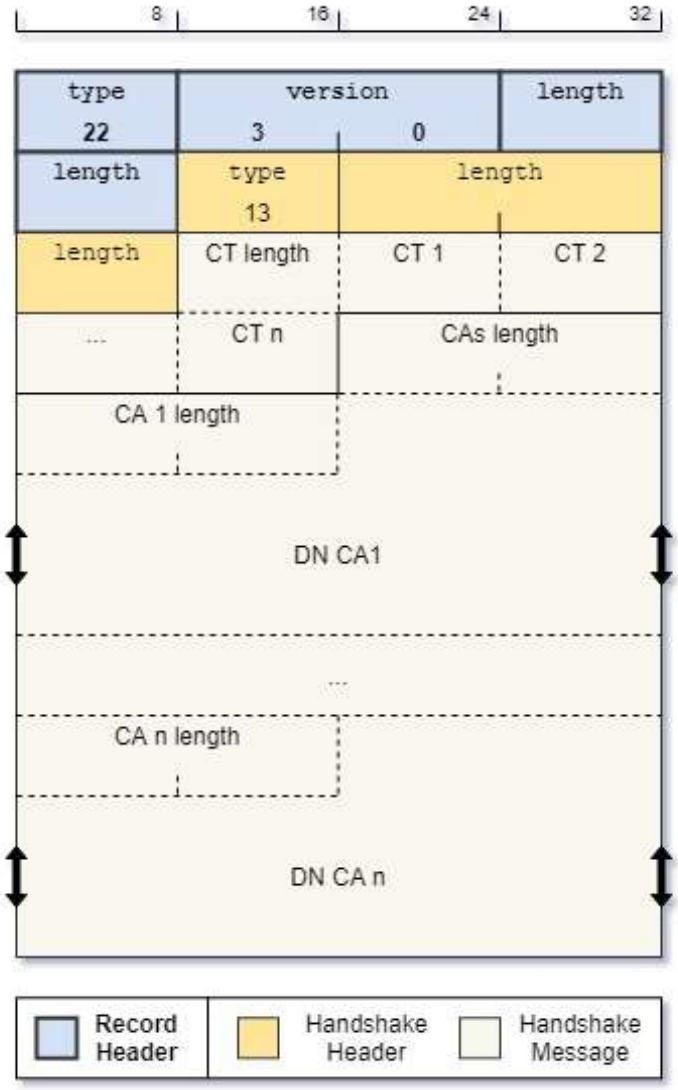

Figura 1.12 - Campos de un mensaje CERTIFICATEREQUEST.

Los tipos de certificados aceptados definidos en la RFC 6101 [1] son los que se enumeran en la siguiente tabla:

Código	Nombre	Descripción
1	rsa_sign	Firma e intercambio de claves RSA.
2	dss_sign	Solo firma DSA.
3	Rsa_fixed_dh	Firma RSA con intercambio de claves DH fijo.
4	Dss_fixed_dh	Firma DSA con intercambio de claves DH fijo.
5	Rsa_ephemeral_dh	Firma RSA con intercambio DH efímero.
6	Dss_ephemeral_dh	Firma DSA con intercambio DH efímero.

Tabla 1.8 – Tipos de certificados definidos por SSL.

Mensaje SERVERHELLODONE

Este mensaje obligatorio enviado por el Servidor marca el final del segundo viaje en la negociación. Se trata simplemente de una señal, por lo que su contenido es nulo.

Record Header	
`type`	22 (0x16). Código de mensaje ***Handshake***.
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	Valor 4. (0x00, 0x04).
Handshake Header	
`type`	14 (0x0E). Mensaje SERVERHELLODONE.
`length`	Valor cero. (0x00,0x00,0x00).

Tabla 1.9 - Campos del mensaje SERVERHELLODONE.

Dado su contenido nulo, el valor de su campo `length` será siempre cero, y el campo de igual nombre en el **Record Header** valdrá siempre 4 si el mensaje viaja solo en un fragmento. La figura siguiente expone su simple estructura.

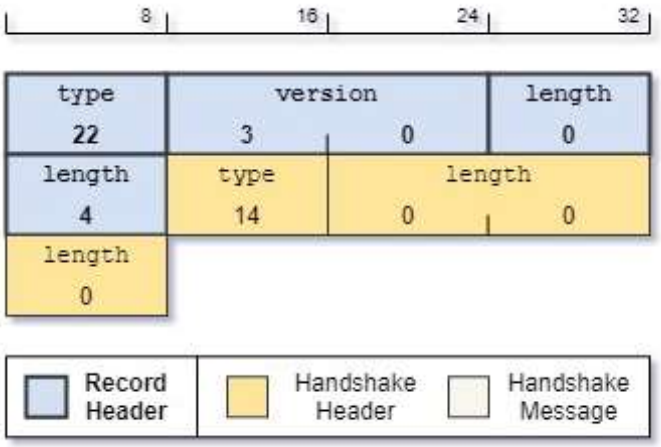

Figura 1.12 - Estructura del mensaje SERVERHELLODONE.

Mensaje CERTIFICATE

Con este mensaje comienza el tercer viaje de la negociación, que se compone de varios que el Cliente ahora envía al Servidor.

El mensaje CERTIFICATE por parte del Cliente es opcional y sucede solo si el Servidor le ha solicitado un certificado. En cuanto a su estructura, es idéntica al mensaje del mismo nombre y también opcional, de la segunda etapa.

Conviene mencionar que el mensaje que el Cliente le envía al Servidor, si es solicitado, debe contener parámetros compatibles con el método de intercambio de claves que ya se viene acordando.

Por ejemplo, si se optó por el método Diffie-Hellmann con parámetros fijos, entonces el grupo y el generador enviados por el Cliente deben coincidir con los recibidos por parte del Servidor.

La autentificación del Cliente comienza con este mensaje y se completa más adelante con el mensaje CERTIFICATEVERIFY.

Mensaje CLIENTKEYEXCHANGE

Se trata del primer mensaje obligatorio de la tercera etapa y uno de los más importantes en todo el *handshake*, dado que consiste en enviar al Servidor los datos necesarios para comenzar efectivamente a proteger las comunicaciones. A estos datos se los suele denominar *key material*, o material para generar claves.

Como ya viene sucediendo en el proceso, su contenido depende del método de intercambio de claves seleccionado. Por lo tanto, en este título vamos a analizar cada caso en detalle por separado, tal como lo hace la propia RFC 6101 [1].

Record Header	
`type`	22 (0x16). Código mensaje ***Handshake***.
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	2 *bytes*. Tamaño del fragmento completo.
Handshake Header	
`type`	16 (0x10). Mensaje CLIENTKEYEXCHANGE.
`length`	3 *bytes*. Tamaño del mensaje.
CLIENTKEYEXCHANGE	
`EncryptedPreMasterSecret` `ClientDiffieHellmanPublic`	El cuerpo del mensaje que depende del método de intercambio de claves.

Tabla 1.10 - Campos del mensaje CLIENTKEYEXCHANGE.

El primer caso corresponde al uso de RSA como método para intercambiar claves, donde el cuerpo completo del mensaje contiene a la clave `pre_master_secret` de 48 bytes de tamaño.

Esta clave viaja cifrada, ya sea utilizando la clave pública presente en el certificado del Servidor, o una temporaria generada en el mensaje SERVERKEYEXCHANGE.

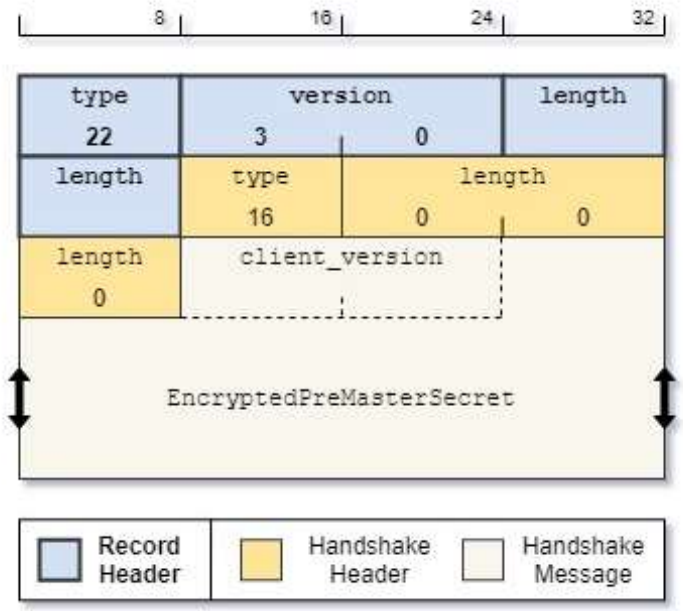

Figura 1.14 - Mensaje CLIENTKEYEXCHANGE con RSA.

El segundo caso concierne al uso del método Diffie-Hellman, ya sea en modo efímero (DHE) o anónimo.

El mensaje contiene el parámetro público Yc, y su tamaño. Recordemos que para DH con parámetros fijos, este mensaje no es necesario dado que el Cliente ya envió previamente su certificado con estos datos.

El Servidor entonces descifra la clave `premaster_secret` con su propia clave privada en el caso de RSA, o bien la calcula por su lado si se usa DH.

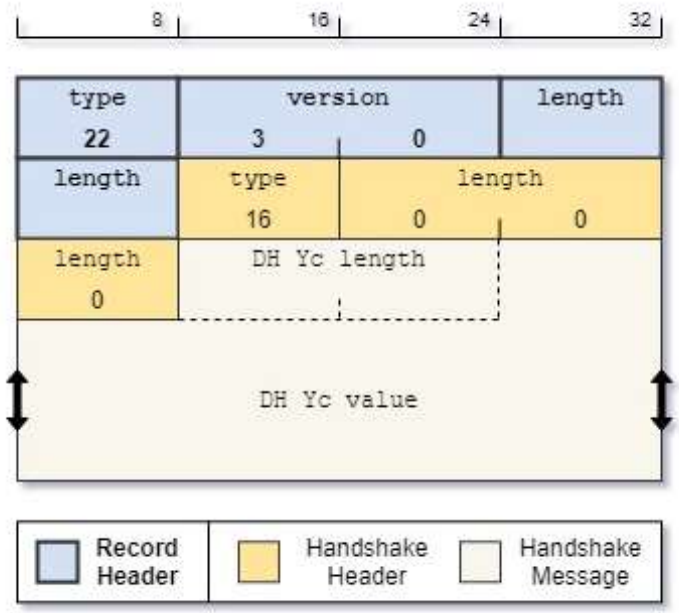

Figura 1.15 - Mensaje CLIENTKEYEXCHANGE con DH efímero o anónimo.

Mensaje CERTIFICATEVERIFY

Si el Cliente envió previamente su certificado, entonces debe completar la autentificación probando que tiene la clave privada asociada a la clave pública del mismo.

Esta comprobación se realiza preparando un mensaje firmado digitalmente con su clave privada, el cual contiene la concatenación de todos los mensajes desde el CLIENTHELLO hasta el anterior a éste.

Debemos aclarar que para los certificados tipificados por la RFC 6101 [1] como `rsa_fixed_dh` y `dss_fixed_dh`, es decir, aquellos con parámetros fijos de Diffie-Hellman, este mensaje no se envía.

Record Header	
`type`	22 (0x16). Código de mensaje ***Handshake***.
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	2 *bytes*. Tamaño del fragmento completo.
Handshake Header	
`type`	15 (0x0F). Mensaje CERTIFICATEVERIFY.
`length`	3 *bytes*. Tamaño del mensaje.
CLIENTKEYEXCHANGE Message	
`Signature`	*Hash* firmado de todos los mensajes previos.

Tabla 1.11 - Campos de un mensaje CERTIFICATEVERIFY.

El cómputo de la firma digital `Signature` que se envía en el cuerpo del mensaje, depende de si el certificado del Cliente usa RSA o DSA para firmar.

```
CertificateVerify.signature.md5_hash:
    MD5(master_secret + pad_2 +
MD5(handshake_messages + master_secret + pad_1));
Certificate.signature.sha_hash:
    SHA(master_secret + pad_2 +
SHA(handshake_messages + master_secret + pad_1));
```

Si el certificado usa RSA como algoritmo de firma, entonces se calculan los dos *hashes* que se ven arriba por separado, se concatenan y se aplica la firma digital sobre ese resultado. Con DSA, solo se usa la fórmula inferior.

El término más relevante de la fórmula es `handshake_messages`, que es como dijimos, la unión de todos los mensajes enviados menos este.

Los rellenos `pad1` y `pad2` son los mismos que mencionamos para mensajes anteriores, con valores repetidos 48 veces para MD5 y 40 veces para SHA-1, mientras que `master_secret` es la clave simétrica negociada.

En ambos casos, el Servidor verifica los datos utilizando la clave pública presente en el certificado enviado por el cliente.

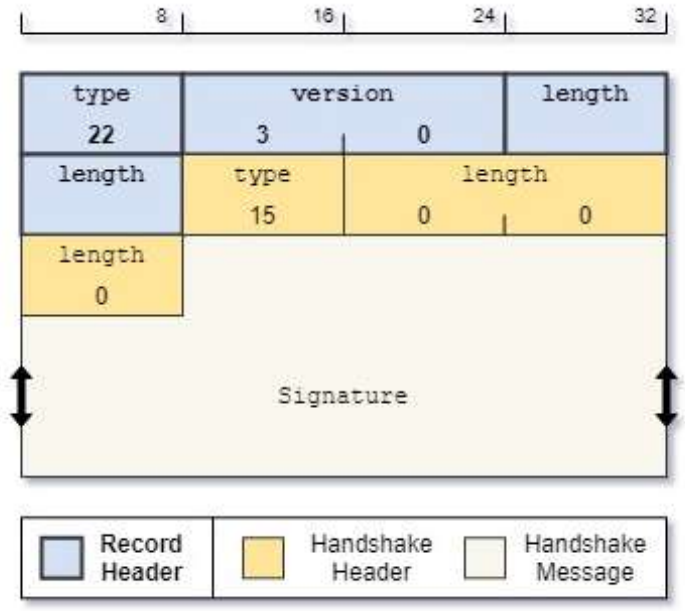

Figura 1.16 - Estructura de un mensaje CERTIFICATEVERIFY.

Mensaje CHANGECIPHERSPEC

Mencionamos este mensaje en este punto, solo para mantener el orden expuesto en la figura 1.4. Se trata de un mensaje que pertenece a otro subprotocolo, y lo describiremos en la sección que corresponde.

Mensaje FINISHED

Siempre inmediatamente después de la señal CHANGECIPHERSPEC se produce este mensaje que es el primero en ser protegido con las claves y algoritmos recién negociados.

Su finalidad es la de verificar que todo el proceso de autentificación y negociación de claves ha terminado en forma exitosa.

Para completar tal verificación, se realiza el cómputo de un *hash* de todos los mensajes enviados en dos versiones. Una vez con el algoritmo MD5, y otra vez con SHA-1.

Record Header	
`Type`	22 (0x16). Código de mensaje ***Handshake***.
`Version`	3.0 (0x03,0x00). Versión de SSL.
`length`	Valor 56 o 60, según el algoritmo utilizado.
Handshake Header	
`Type`	20 (0x14). Mensaje FINISHED.
`length`	Valor 36. Tamaño de los dos *hashes*.
FINISHED Message	
`md5_hash` `sha_hash`	Se envían los dos *hashes* al mismo tiempo.

Tabla 1.12 - Campos de un mensaje FINISHED.

Por primera vez, y aprovechando que este mensaje ya viaja protegido, la figura siguiente exhibe tanto el cálculo del MAC al final como el posterior cifrado resaltado en color rojo, donde ambos son parte del procesamiento que realiza el subprotocolo ***Record***.

En realidad, esto ya venía sucediendo en todos los mensajes del *handshake*, y no se mostró en las figuras anteriores para preservar la simplicidad. Recordemos que siempre está vigente algún método de cifrado y de autentificación, solo que inicialmente son nulos, a causa de utilizar el conjunto de cifrado SSL_NULL_WITH_NULL.

Cabe comentar asimismo que una vez enviado el FINISHED no es necesaria ninguna confirmación desde el otro extremo, y se puede comenzar a transmitir datos inmediatamente.

Si bien el mensaje incluye a los dos *hashes* al mismo tiempo, solo uno de ellos corresponde al método establecido por el conjunto de cifrado acordado. Por lo tanto el cálculo del MAC final, corresponde solo a uno de ellos.

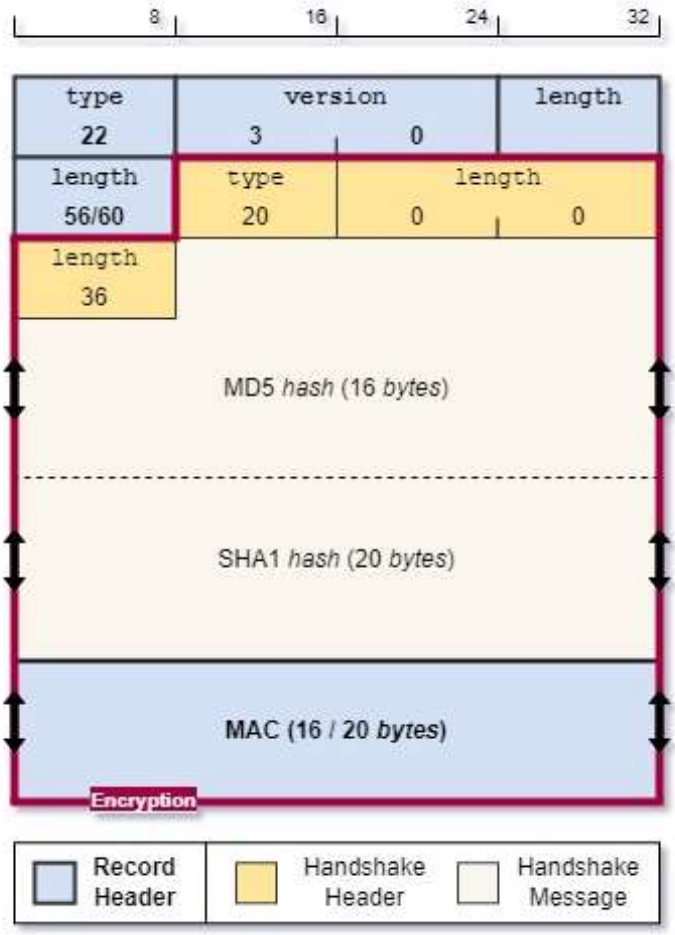

Figura 1.17 - Estructura de un mensaje FINISHED.

El cómputo de ambos *hashes*, que en realidad también son MACs, se realiza de la siguiente forma:

```
md5_hash: MD5(master_secret + pad2 +
MD5(handshake_messages + Sender + master_secret + pad1));

sha_hash: SHA(master_secret + pad2 +
SHA(handshake_messages + Sender + master_secret + pad1));
```

En cada fórmula podemos notar que una vez se realiza la concatenación de todos los mensajes `handshake_messages` enviados desde el inicio hasta el previo, pero sin contar al CHANGECIPHERSPEC que, como dijimos, no pertenece al subprotocolo ***Handshake***.

Aparece también un nuevo actor denominado `Sender`, que corresponde al Cliente o al Servidor según estas definiciones constantes:

```
enum { client(0x434C4E54), server(0x53525652) } Sender;
```

Y por supuesto, ya está en vigencia la clave `master_secret` y los elementos adicionales `pad1` y `pad2` que sirven para completar los bloques.

En este punto hemos finalizado la descripción del complejo proceso de “apretón de manos inicial”. Como mencionamos antes, una vez emitido el mensaje FINISHED cualquiera de los dos extremos puede empezar a transmitir datos sin necesidad de una confirmación.

A continuación entonces, pasaremos a describir los tres subprotocolos restantes, igualmente relevantes pero de mayor sencillez.

1.4 Subprotocolo *Change Cipher Spec*

Un mensaje del tipo CHANGECIPHERSPEC es simplemente una señal que uno de los extremos envía al otro, indicando que cambia a partir de la misma el conjunto de cifrado y por ende la estrategia de protección criptográfica.

El diseño de SSL ha previsto el considerar a estos mensajes en un subprotocolo por separado, es decir, que no corresponden al ***Handshake***. Sin embargo, la señalización sucede dentro del "apretón de manos".

Record Header	
`type`	20 (0x14). Código del ***Change Cipher Spec.***
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	Valor 1 fijo.
ChangeCipherSpec	
`type`	1 (0x01). Mensaje FINISHED.

Tabla 1.13 - Campos de un mensaje CHANGECIPHERSPEC.

Dado que se trata de un mensaje perteneciente a otro subprotocolo, y su protección criptográfica corresponde al conjunto de cifrado activo y no al pendiente como ocurre con su sucesor inmediato el FINISHED, debe ser transmitido en un registro por separado.

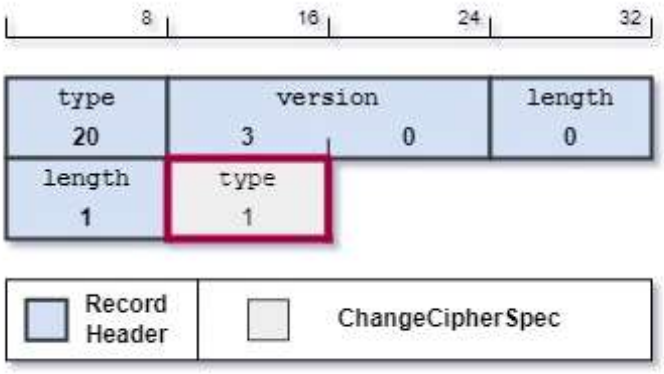

Figura 1.18 - Estructura de un mensaje CHANGECIPHERSPEC.

1.5 Subprotocolo *Alert*

De manera similar a lo que ocurre con el subprotocolo anterior, ***Alert*** también constituye una señalización. O bien se utiliza para marcar el cierre ordenado de las comunicaciones, o para evidenciar algún problema durante la transmisión de los datos.

En esencia, cada mensaje que se envía como alerta contiene un nivel de severidad (*alert level*) y su descripción codificada (*alert description*).

Record Header	
`type`	21 (0x15). Código del mensaje **Alert.**
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	Valor 2 (0x02) fijo.
Alert	
`level`	1 (0x01) / 2 (0x02). Warning / Fatal.
`description`	1 *byte*. Código de la descripción del alerta.

Tabla 1.14 - Campos de un mensaje ALERT.

En la siguiente figura, podemos notar que al igual que el subprotocolo ***Change Cipher Spec***, ambos poseen una estructura simple.

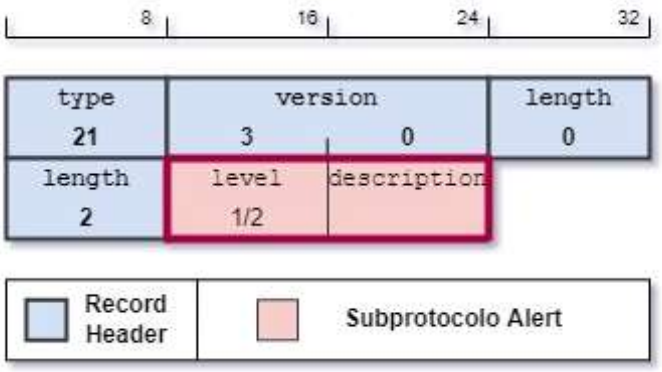

Figura 1.19 – Estructura de un mensaje ALERT.

Amén de la simpleza señalada, el subprotocolo ***Alert*** conlleva ciertas particularidades que lo convierten en una especificación un tanto más compleja.

La RFC 6101 [1] define dos tipos de mensajes de alerta, los de cierre (*Closure Alerts*) y los que avisan sobre un error (*Error Alerts*), como mencionamos al principio.

El documento también incluye la lista de los códigos de alerta, y sus nombres respectivos. En cuanto a los niveles de severidad, solo existen dos: el de "advertencia" (*Warning*) con valor 1 y el "fatal" (*Fatal*) con valor 2.

Existe un único mensaje de cierre ordenado denominado `close_notify`, y cualquiera de las partes puede enviarlo para iniciar el proceso. Este mensaje tiene nivel de severidad igual a *Warning*. Una vez enviado, se terminan de intercambiar los datos pendientes y luego se ignora cualquier otro mensaje posterior a la señal.

Es importante tomar en cuenta que las conexiones deben ser adecuadamente cerradas mediante el intercambio de estos mensajes. Sin ellos, la sesión no puede reanudarse.

Con respecto a los mensajes de error, que son todos los restantes, los que tienen nivel de severidad *Fatal* producen el cierre inmediato de la conexión. Pueden continuar otras conexiones bajo la misma sesión, pero el identificador de la misma debe invalidarse para que no se puedan establecer nuevas.

En el caso de errores con severidad igual a *Warning*, queda a discreción de quien lo recibe su interpretación. Si el error no tiene asociado un nivel de severidad, entonces lo determina quien lo envía.

La lista completa de mensajes de alerta se muestra en la tabla a continuación, y comprende tan solo a 12 elementos, con un único mensaje de cierre y 11 diferentes avisos de error. Los resaltados en naranja son de severidad *Fatal*, y en amarillo los clasificados como *Warning*.

Mensaje de Alerta	Cód.	Descripción Breve
close_notify	0	Una parte notifica a la otra que no enviará más mensajes en esta conexión.
unexpected_message	10	Mensaje inadecuado. Esto nunca debe ocurrrir.
bad_record_mac	20	MAC incorrecta en el subprotocolo **Record**.
decompression_failure	30	Problemas al descomprimir datos.
handshake_failure	40	Falla en la negociación del **Handshake**.
no_certificate	41	No hay un certificado disponible.
bad_certificate	42	El certificado provisto está corrupto.
unsupported_certificate	43	El certificado provisto no está soportado.
certificate_revoked	44	Certificado revocado por el firmante.
certificate_expired	45	El certificado está vencido.
certificate_unknown	46	Cualquier otro error de certificados.
Illegal_parameter	47	Algún campo del Handshake no es consistente.

Tabla 1.15 - Lista de mensajes del subprotocolo **Alert**.

1.6 Subprotocolo *Application Data*

Habiendo completado sin errores el complejo proceso de “apretón de manos” inicial, o bien luego de reanudar una sesión ya iniciada, finalmente llega el momento de intercambiar datos.

El subprotocolo ***Application Data*** tiene esa finalidad, el transmitir en forma segura desde un extremo al otro los datos de la capa de Aplicación del modelo TCP/IP.

Su funcionamiento es sencillo, y consiste en encapsular y procesar esos datos según el mecanismo previsto por el subprotocolo ***Record*** y los parámetros criptográficos corrientes de la máquina de estados.

Record Header	
`type`	23 (0x17). Código del ***Application Data*.**
`version`	3.0 (0x03,0x00). Versión de SSL.
`length`	2 *bytes*.
Application Data	
`data`	Datos de la capa de Aplicación.

Tabla 1.16 - Campos del subprotocolo **Application Data**.

Esto implica el tomar los datos de la capa de Aplicación sin importar su contenido y primero fragmentarlos, luego opcionalmente comprimir cada fragmento, agregarles protección criptográfica, cifrar cada resultado y por último añadirles un encabezado encapsulándolos.

El extremo que recibe la información a su vez realiza el proceso inverso, removiendo el encabezado, descifrando cada fragmento, verificando, descomprimiendo y armando de nuevo los datos a partir de su ensamble.

La estructura interna de cada mensaje con los datos intercambiados depende del tipo de cifrado elegido.

El caso en el que se aplica un cifrado por flujo (*stream cipher*), es el más simple y puede verse en la siguiente figura:

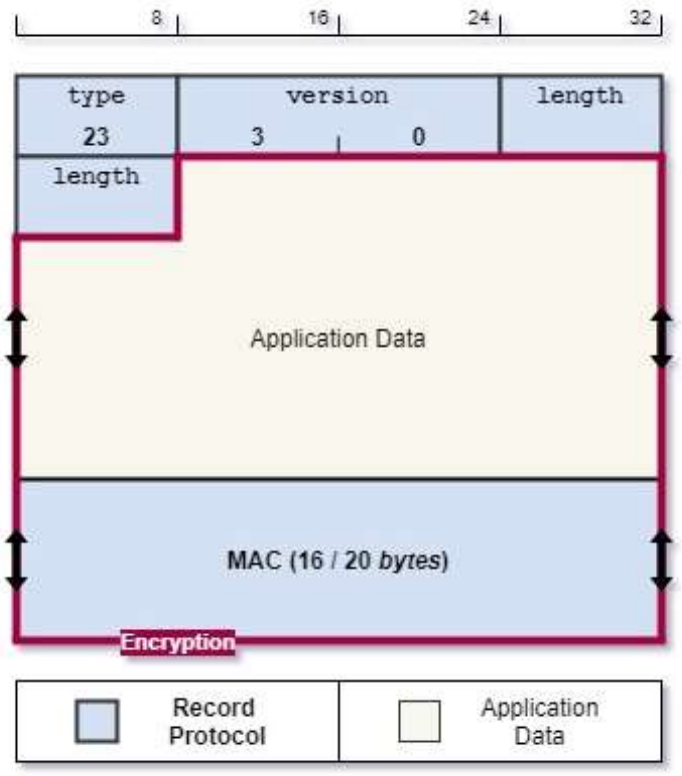

Figura 1.20 - Subprotocolo ***Application Data*** con cifrado por flujo.

En ella se puede apreciar el procesamiento completo de un fragmento de datos proveniente de la capa de Aplicación, incluyendo el MAC calculado que puede ocupar 16 o 20 *bytes* según el algoritmo utilizado, y el cifrado en líneas rojas que incluye a todo lo anterior.

Cuando el método de cifrado seleccionado es el que utiliza bloques (*block cipher*), al mensaje se le debe agregar un relleno (padding) al final y antes de realizar ese cifrado.

El mismo esquema de procesamiento se ejemplifica en la segunda figura, donde se puede ver el relleno al final, con un *byte* que indica el tamaño.

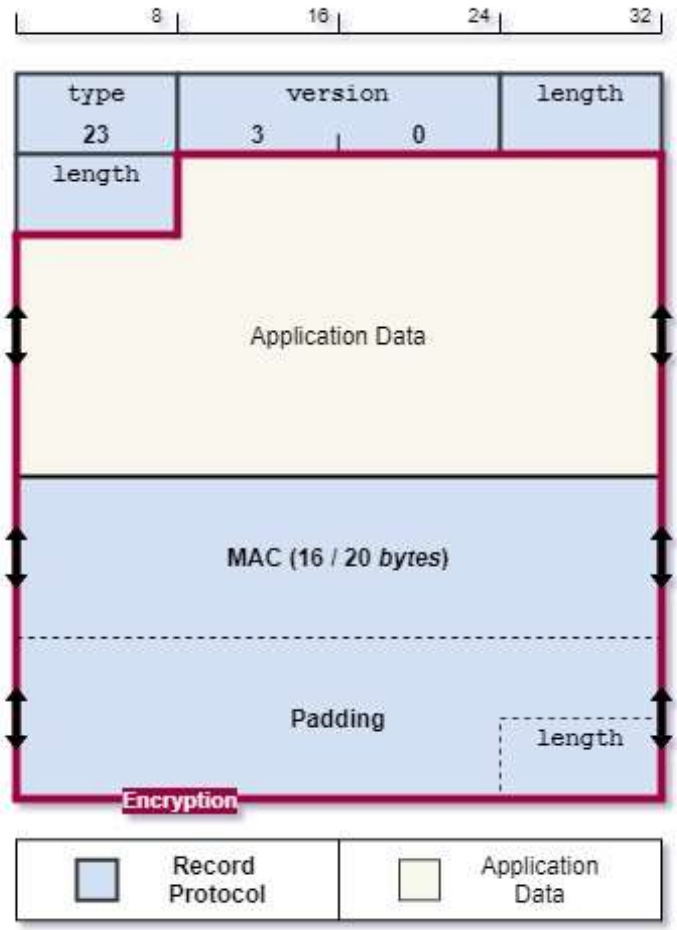

Figura 1.21 – Subprotocolo ***Application Data*** con cifrado por bloque.

2. Protocolo TLS 1.2

Transport Layer Security (TLS) es el nombre con el cual la IETF designó a la nueva y hoy vigente versión del protocolo. La primera publicada fue la 1.0 de 1999 vía la RFC 2246 [5], y la segunda denominada 1.1 vio la luz en el 2006 a través de la RFC 4346 [6].

En esta sección nos ocuparemos principalmente de la tercera versión de TLS la 1.2, del año 2008 cuya especificación es la RFC 5246 [7]. Nótese cómo la IETF ha asignado a las sucesivas publicaciones, números que terminan en los dos mismos dígitos.

Las versiones 1.0 y 1.1 han sido declaradas obsoletas (*deprecated*) por esa organización mediante la RFC 8996 [8], y en ese documento se pueden revisar los motivos técnicos y de seguridad por los cuales se tomó tal decisión.

Por su parte la versión 1.2 si bien no ha sido descartada, ya no es la vigente, reemplazada en 2018 por la 1.3 de la cual nos ocuparemos en la sección siguiente. Curiosamente y siendo esto parte de lo que se plantea en el presente trabajo, TLS 1.2 sigue siendo la versión más utilizada.

A continuación analizaremos las diferencias que existen entre SSL y TLS en general, y donde corresponda describiremos cambios que se introdujeron en cada versión específica, pero siempre organizando el análisis por tema y dando prioridad a lo remanente en la versión 1.2.

Cabe tomar en cuenta que TLS es estructuralmente idéntico a su predecesor, compuesto por las dos mismas capas, y los mismos subprotocolos, aunque se prevé la creación de nuevos. La nomenclatura de los conjuntos de cifrado es la misma pero ahora con el prefijo TLS.

Las cuestiones que sí se modificaron, entre las cuales la más significativa es sin duda el mecanismo de extensiones de TLS 1.2, se describen en los puntos siguientes.

2.1 Sesiones y Conexiones

Uno de los primeros cambios que introduce el protocolo TLS en general, es la composición de los datos que se almacenan en la máquina de estados.

Al igual que sucedía en SSL, se sigue manteniendo un registro de datos sobre sesiones y conexiones. Los datos que definen a una sesión y que son negociados por el protocolo ***Handshake*** siguen siendo exactamente los mismos y los presentamos nuevamente aquí en forma abreviada:

Nombre Original	Descripción
`session identifier`	Identificador de sesión generado por el Servidor.
`peer certificate`	El certificado versión X.509v3 del otro extremo.
`compression method`	Algoritmo de Compresión.
`cipher spec`	Algoritmos de cifrado y de MAC, con información adicional.
`master secret`	La clave maestra de 48 *bytes* compartida entre las partes.
`is resumable`	Si la sesión es reanudable.

Tabla 2.1 – Elementos del estado de una sesión en TLS.

Sin embargo, con las conexiones sucede algo diferente. No solo se cambiaron, reordenaron y agregaron elementos, sino que además se distingue entre “estados de conexión” (*Connection States*) y “parámetros de seguridad” (*Security Parameters*) los cuales se muestran en las dos tablas siguientes.

Cabe notar en la segunda tabla, que aparece entre los elementos nuevos, el “*PRF algorithm*”, del cual hablaremos más adelante.

Nombre Original	Descripción
`compression state`	**Estado de la Compresión** El estado actual del algoritmo de compresión.
`cipher state`	**Estado del Cifrado** El estado actual del algoritmo de cifrado.
`MAC secret`	La clave MAC utilizada para esta conexión.
`sequence number`	**Número de Secuencia** Valor de 64 *bits* asignado a cada registro transmitido.

Tabla 2.2 - Elementos del estado de una conexión en TLS.

Nombre Original	Descripción
`connection end`	**Extremo de la Conexión** Si se trata del “Cliente” o del “Servidor”.
`PRF algorithm`	Algoritmo utilizado para derivar claves a partir de la `master secret`.
`bulk encryption algorithm`	**Algoritmo de Cifrado** El algoritmo y sus parámetros: Clave, tipo y otros datos.
`MAC algorithm`	**Algoritmo de MAC** Utilizado para la autentificación de los mensajes.
`compression algorithm`	**Algoritmo de Compresión** Valor de 64 *bits* asignado a cada registro transmitido.
`master secret`	La clave maestra de 48 *bytes* compartida entre las partes.
`client random`	Valor aleatorio de 32 *bytes* provisto por el Cliente.
`server random`	Valor aleatorio de 32 *bytes* provisto por el Servidor.

Tabla 2.3 - Parámetros de seguridad de una conexión en TLS.

Estas modificaciones no suponen cambios de fondo en el diseño del protocolo TLS con respecto a su predecesor. La máquina de estados sigue funcionando igual, usando el mecanismo de lectura y escritura y parámetros corrientes y pendientes, tal como lo explicamos para SSL.

2.2 Generación de Claves

Otro de los cambios implementados en TLS que revisten interés, es el método por el cual se generan las claves necesarias para proteger a la información transmitida.

En primer lugar, debemos revisar la modificación realizada en la función de generación de números aleatorios, conocida como TLS PRF. Esta función en general depende de tres argumentos, que son un secreto (*secret*), una semilla (*seed*), y una etiqueta (*label*).

A su vez, el TLS PRF hace uso de otra función de expansión, cuya definición genérica es:

```
P_hash(secret,seed)= HMAC_hash(secret,A(1) + seed)  +
          HMAC_hash(secret,A(2) + seed)  +
         HMAC_hash(secret,A(3) + seed)  + …

                A(0) = seed
          A(i) = HMAC_hash(secret, A(i-1)
```

En las versiones 1.0 y 1.1 del protocolo, se utilizan los algoritmos de *hash* MD5 y SHA-1 en forma combinada, práctica muy común en criptografía. Sin embargo, para TLS 1.2, se utiliza uno solo, por ejemplo SHA-256 que es mucho más seguro, por lo que centraremos nuestra atención en la versión más reciente y de mayor simplicidad.

```
PRF(secret,label,seed) = P_hash(secret,label+seed)
```

Donde la función *hash* utilizada proviene del conjunto de cifrado negociado, siendo el caso típico el de SHA-256.

La especificación de TLS 1.2 prevé la posibilidad de agregar otros valores al parámetro de seguridad `prf_algorithm`, aunque el único valor propuesto es `tls_prf_sha256`.

El uso principal de la función TLS PRF es para generar las claves que mencionamos al principio de este punto. En este caso, la `pre_master_secret` que surge del intercambio de claves asimétrico es uno de los argumentos de la siguiente construcción y redunda en un parámetro del estado de sesión:

```
master_secret = PRF(pre_master_secret, "Master Secret",
            client_random + server_random)
```

Una vez generada la `master_secret` de 48 bytes la misma opera como fuente de entropía para armar el `key_block`:

```
key_block = PRF(master_secret,
"key expansion",server_random + client_random)
```

El cual es a su vez particionado en tamaños adecuados logrando las siguientes claves:

```
client_write_MAC_secret
server_write_MAC_secret
client_write_key
server_write_key
client_write_IV
server_write_IV
```

2.3 Los Registros en Línea

La versión TLS 1.1 introduce una modificación de importancia a la forma de documentar los parámetros relacionados con el protocolo y sus valores tales como las alertas, los conjuntos de cifrado (*Cipher Suites*) y muchos otros.

En vez de que la lista completa de los mismos esté incluida como se venía haciendo en el propio texto de cada RFC, se crearon "Registros" (*Registries*) en línea en el sitio web iana.org.

Este cambio por supuesto es una mejora a la forma de administrar toda esa información. Si se desea agregar o modificar cualquier parámetro, simplemente se realiza ese cambio en el *Registry* correspondiente y no hace falta actualizar la RFC.

Uno de esos registros se encuentra en la página:

https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml

E incluye por ejemplo la lista actualizada de los *Cipher Suites* y de las alertas vigentes. En esta otra página, se pueden encontrar las "Extensiones" que introdujo la versión TLS 1.2, de gran relevancia y que veremos en detalle más adelante.

https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml

En general, toda la información relacionada con el protocolo TLS se encuentra clasificada dentro del subsitio "*Protocol Registries*" de IANA, bajo el título "*Transport Layer Security (TLS)*":

https://www.iana.org/protocols

2.4 Conjuntos de Cifrado

La especificación original de SSL versión 3.0, hoy plasmada en la RFC 6101 [1] de carácter histórico incluye el listado de los 31 conjuntos de cifrado (*Cipher Suites*) utilizables en los mensajes CLIENTHELLO y SERVERHELLO durante el *handshake*.

Desde la propuesta inicial hasta la llegada de la versión TLS 1.2 del protocolo, esa lista ha crecido notablemente con documentos complementarios, y ha sido traslada a un registro especial mantenido en Internet por IANA, tal como mencionamos en el punto anterior.

Haremos entonces un recorrido por los distintos cambios introducidos a partir de cada versión previa de TLS, analizando sus particularidades.

2.2.1 Cambios en TLS 1.0

Como primera modificación general todos los conjuntos de cifrado pasan a tener el prefijo “TLS” en vez de “SSL”, tal como en el siguiente ejemplo:

SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA

Pasa a ser:

TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA

Un cambio menor, adicional, es que los *Cipher Suites* que incluyen al método de intercambio de claves FORTEZZA se eliminaron, por lo que la lista de 31 conjuntos propuestos por SSL se disminuye en tres, quedando 28.

TLS 1.0 introduce una transformación importante en la forma en que se realiza el cómputo del MAC para la autentificación de los mensajes, que en SSL era una construcción “hecha a mano”:

```
hash(MAC_write_secret + pad_2 + hash(MAC_write_secret +
        pad_1 + seq_num + SSLCompressed.type +
    SSLCompressed.length + SSLCompressed.fragment));
```

Esta fórmula pasa a ser la siguiente, más afín a la construcción estándar que se conoce como HMAC [4].

```
HMAC_hash(MAC_write_secret, seq_num + TLSCompressed.type
    + TLSCompressed.version + TLSCompressed.length +
            TLSCompressed.fragment));
```

Notemos que en la primera, se pueden visualizar los *hash* interno y externo más los *paddings* mientras que en la segunda, todos son parte de la definición del algoritmo. Este corresponde al parámetro MAC de seguridad de la conexión que proviene del conjunto de cifrado negociado.

El argumento `MAC_write_secret` puede ser el que corresponde al Cliente o el que corresponde al Servidor según quién realice el envío.

También se puede observar que se han incorporado al cómputo el número de secuencia `seq_num` que agrega una protección adicional ante ciertos ataques y el campo `.version` que antes no se consideraba.

Otro aspecto novedoso que agrega esta versión de TLS es el concepto de conjunto de cifrado “obligatorio” (*mandatory*).

El requerimiento consiste en hacer que cualquier versión del protocolo deba implementar como mínimo un conjunto de cifrado específico, que en TLS 1.0 corresponde a:

TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA

Cabe observar que en este *Cipher Suite* obligatorio, se ha optado por 3DES como el método de cifrado, el más seguro en ese momento.

Merece ser incluida bajo las novedades que corresponden a TLS 1.0 la adición de un grupo especial de conjuntos de cifrado conocidos como los "Camellia". El nombre Camellia refiere a un método de cifrado en bloque en modo CBC desarrollado en Japón por Mitsubishi y la *Nippon Telegraph and Telephone Corporation* (NTT).

La primera RFC publicada al respecto fue la 4132 [9] del año 2005 que introdujo 12 *Cipher Suites* contemplando a este método, los sistemas de intercambio de claves habituales y el algoritmo de hash SHA1. más adelante en 2010 se agregó la RFC 5932 [10] que agrega 12 más, esta vez con SHA256.

Por último, se agregó una RFC adicional con status de "informativa" (*informational*) numerada 6367 [11] que propone más variantes de conjuntos de cifrado. Algo similar ocurre con otro grupo utilizado en Corea denominado ARIA, detallado en la RFC 6209 [12].

2.2.2 Cambios en TLS 1.1

La modificación más significativa aportada por la versión 1.1 del protocolo en relación con los conjuntos de cifrado es sin duda la creación de los "registros" (*registries*) en línea.

A partir de la RFC 4346 [6] entonces, ya no es necesario el modificar la especificación del protocolo para gestionarlos. La información actualizada sobre los *Cipher Suites* vigentes se encuentra de ahora en más en la página de IANA.

Sin embargo y como sucede en las especificaciones anteriores del protocolo, la que corresponde a TLS 1.1 también incluye un listado estándar de los mismos, que difiere de la versión anterior según las consideraciones expuestas a continuación.

En primer lugar, se prohíbe la negociación de conjuntos de cifrado que incluyan algoritmos “exportables” (*Export-Grade*), y solo pueden ofrecerse con el fin de brindar compatibilidad con versiones anteriores.

En segundo lugar, se incorporan a TLS 1.1 los conjuntos que hacen uso de AES como método de cifrado simétrico. Ellos están enumerados en la RFC 3268 [13].

Por último, se agregan los *Cipher Suites* que utilizan Kerberos como mecanismo de intercambio de claves y autenticación por vía simétrica, según lo que propone la RFC 2712 [14] pero dejando de lado aquellos que son exportables, por el motivo planteado más arriba.

En TLS 1.1 el conjunto de cifrados obligatorio se cambia al que presenta esta denominación: TLS_RSA_WITH_3DES_EDE_CBC_SHA.

2.2.3 Cambios en TLS 1.2

La especificación de esta versión de TLS publicada como RFC 5246 [7] incluye 37 conjuntos de cifrado estándar, a los cuales se les deben sumar otros agregados o sugeridos por documentos complementarios.

Cabe notar que en conjunto, todas las versiones de TLS hasta la 1.2 suman un total de 229 *Cipher Suites*, cifra que es demasiado abultada, con gran propensión a problemas de seguridad y compatibilidad.

En ese primer grupo, aparecen como en las anteriores versiones, un conjunto predeterminado:

TLS NULL WITH NULL NULL

Y otro conjunto obligatorio, que en este caso corresponde a:

TLS RSA WITH AES 128 CBC SHA

Se puede notar que en el nuevo listado propuesto de *Cipher Suites* sólo figuran los algoritmos de cifrado por bloques 3DES y AES, y RC4 para el método por flujo.

En cuanto al mecanismo de *hash*, en general se utiliza SHA-1, y versiones más seguras como SHA-256. Sobrevivieron también algunos pocos conjuntos que hacen uso de MD5.

Una modificación de gran importancia es el agregado de los algoritmos de "cifrado autentificado con datos adicionales", AEAD (*Authenticated Encryption with Additional Data*), propuestos y descritos en la RFC 5116 [15].

Estos algoritmos como el nombre lo indica permiten combinar en una sola transformación criptográfica, el cifrado y la autentificación. De esta manera proveen confidencialidad e integridad al mismo tiempo, y en forma más eficiente y segura que con el método tradicional en dos etapas, eliminando también la discusión sobre si conviene cifrar antes de autentificar o viceversa.

Es de interés mencionar que la RFC 5116 [15] define tanto una interfaz para el uso de tales algoritmos, como un nuevo registro por parte de IANA para los mismos, el cual hoy se encuentra en esta página:

https://www.iana.org/assignments/aead-parameters/aead-parameters.xhtml

La inclusión de algoritmos AEAD según lo mencionado implica un cambio en el proceso de protección sobre los datos que realiza el subprotocolo ***Record***, y por ende cierta modificación en el diseño de TLS.

Es por ello por lo que la RFC 5246 [7] en la sección 6.2.3 donde refiere a ese proceso de protección (*Record Payload Protection*) explica cómo se debe hacer uso de la interfaz.

Básicamente se trata de llamar a una función de "cifrado AEAD" la cual toma a la estructura `TLSCompressed` y la convierte directamente a otra del tipo `TLSCiphertext` en una sola operación.

El cifrado AEAD toma como entradas los parámetros que se muestran en la siguiente fórmula que plantea la RFC 5246 [7]:

```
AEADEncrypted = AEAD-Encrypt(write_key, nonce,
plaintext,additional_data)
```

Lo que la función llama `plaintext` es en realidad la estructura `TLSCompressed` del subprotocolo ***Record***.

El mecanismo AEAD en general, y que varía según cada algoritmo en particular, hace uso de un `nonce` que es un valor aleatorio que no se repite y que agrega entropía a la transformación.

Asimismo, el parámetro `additional_data` corresponde a "datos adicionales" que se deben autentificar pero no cifrar. En TLS 1.2 este valor se compone de la siguiente forma:

```
additional_data = seq_num + TLSCompressed.type +
TLSCompressed.version + TLSCompressed.length;
```

Por último, dado que AEAD no realiza el cálculo de un MAC luego de cifrar, solo necesita una única clave que puede ser según el caso el `client_write_key` o el `server_write_key`.

La RFC 5116 [15] propone inicialmente 4 algoritmos AEAD, que son variantes del método de cifrado AES en diferentes modos de operación:

Id	Nombre
1	AEAD_AES_128_GCM
2	AEAD_AES_256_GCM
3	AEAD_AES_128_CCM
4	AEAD_AES_256_CCM

Tabla 2.4 - Algoritmos AEAD iniciales.

Sin embargo, estamos hablando de algoritmos y no de conjuntos de cifrado.

Los *Cipher Suites* que corresponden a AES en modo GCM están indicados en la RFC 5288 [16] con carácter de estándar y en la 5289 [17] informativa, mientras que los relacionados con AES en modo CCM se encuentran en la RFC 6655 [18].

AEAD es relevante por las mejoras comentadas y por el hecho de que es un requerimiento en TLS 1.3. En la sección dedicada a esta última versión seguiremos comentando sobre este algoritmo.

Otra de las adiciones que contempla la versión TLS 1.2 es la que permite utilizar las “claves precompartidas” (*PSK – Pre-Shared Keys*), en el proceso de conexión.

Esta variante puede ser aplicable en los casos en que existen recursos limitados, o bien otras complicaciones administrativas que impiden el uso de la infraestructura de claves públicas PKI (*Public Key Infrastructure*) característico del protocolo.

La RFC 4279 [19] con carácter de estándar presenta 12 conjuntos de cifrado complementarios clasificables en tres categorías según su prefijo:

- La primera (TLS_PSK) utiliza claves simétricas para el intercambio de claves.
- La segunda (TLS_DHE_PSK) autentifica con claves simétricas al intercambio de claves Diffie-Hellman efímero.
- La tercera (TLS_RSA_PSK) hace uso de claves públicas para el Servidor, y claves simétricas para el Cliente.

La implementación de claves simétricas precompartidas tiene su efecto obviamente en el diseño del protocolo, más precisamente en el flujo del *handshake*.

Tal como lo expresa gráficamente la propia RFC 4279 [19], los mensajes CERTIFICATE y CERTIFICATEREQUEST del Servidor, más los mensajes CERTIFICATE y CERTIFICATEVERIFY del Cliente ya no tienen sentido y se excluyen del apretón de manos. Además, el mensaje SERVERKEYEXHANGE queda supeditado a ciertos casos especiales.

Caben mencionar aquí algunas RFC adicionales que pueden ser de interés. La RFC 5487 [20] que agrega más conjuntos de cifrado con funciones de hash más recientes tales como SHA-256 y SHA-384, más AES en modo GCM. La RFC 5489 [21] que incluye conjuntos ECDHE_PSK, y la RFC 4785 [22] que describe el uso de PSK sin cifrado.

2.5 Extensiones

El cambio más importante que introduce la versión TLS 1.2 es sin duda alguna el mecanismo de “Extensiones”, el cual ingeniosamente permite agregar funcionalidad al protocolo sin modificar el diseño original, y donde los clientes o servidores que no soportan las extensiones pueden seguir comunicándose con los que sí.

En la propia especificación de la versión que corresponde a la RFC 5246 [7], se explica conceptualmente el cómo deberían funcionar estas extensiones. Sin embargo, la lista inicial de extensiones disponibles fue publicada en un documento complementario, el RFC 6066 [23].

ExtensionType	**#**	**Descripción**
`server_name`	0	Nombre completo FQDN del servidor.
`max_fragment_length`	1	Tamaño máximo de los fragmentos de **Record**.
`client_certificate_url`	2	URL del certificado del Cliente.
`trusted_ca_keys`	3	Cuáles CA son aceptadas por el Cliente.
`truncated_hmac`	4	Limitar el cómputo de la autentificación vía HMAC.
`status_request`	5	Solicitar al Servidor que verifique un certificado.

Tabla 2.5 - Las seis primeras extensiones de TLS 1.2.

Es interesante comentar que, al introducir extensiones que pueden alterar el funcionamiento del protocolo, también deben incluirse nuevos mensajes de alerta. La RFC complementaria entonces agrega una serie de cuatro nuevos mensajes, algunos de ellos relacionados con esos cambios.

En la tabla que se presenta a continuación, se enumeran y describen las nuevas alertas.

Alerta	#	Descripción
`unsupported_extension`	110	El Cliente no soporta una extensión enviada por el Servidor en un mensaje SERVERHELLO.
`certificate_unobtainable`	111	El Servidor no puede encontrar una cadena de certificados recibida en un mensaje CERTIFICATEURL desde el Cliente.
`unrecognized_name`	112	El Servidor no reconoce el nombre de servidor enviado en una extensión por el Cliente.
`bad_certificate _status_response`	113	El Cliente indica al Servidor que ha encontrado problemas con el certificado recibido.
`bad_certificate _hash_value`	114	El Servidor notifica al Cliente que el valor de *hash* del certificado no coincide con el recibido.

Tabla 2.6 - Nuevos mensajes de alerta introducidos por la RFC 6066.

Las extensiones se activan al ser enviadas como parte de los mensajes CLIENTHELLO y SERVERHELLO durante el ***Handshake***.

Dado que siempre se pudo enviar información adicional al final de un mensaje CLIENTHELLO, este agregado no afecta a un servidor TLS que “no las entienda”. Por otra parte, cuando el Servidor responde con extensiones, solamente lo hace enviando las mismas extensiones que recibió del Cliente, y de esta manera también se previene que se lo afecte.

Algo curioso para notar, es que no existe un límite superior para la cantidad de extensiones que se pueden agregar a un mensaje CLIENTHELLO, por lo que en teoría un Cliente podría sobrecargar al Servidor con un mensaje demasiado largo.

Un mensaje CLIENTHELLO que contiene extensiones, las incluye en un bloque de datos al final del mismo. Y la estructura de esta información adicional es muy simple, siendo similar para cada extensión.

En primer lugar, se especifica el tipo de extensión `ExtensionType` en un campo de dos *bytes*. Y luego, un campo de extensión variable, que incluye la información sobre su tamaño total en otros dos *bytes*, el tamaño de los datos en uno o dos *bytes*, y los propios datos.

A modo de ejemplo y para una mejor comprensión, se muestra la extensión signature_algorithms (*Algoritmos de Firma*) que permite al protocolo operar con opciones más robustas:

```
00 0d 00 12 00 10 04 01 04 03 05
01 05 03 06 01 06 03 02 01 02 03
```

En este bloque de valores hexadecimales, el detalle del contenido es el que se describe en la lista:

00 0d – Valor asignado a la extensión `signature_algorithms`.
00 12 - 0x12 (18) *bytes* es el tamaño completo de la extensión.
00 10 - 0x10 (16) *bytes* es el tamaño de la lista de algoritmos.
04 01 – Valor asignado a RSA/PKCS1/SHA256.
04 03 - Valor asignado a ECDSA/SECP256r1/SHA256.
05 01 - Valor asignado a RSA/PKCS1/SHA386.
05 03 - Valor asignado a ECDSA/SECP384r1/SHA384.
06 01 - Valor asignado a RSA/PKCS1/SHA512.
06 03 - Valor asignado a ECDSA/SECP521r1/SHA512.
02 01 - Valor asignado a RSA/PKCS1/SHA1.
02 03 - Valor asignado a ECDSA/SHA1.

El bloque mencionado estaría ubicado en el mensaje CLIENTHELLO a continuación del campo `compression_method`, junto a otras extensiones que también se estén enviando, y sin ningún orden específico para las mismas. Dos *bytes* previos anuncian el tamaño de todas las extensiones.

Aprovechando que hemos utilizado como ejemplo a la extensión denominada `signature_algorithms`, aclaramos que esta es la única a la que se refiere puntualmente la RFC 5246 [7], es decir, la propia especificación del protocolo TLS 1.2. Lo mismo ocurre con el mensaje de alerta `unsupported extension`.

Comenzamos entonces describiendo las primeras seis extensiones en orden, y luego a la que mencionamos arriba. Como venimos haciendo, utilizaremos sus nombres en inglés tal como aparecen en la documentación.

Server Name Indication

Esta extensión permite al Cliente indicar a qué Servidor se desea conectar en particular, en un entorno de "*Virtual Hosting*", es decir, donde varios Servidores Web residen en el mismo proveedor, probablemente con la misma dirección IP y sobre el mismo *hardware*.

Tal funcionalidad es sin duda un avance a favor del despliegue de TLS a gran escala, permitiendo que un Servidor Web pueda alojar varios servidores virtuales y al mismo tiempo ofrecer soporte para el protocolo.

El Cliente envía su extensión SNI incluyendo el nombre completo del Servidor específico al que desea conectarse, en formato FQDN, y solo debe enviar un nombre y no varios.

La RFC 6066 [23] presenta consideraciones sobre cómo implementar correctamente la extensión, y esta referencia profundiza sobre los posibles problemas de seguridad [24].

Maximum Fragment Length Negotiation

Aquí se trata de negociar tamaños de fragmento menores al estándar propuesto desde SSL, que como hemos visto anteriormente es de 2^14=65536 *bytes*.

Esta es una de las extensiones que tienen como propósito ayudar en contextos de pocos recursos tales como memoria o ancho de banda. El mismo documento RFC 6066 [23] habla por ejemplo de entornos *wireless,* lo que tenía sentido a la fecha de su publicación.

El mecanismo es sencillo, y consiste en elegir uno de cuatro valores preestablecidos: 1 (2^9 bytes), 2 (2^10 bytes), 3 (2^11 bytes), y 4 (2^12 bytes).

Una vez negociado, este valor de fragmento menor se aplica a la sesión y sus reanudaciones, y no puede ser cambiado durante la ejecución.

Client Certificate URL

Con el fin de ahorrarse trabajo de cómputo, el Cliente puede elegir enviarle al Servidor la URL de su certificado, en vez de enviarlo completo o la cadena completa en un mensaje CERTIFICATE durante el *handshake*.

Cuando el Cliente hace uso de esta extensión, primero avisa al Servidor a través de la misma y luego el envío del certificado pasa a ser opcional. Esto quiere decir que el Cliente puede o no enviar el mensaje CERTIFICATE más adelante en la negociación.

La RFC 6066 [23] introduce entonces un nuevo tipo de mensaje en el ***Handshake*** denominado CERTIFICATEURL que es la otra opción. Cabe comentar que este mecanismo supone cuestiones de seguridad y no debería ser implementado sin agregar otras protecciones complementarias.

Trusted CA Keys

Cuando el Servidor envía su mensaje CERTIFICATE durante el *handshake*, es posible que el Cliente no tenga registradas las claves de las autoridades certificantes. Esto ocurre con los Clientes que tienen pocas CA registradas en su base de datos local.

Si el Cliente no cuenta con ese registro, entonces rechazará el certificado o la cadena, y la negociación deberá repetirse. Ello supone una pérdida en tiempo y en recursos.

Con esta extensión, puede avisar al Servidor con antelación qué certificados está preparado para aceptar. Para ello, envía en el campo de datos una lista de CA's codificada en alguna de varias formas disponibles, y el Servidor responde con la misma extensión vacía indicando que acepta el mecanismo. De todas formas, el certificado se envía luego en el mensaje CERTIFICATE como de costumbre.

Truncated HMAC

Una vez más en relación con entornos de pocos recursos, se propone aquí el utilizar una versión de menor tamaño del resultado de la construcción denominada HMAC, reduciendo su salida a 80 bits.

Este cambio afecta solamente al proceso de autentificación ejecutado por el subprotocolo ***Record*** y no a la aplicación de HMAC para la generación de claves relacionadas con el mecanismo TLS PRF.

Simplemente citamos en forma breve a la extensión para seguir el orden propuesto, sin mayor detalle considerando que la misma ha dejado de ser recomendada debido a problemas relacionados con su criptoanálisis.

Certificate Status Request

Se trata de otra extensión que ahorra trabajo de cómputo a un Cliente con pocos recursos, y además, es el otro caso en el que se modifica el diseño del protocolo utilizando un nuevo tipo de mensaje en el *handshake*.

Normalmente, cuando se intercambian certificados durante la negociación, el extremos que recibe el certificado es el que se ocupa de verificar su estado, ya sea a través de una lista CRL (*Certificate Revocation List*) que incluye a todos los certificados revocados, o bien por medio del protocolo OSCP (*Online Certificate Status Protocol*) que permite validar los certificados directamente en línea.

Utilizando la extensión `status_request` el Cliente puede transferir esa responsabilidad al Servidor, incluyendo en ella también una lista de las fuentes confiables que desea sean consultadas. Este último hará uso entonces del mensaje CERTIFICATESTATUS inmediatamente después del CERTIFICATE más adelante en el handshake proveyendo una respuesta OCSP completa. Solo se puede enviar una respuesta.

El mecanismo de consulta que genera esta extensión, es denominado "*OCSP Stapling*", por su relación con ese protocolo. Cabe mencionar también que la RFC 6961 [25] propone una nueva versión de la extensión en donde se pueden realizar múltiples consultas.

Signature Algorithms

Habiendo descrito a las seis primeras extensiones introducidas por la RFC 6066 [23], conviene ocuparnos de esta que es la única incluida en la especificación de TLS 1.2.

La extensión permite al Cliente indicarle al servidor cuáles son los pares de algoritmos de firma y de *hash* que es capaz de utilizar en las firmas digitales.

Si no se incluye esta extensión, entonces el Servidor asumirá que el método de *hash* a aplicar es SHA-1, y el de firma será el que corresponda al conjunto de cifrado (*Cipher Suite*) negociado con el Cliente, en particular el algoritmo de intercambio de claves.

Al corriente, estos algoritmos se encuentran enunciados y mantenidos en un registro de IANA por separado para cada uno, en la página correspondiente a parámetros que ya presentamos en la sección “Los Registros en Línea”.

Por último, conviene aclarar que no todos los pares posibles de algoritmos son factibles. Un ejemplo es que DSA, sólo se puede combinar con SHA-1.

La propia especificación del protocolo aclara que “la semántica de esta extensión es un tanto complicada”, debido en parte a que los conjuntos de cifrado incluyen algoritmos de firma pero no de *hash*.

Hay entonces aclaraciones que implican reglas de uso enunciadas no solo en la sección que trata este tema, sino también en las secciones correspondientes a los mensajes denominados SERVERCERTIFICATE y SERVERKEYEXCHANGE.

Para comprender correctamente el alcance del uso o no de esta extensión, se recomienda revisar la especificación de TLS 1.2 detalladamente.

2.6 Más Extensiones

La página de IANA que mencionamos en la sección sobre los "*Registries*" presenta hoy la lista completa de extensiones disponibles. A continuación, veremos algunas de las adicionales que se incluyeron a partir de otras RFC, y que son de interés para comprender los cambios introducidos por TLS 1.2.

De las extensiones que presenta la lista, cabe notar que algunas han tomado gran importancia tal como la que corresponde a las curvas elípticas, y otras han caído en desuso o sido desestimadas.

User Mapping

La extensión `user_mapping` es descrita en la RFC 4681 [26] que a su vez está basada en un mecanismo general de intercambio de datos suplementarios para TLS propuesto por la RFC 4680 [27].

Lo que la misma habilita, es la posibilidad de que el Cliente envíe al Servidor información adicional que permite comprobar ("mapear") la cuenta de acceso de un usuario ante la base de datos que se encuentre en uso. Es decir, permite implementar autorizaciones de usuarios.

Se trata de otra más de las extensiones que modifican el diseño original del protocolo. En este caso, previo haber inicializado la misma incluyéndola en el CLIENTHELLO se agrega un mensaje SUPPLEMENTALDATA justo después del SERVERHELLO, y el Cliente responde con otro mensaje SUPPLEMENTALDATA después del SERVERHELLODONE.

Finalmente, será el Servidor quien interprete a su criterio los datos suplementarios recibidos.

ExtensionType	#	Descripción
user_mapping	6	Vincular a usuarios con sus cuentas.
client_authz	7	Agregar autorización para el Cliente.
server_authz	8	Agregar autorización para el Servidor.
cert_type	9	Permite utilizar certificados no X.509.
elliptic_curves	10	Permite utilizar curvas eíípticas en TLS.
ec_point_formats	11	Utilizar compresión en algunos parámetros de curvas.
srp	12	Intercambio de claves cifrado.
heartbeat	15	Mantener “viva” la conexión TLS.
application_layer	16	Negociación de protocolos a nivel de aplicación.
signed_certificate	18	Relacionado con “*Certificate Transparency*” de Google.
client_certificate_type	19	Uso de claves públicas “en crudo”.
server_certificate_type	20	Uso de claves públicas “en crudo”.
encrypt_then_mac	22	Uso de cifrado previo a la autentificación.
extended_master_secret	23	Clave master_secret extendida.
session_ticket	35	Tickets para reanudar sesiones.
renegotiation_info	*	Protección ante ataques de renegociación.

Tabla 2.7 - Más extensiones de TLS 1.2.

Client Authz y *Server Authz*

Esta vez hacemos referencia a dos extensiones planteadas en la RFC 5878 [28] la cual posee status de experimental. Con cualquiera de las dos o con ambas, se agrega a TLS el servicio de autorización, que el protocolo no incluye originalmente.

Su funcionamiento es similar a lo comentado en el punto anterior. Se agregan por supuesto al CLIENTHELLO y/o a su contraparte del Servidor, y los datos adicionales viajan en un mensaje SUPPLEMENTALDATA.

Cert Type

La familia de protocolos SSL/TLS ofrece soporte únicamente para certificados del tipo X.509. Por lo tanto, en casos en donde es necesario utilizar otros estándares, por ejemplo, cuando no hay certificados X.509 disponibles, entonces toma relevancia esta extensión.

El detalle sobre su funcionamiento se expone en la RFC 6091 [29] la cual tiene status de “informativa”. Y en realidad, se trata únicamente de agregar el soporte para certificados OpenPGP, tal como ya lo indica su nombre.

Es por ello por lo que en el campo de datos de la extensión `cert_type` agregada al mensaje CLIENTHELLO la lista de alternativas se ciñe únicamente a los valores cero para X.509 y uno para OpenPGP. Si el Servidor no acepta este tipo de certificados, entonces hará uso del nuevo alerta `unsupported_certificate` que ya hemos mencionado.

El uso de certificados OpenPGP tendrá su efecto sobre los mensajes CERTIFICATE enviados a continuación, y una modificación menor al CERTIFICATEREQUEST en donde la lista de autoridades certificantes estará vacía, ya los certificados de este tipo son emitidos por entidades pares.

Elliptic Curves

La incorporación de la criptografía con curvas elípticas (ECC) al protocolo TLS se inicia con la RFC 4492 [30] cuyo status es “informativa” y propuso en su momento este agregado a las versiones 1.0 y 1.1.

En TLS 1.2, la extensión que permite operar con ECC se denomina `elliptic_curves` y sin duda es una de las de mayor importancia ya que permite operar con tamaños de claves menores que otros criptosistemas tales como RSA, para el mismo nivel de seguridad.

Esto desde ya es un punto a su favor considerando que las extensiones en general tienen como objetivo el alivianar la carga de cómputo en ambientes donde hay pocos recursos disponibles.

La RFC 4492 [30] sugería el implementar la versión de Diffie-Hellman que utiliza curvas elípticas, como el algoritmo de intercambio de claves, ya sea en su versión con parámetros fijos ECDH, o con modalidad efímera ECDHE, combinando cualquiera de ellos con el mecanismo de autenticación RSA, o con el conocido como ECDSA, que no es más que DSA también con curvas elípticas.

Se generan entonces cuatro posibles combinaciones, cuyas características se detallan en la tabla de la siguiente página. Un quinto algoritmo funciona en modo anónimo, sin autentificación.

Cada uno de los cinco algoritmos, puede ser combinado con un método de cifrado y *hash* para producir un conjunto de cifrado (*Cipher Suite*).

Sin embargo, solo los dos que utilizan ECDHE permiten obtener una característica de enorme importancia que es el “secreto perfecto hacia adelante” (*Perfect Forward Secrecy* – PFS), al cual haremos referencia en la sección sobre TLS 1.3.

Combinación	Detalle
ECDH_ECDSA	**ECDH fijo más certificados firmados con ECDSA.** • El certificado del Servidor debe incluir una clave pública fija ECDH firmada con ECDSA. • No hace falta enviar un mensaje SERVERKEYEXCHANGE. • El Cliente genera su par de claves a partir de la misma curva que utiliza el Servidor. • El Cliente puede enviar un mensaje CLIENTKEYEXCHANGE con su clave pública. • Se produce luego un intercambio de claves ECDH y se utiliza el resultado como el `premaster_secret`.
ECDHE_ECDSA	**ECDH efímero más firmas ECDSA.** • El certificado del Servidor debe incluir una clave pública ECDSA firmada con ECDSA. • El Servidor envía un mensaje SERVERKEYEXCHANGE con su clave pública efímera ECDH y los datos de la curva elíptica a utilizar. • Los parámetros enviados son firmados con ECDSA y la clave pública del certificado. • El Cliente genera su par de claves efímeras ECDH sobre la misma curva y envía un mensaje CLIENTKEYEXCHANGE. • Se produce luego un intercambio de claves ECDH y se utiliza el resultado como el `premaster_secret`.
ECDH_RSA	**ECDH fijo más certificados firmados con RSA.** • Idéntico al primer caso, pero esta vez el certificado del Servidor está firmado con RSA.
ECDHE_RSA	**ECDH efímero más firmas RSA.** • Idéntico al segundo caso, salvo que esta vez el certificado del Servidor debe incluir una clave pública RSA habilitada para la firma. • La firma en el mensaje SERVERKEYEXCHANGE debe ser generada con la clave privada RSA. • El certificado del Servidor debe ser firmado con RSA.
ECDH_anon	**ECDH anónimo sin autentificación.** • No se requieren firmas. • No se necesita certificados. • El intercambio de claves púbicas ECDH se realiza en los mensajes SERVERKEYEXCHANGE y CLIENTKEYEXCHANGE.

Tabla 2.8 - Algoritmos de intercambio de claves con curvas elípticas.

En realidad existen dos extensiones relacionadas con curvas elípticas, y ellas son `elliptic_curves` y `ec_point_formats`. El Cliente puede incluir una o las dos en su mensaje CLIENTHELLO.

Con la primera de ellas, el Cliente especifica cuáles son las curvas con las que puede y desea trabajar. La página de IANA referida a parámetros de TLS mantiene el listado de las curvas recomendadas y los documentos RFC que las describen. Esos documentos tales como la RFC 8422 [31] también hacen referencia a organizaciones que producen estándares tales como NIST y ANSI.

La segunda extensión consiste simplemente en una opción para comprimir los parámetros de las curvas, más teórica que práctica.

SRP

El protocolo SRP (*Secure Remote Password*) es una variante del método Diffie-Hellman de intercambio de claves. Por lo tanto, se trata de un algoritmo más que se puede combinar con cifrado más *hash* para producir conjuntos de cifrado adicionales.

Mencionamos la extensión solo para completar la lista enunciada anteriormente, y hacemos una breve descripción dado que la misma no ha sido generalmente adoptada.

La RFC 5054 [32] de carácter informativo propone utilizar al protocolo SRP como método de autentificación en TLS. Una página de la Universidad de Stanford también está dedicada al mismo [33].

El Cliente envía su mensaje CLIENTHELLO agregando la extensión `srp`, y si el Servidor la acepta entonces realizan un intercambio de mensajes SERVERKEYEXCHANGE y CLIENTKEYEXCHANGE con formato especial, y con ellos logran el cómputo de la `master_secret`.

Heartbeat

El protocolo TLS no ofrece mecanismo alguno para mantener la conexión “viva” entre dos pares. La extensión definida en la RFC 6520 [34] habilita tal funcionalidad.

En principio, se utiliza la extensión `heartbeat` para indicar que se desea activar el envío de mensajes con este propósito, y existen dos modalidades de trabajo para cada extremo, en donde se puede optar por enviar y recibir notificaciones o solamente enviarlas.

Es interesante señalar que la extensión introduce un nuevo tipo de mensaje al protocolo TLS que en realidad, es un subprotocolo ***Heartbeat*** adicional en la capa superior con su propio formato y código 24, tal como lo son ***Alert***, ***Handshake***, ***Change Cipher Spec*** y ***Application Data***.

Application Layer Protocol Negotiation

Tal como reza la RFC 7301 [35] que tiene carácter de estándar de Internet (*standards track*), esta extensión abreviada como ALPN permite que la capa de Aplicación negocie cuál protocolo será utilizado, desde dentro de la conexión TLS.

Su función toma sentido cuando se debe usar el mismo puerto TCP, siendo el ejemplo más común el 443, para ejecutar distintos protocolos, y originalmente fue solicitada con el fin de implementar HTTP2 sobre TLS.

Hoy la lista de protocolos habilitados se encuentra en la página de IANA dedicada a las extensiones, e incluye a nombres conocidos tales como FTP, IMAP y POP3.

El funcionamiento de la extensión es simple y consiste en enviarla como parte del mensaje CLIENTHELLO, para luego ser aceptada o no por el Servidor. Su nombre es `application_layer_protocol_negotiation`.

Signed Certificate Timestamp

Certificate Transparency (CT) es el nombre de un proyecto impulsado por Google desde el año 2013, en el cual también se publicó la RFC 6962 [36] de carácter experimental.

Dicho proyecto propone un mecanismo mediante el cual las autoridades certificantes (CA) pueden enviar una copia de los certificados que emiten a un registro público mantenido en línea con tal propósito, y se puede consultar a los mismos obteniendo un comprobante que se denomina "marca temporal firmada del certificado" (*Signed Certificate Timestamp – SCT)*.

El sistema es más complejo que lo enunciado en el párrafo previo, y comprende a lo que llaman un "ecosistema" en el que participan varios actores, tales como las CA's, los servidores de registros (*logs*), servidores que operan como monitores de esos registros y finalmente los navegadores clientes que auditan (*auditors*) los certificados.

Todo ello se encuentra bien detallado en el sitio web dedicado al proyecto [37]. Al fin de cuentas, la idea es que la SCT acompañe al certificado y se la pueda consultar de una de tres formas: o bien mediante una referencia en una extensión del propio certificado X509.3 que se denomina "*sct list*", una extensión de TLS con nombre `signed_certificate_timestamp` que es la que nos ocupa aquí, o mediante el ya conocido *OCSP Stapling*.

Certificate Type

Existen situaciones en las que no es necesario, o es demasiado costoso en términos de cómputo el procesar toda la información incluida en un certificado X.509, en uno del tipo OpenPGP, o incluso en uno "auto firmado" (*Self Signed*).

Para aquellos casos en los que solo hace falta utilizar las claves públicas "en crudo" (*Raw Public Keys*), se publicó la RFC 7250 [38] con carácter de estándar de Internet, que introduce dos nuevas extensiones, una de ellas denominada `client_certificate_type` y la otra con nombre `server_certificate_type`.

El documento mencionado describe en detalle el mecanismo mediante el cual se propone hacer uso de estas claves en crudo, en donde además de agregar una o las dos extensiones a los mensajes HELLO se modifica en cierta medida el proceso de negociación.

Cabe señalar que en la RFC 7250 [38] se presentan como claves utilizables las correspondientes a RSA, DSA, y ECDSA.

Encrypt then MAC

El procesamiento de los datos que realiza el subprotocolo ***Record*** utiliza el método "autenticar y luego cifrar" (*Authenticate then Encrypt – AtE*).

Ese método que parecía apropiado en la década de los '90, cuando se presentó el diseño original de SSL, ha encontrado más tarde numerosas vulnerabilidades, y hoy es preferido el orden inverso, *EtA*.

La extensión `encrypt_then_mac` permite cambiar ese orden y su implementación se describe en la RFC 7366 [39] que es un estándar de Internet.

La propuesta es sencilla e implica como de costumbre, el enviar una señal durante los mensajes HELLO. Una vez aceptado el cambio, el subprotocolo ***Record*** altera su modo de operación, en donde el MAC queda por fuera de la función de cifrado, como lo plantea la propia RFC:

```
encrypt( data || MAC || pad )
```

Se transforma en:

```
encrypt( data || pad ) || MAC
```

Lo que produce una serie de detalles a tomar en cuenta que en ese documento se tratan.

La extensión es de gran interés dado que introduce una modificación al modo de operación del subprotocolo ***Record***. Sin embargo, su utilidad se verá opacada por la implementación de los conjuntos de cifrado AEAD que vimos en el punto 2.2.3.

Extended Master Secret

La RFC 7627 [40] con status de estándar de Internet aborda un problema que está relacionado con el hecho de que la clave `master_secret` no se vincula criptográficamente con parámetros de sesión importantes tales como el certificado del Servidor, y ello hace posible que se pueda montar un ataque conocido como “apretón de manos triple” (*Triple Handshake Attack*).

Es por ello por lo que este documento propone modificar la construcción de esta clave que en TLS 1.2 se produce de esta manera:

```
master_secret = PRF(pre_master_secret, "master secret",
    ClientHello.random + ServerHello.random)[0..47];
```

Reemplazando tal fórmula por la siguiente a la cual llama “clave maestra extendida”:

```
master_secret = PRF(pre_master_secret,
"extended master secret", session_hash)[0..47];
```

Notemos que ha cambiado la etiqueta de texto y se han eliminado los valores `random` de ambos extremos. En su lugar, se introduce un nuevo componente denominado `session_hash`. Este cómputo debe incluir a todos los mensajes del *handshake* desde el inicio y hasta el CLIENTKEYEXCHANGE inclusive.

La funcionalidad se activa como ya es costumbre, enviando la extensión de nombre `extended_master_secret` en el mensaje CLIENTHELLO y respondiendo con la misma desde el SERVERHELLO.

Existen varias consideraciones relacionadas tanto con la negociación correspondiente a un *handshake* completo, a su versión abreviada y a la interoperabilidad con Clientes o Servidores que operan con versiones anteriores del protocolo.

En síntesis, lo que se consigue a través de la clave maestra extendida es que cada conexión de TLS genere una distinta.

Session Tickets

Para que los Clientes puedan reanudar sesiones evitando la carga de cómputo que pesa sobre ambas partes al realizar un apretón de manos completo, el Servidor debe mantener un registro del estado de cada una de ellas.

La extensión de nombre `session_ticket` descrita en la RFC 5077 [41] con categoría de estándar propone un mecanismo para completar la reanudación sin mantener el estado de las sesiones de cada Cliente.

Durante la primer negociación completa, el Cliente envía esta extensión vacía en el mensaje CLIENTHELLO y el Servidor responde de la misma forma en el SERVERHELLO.

Ese intercambio sirve solo para señalizar que ambas partes soportan y desean utilizar los tickets de sesión.

Los parámetros de la sesión al momento de los mensajes iniciales no están todavía disponibles. Más adelante en este handshake completo, el Servidor enviará un nuevo tipo de mensaje cuyo nombre es NEWSESSIONTICKET, lo cual modifica el diseño del subprotocolo. Esto se realiza justo antes del CHANGECIPHERSPEC y FINISHED.

La información que se envía en el ticket de sesión, comprende a todos sus parámetros en formato cifrado, y las claves para descifrarla las posee el Servidor. Entre esos datos se encuentra la `master_secret` y el conjunto de cifrado vigentes.

Cuando el Cliente desee reanudar una sesión ya negociada, enviará la extensión `session_ticket` con el contenido del ticket recibido anteriormente. Se realizará entonces un *handshake* abreviado, en donde o bien se utiliza directamente el *ticket*, o se puede renovar el mismo. En el segundo caso el Servidor enviará otra vez el mensaje NEWSESSIONTICKET pero esta vez a continuación del SERVERHELLO.

Es importante notar que el uso de los tickets de sesión rompe la propiedad de “secreto hacia adelante” (*PFS – Perfect Forward Secrecy*) que brindan ciertos algoritmos tales como DHE y ECDHE, en el caso en que un atacante pueda hacerse de las claves de cifrado con que el Servidor protege al ticket.

Renegotiation Info

Una vulnerabilidad que afecta al proceso de renegociación de sesiones tanto en TLS 1.2 como en sus versiones anteriores, y que se denomina "ataque de renegociación" (*Renegotiation Attack*) originó la propuesta documentada en la RFC 5746 [42] con status de estándar.

En ella, se describe la implementación de una extensión de nombre `renegotiation_info` cuyo código es curiosamente 65281 (0xFF01) lo que la sitúa al final del listado mantenido hoy por IANA.

Lo interesante de la extensión es que, con el fin de crear una vinculación criptográfica entre cada renegociación y su respectiva conexión, la misma genera tres nuevos parámetros a ser mantenidos en el estado de conexión, que se exhiben en la siguiente tabla:

Parámetro Nuevo	Detalle
secure_renegotiation	Una bandera (*flag*) que indica si la renegociación segura está activa en esta conexión.
client_verify_data	Un valor de 12 bytes en TLS que corresponde al cómputo `verify_data` del mensaje FINISHED del Cliente en el *handshake* inmediatamente anterior.
server_verfiy_data	Un valor de 12 bytes en TLS que corresponde al cómputo `verify_data` del mensaje FINISHED del Servidor en el *handshake* inmediatamente anterior.

Tabla 2.9 – Parámetros adicionales para la renegociación.

Adicionalmente, la RFC 5746 [42] también propone un nuevo conjunto de cifrado TLS_EMPTY_RENEGOTIATION_INFO_SCSV que no se utiliza como tal sino con el mismo propósito que la extensión en versiones anteriores de TLS (1.0 y 1.1) e incluso para SSL3 donde la misma no sería reconocida.

Cabe comentar que el ataque "*Triple Handshake*" mencionado anteriormente para la extensión `extended_master_secret` logra superar a la protección aquí enunciada.

2.7 Otros Cambios

En este punto revisaremos otras modificaciones adicionales realizadas al protocolo TLS a través de sus tres primeras versiones, que revisten interés a fin de completar nuestro análisis.

Abordaremos primero los cambios realizados al manejo de certificados, para luego ocuparnos de los mensajes de alerta, y finalmente algunas cuestiones restantes individuales.

2.7.1 Certificados

En TLS 1.0 se reduce la complejidad de la verificación de los certificados, permitiendo que la cadena de los mismos se revise hasta la primer autoridad CA de confianza. Esto supone una mejora con respecto a SSL 3.0 en donde se exigía que se controle toda la cadena hasta la raíz.

También se reduce la cantidad de tipos de certificados soportados, quedando solo 4 válidos, los primeros de la tabla 1.8. Los que se quitaron en esta versión, vuelven en TLS 1.1 pero con status de reservado, lo que es una recomendación para que no se los use.

Finalmente en TLS 1.2, con la incorporación de criptografía de curvas elípticas (*ECC*) se agregan tres tipos adicionales, que son primero el `ecdsa_sign` con una clave pública válida para la firma digital que utilice el mismo algoritmo de hash que el mensaje CERTIFICATEVERIFY.

En segundo y tercer lugar, los certificados del tipo `rsa_fixed_ecdh` que utiliza RSA para autentificación y `ecdsa_fixed_ecdh`, que usa por parte ECDSA. En todos los casos, los tres deben implementa la misma curva elíptica y formato de punto soportados por el Servidor, y firmados con un par apropiado de algoritmos de *hash* y generación de firma.

2.7.2 Mensajes de Alerta

TLS 1.0 ya introduce una lista de 23 mensajes de alerta que supera a la lista original propuesta por SSL. Dicha lista puede revisarse en la especificación, y uno de los mensajes denominado `no_certificate` pasa a estado reservado.

En TLS 1.1 se agregan dos mensajes más al estado reservado, uno de ellos es `export_restriction` a causa de la eliminación de los algoritmos exportables, y `decryption_failed` por cuestiones de seguridad.

La versión 1.2 agrega un nuevo mensaje denominado `unsupported_extension` obviamente relacionado con el mecanismo de extensiones que introduce, el cual es enviado por un Cliente que recibe un mensaje SERVERHELLO con una extensión que no especificó. También se adicionan 4 mensajes más en otra RFC, que pueden verse en la tabla 2.6.

2.7.3 Varios

En TLS 1.0 cambia el diseño de los mensajes CERTIFICATEVERFIY y FINISHED, haciéndolos más simples y en sintonía con la construcción TLS PRF.

En el primer caso, se aplica un firma digital sobre todos los mensajes previamente intercambiados concatenados, donde la clave de esa firma corresponde a la clave pública del certificado del Cliente.

En el segundo, se compone el mensaje de una forma muy similar al TLS PRF, donde se combinan la clave `master_secret`, los *hash* MD5 y SHA-1 de la concatenación y una etiqueta. Esto cambia en TLS 1.2 en donde se utiliza un solo tipo de *hash*, y el tamaño `verify_data` de la verificación pasa de ser fijo y de 12 bytes a variable y dependiente del conjunto de cifrado negociado.

A partir de TLS 1.1 se permite que las sesiones que no se cerraron adecuadamente con el intercambio de alertas `close_notify` se puedan reanudar dadas ciertas condiciones.

Finalmente, comentamos que la RFC 3749 [43] introduce un nuevo algoritmo de compresión para TLS, recordando que la recomendación general es no usar la compresión, por cuestiones de seguridad.

3. Protocolo TLS 1.3

El 10 de Agosto de 2018 se publicó la RFC 8446 [44], documento que presenta a la última y hoy vigente versión del protocolo, producto de diez años de investigación y avances desde la puesta en servicio de TLS 1.2.

Su concepción fue iniciada por una propuesta de Eric Rescorla en 2013 durante el encuentro 87 del IETF [45] la cual expone una “lista de deseos” para una nueva versión de TLS. En esa lista, se incluían los siguientes:

- Reducir la latencia en la negociación.
- Agregar más cifrado al *handshake*.
- Prevenir ataques de protocolos cruzados.
- Implementar únicamente algoritmos AEAD.
- Relacionar mejor los *Cipher Suites* con las versiones.
- Obsolescer a la versión SSL 2.0.
- Aumentar el tamaño de los *random*.
- Modificar la lista de conjuntos de cifrado.

Esta *wishlist* inició entonces un proceso que tomó cinco años por parte de la IETF y en donde los objetivos generales fueron el mejorar tanto la velocidad como la seguridad y la privacidad. A diferencia de lo ocurrido con TLS 1.2, en donde la principal preocupación era solo la seguridad.

Con respecto a la velocidad de adopción, cabe destacar que todavía en 2022, la versión mayormente desplegada de TLS sigue siendo la anterior, aún cuando ha sido reemplazada oficialmente por la 1.3.

Y en ese respecto, vale recordar que algo similar sucedió con la puesta en servicio real de TLS 1.2, en donde recién se produjo hacia 2013, habiendo sido publicada su respectiva RFC cinco años antes.

A diferencia de las iteraciones anteriores, TLS 1.3 representa un gran salto hacia adelante. Los cambios introducidos en esta nueva versión, que algunos creen debería haberse llamado TLS 2.0, no son menores y comprenden modificaciones tanto al diseño del protocolo como a sus conceptos fundamentales, pero sin distanciarse del la propuesta original.

Un ejemplo de ello es que en el nuevo estándar, ciertas extensiones son ahora obligatorias, dado que parte de la funcionalidad ha sido trasladada a ellas para mantener la compatibilidad con versiones anteriores.

En este capítulo vamos a utilizar un estilo de análisis distinto, no tan minucioso y más conceptual. Para desarrollarlo, primero veremos brevemente la lista de los principales cambios que se anuncian al inicio de la RFC 8446 [44], y luego estudiaremos las dos modificaciones al protocolo consideradas de mayor relevancia, relacionadas con la negociación inicial y su velocidad.

Finalmente, y luego de haber recorrido esos dos cambios fundamentales, estudiaremos los restantes, en el orden que plantea la lista más algunos adicionales.

Esta tercera parte del trabajo está basada en las presentaciones realizadas por la IETF [46], Alex Balducci de NCC Group [47] y los analistas Filippo Valsorda y Nick Sullivan de Cloudflare [48]. Las figuras que se exponen a continuación son adaptaciones de las que allí se utilizaron.

También nos apoyamos en lo ya visto en los dos capítulos anteriores, en las RFC obviamente, y es aquí donde realizamos comparaciones entre lo logrado hasta TLS 1.2 y la nueva propuesta, incluyendo referencias a las vulnerabilidades encontradas a través de los años y cómo han sido resueltas.

3.1 Principales Cambios

Como ya es costumbre en las sucesivas especificaciones de TLS, la RFC 8446 [44] incluye una sección 1.2 que enumera las “diferencias principales” respecto de la versión anterior.

Dichas diferencias se resumen en la siguiente tabla, la cual no es exhaustiva y está planteada a criterio propio:

Tema	Descripción Breve
Conjuntos de Cifrado	Se descartan todos los algoritmos simétricos antiguos.
	Solamente se permiten algoritmos AEAD.
	Cambia la forma de definir los *Cipher Suites*.
0-RTT	Se agrega un modo de negociación con este nombre.
Forward Secrecy	Se eliminan los intercambios estáticos de claves RSA y DH.
Cifrado del *Handshake*	Los mensajes se cifran a partir del SERVERHELLO y se agrega un nuevo tipo de mensaje ENCRYPTEDEXTENSIONS.
HKDF	Nuevo mecanismo mejorado para la derivación de claves.
Máquina de Estados	Se modifica su estructura y se elimina el *ChangeCipherSpec*.
Curvas Elípticas y Algoritmos de Firma	Las curvas elípticas son parte de la especificación base.
	Se agregan nuevos algoritmos de firma ECDSA.
	Se permite un solo formato de punto para cada curva.
Mejoras Criptográficas	Se implementa el RSASSA-PSS.
	Se elimina la compresión.
	Se elimina la firma DSA.
	Se eliminan los grupos DHE personalizados.
Negociación de Versión	El mecanismo de la versión anterior de TLS se reemplaza por una lista enviada en una extensión.
PSK	Se introduce un único intercambio de claves precompartidas.

Tabla 3.1 – Principales cambios en TLS 1.3.

Se aclara también que se han aggiornado referencias a las RFC actualizadas a través de todo el estándar, y se incluyen también cambios que afectan a las implementaciones actuales de TLS 1.2.

A continuación, centraremos primero nuestra atención en las dos mejoras más importantes introducidas con esta nueva versión.

3.2 Conjuntos de Cifrado

Se trata de uno de los cambios más significativos que introduce esta nueva versión y corresponde a la redefinición del concepto de *Cipher Suite*.

Si consideramos todas las versiones previas de TLS, se han acumulado más de 200 conjunto de cifrados disponibles. TLS 1.2 en particular define 37 más distintas adiciones. Y no todas son ya opciones seguras, a partir de las vulnerabilidades encontradas a lo largo del tiempo.

El nuevo estándar define solo cinco de ellos, purgando así de manera contundente los problemas de esa enorme lista anterior. Y además, flexibiliza la negociación de parámetros, como veremos a continuación.

Los *Cipher Suites* de TLS 1.3 establecen solo un par de algoritmos simétricos lo que incluye uno del tipo AEAD, más otro de *hash* que lo acompaña. Por lo tanto, su formato responde ahora al siguiente esquema:

```
CipherSuite TLS_AEAD_HASH = VALUE;
```

Donde `TLS` es el prefijo, `AEAD` es el algoritmo utilizado para la protección en el subprotocolo ***Record*** del que ya hablamos en el capítulo anterior, y `HASH` es el que sirve para la función HKDF de derivación de claves que describiremos más adelante. `VALUE` por supuesto es su código.

TLS 1.3 Cipher Suite:	Código:	Definido en:
`TLS_AES_128_GCM_SHA256`	`{0x13,0x01}`	RFC 5116 [15]
`TLS_AES_256_GCM_SHA384`	`{0x13,0x02}`	RFC 5116 [15]
`TLS_CHACHA20_POLY1305_SHA256`	`{0x13,0x03}`	RFC 8439 [49]
`TLS_AES_128_CCM_SHA256`	`{0x13,0x04}`	RFC 5116 [15]
`TLS_AES_128_CCM_8_SHA256`	`{0x13,0x05}`	RFC 6655 [50]

Tabla 3.2 – Los 5 conjuntos de cifrado de TLS 1.3.

Como se puede ver en la tabla, la especificación define solo cinco conjuntos de cifrado, donde cada algoritmo AEAD se describe en la RFC señalada, y los de *hash* provienen de la publicación NIST.FIPS.180-4 [51].

Se elimina de ellos entonces la selección del método de intercambio de claves, y el de firma digital para la autentificación durante el *Handshake*, que se negocian por separado usando extensiones en el CLIENTHELLO:

- Algoritmos AEAD + *Hash*: *Cipher Suite*.
- Método de Intercambio de Claves: *Key Exchange*.
- Algoritmo de Firma Digital: *Signature Algorithm*.

Los métodos de intercambio de claves disponibles han sido también reducidos, quedando tres opciones vigentes, donde la primera y la última ofrecen la propiedad de “secreto hacia adelante” (*Forward Secrecy)*:

- EC(DHE):
    - Diffie-Hellman Efímero sobre campos finitos o curvas elípticas.
- PSK:
    - Clave precompartida (*Pre Shared Key*), ya sea por otro medio o resultante de una negociación anterior.
- PSK combinado con EC(DHE).

Y por último, queda la selección del algoritmo de firma digital. En este caso, se debe tomar en cuenta que el mismo se relaciona con los certificados, y con la comprobación de todo el *Handshake*, lo que veremos en detalle más adelante.

3.3 *Performance*

Aquí analizaremos en detalle la mejora más importante que introduce TLS 1.3, que consiste en reducir la latencia del *Handshake*, uno de los primeros “deseos” en la lista que mencionamos al inicio del capítulo.

Esta mejora es fundamental para la *performance*, sobre todo para el caso de las conexiones distantes geográficamente y para las redes móviles, en donde el proceso de negociación inicial que se llevaba a cabo en las versiones anteriores suponía un alto costo en tiempo de ejecución.

Pero también es relevante para las conexiones internas en las organizaciones, en donde una mayor velocidad de negociación permite considerar como más viable el proteger el tráfico entre dispositivos de red, servidores, y elementos de “Internet de las Cosas” (*IoT* – *Internet of Things*).

3.3.1 Modo 1-RTT

Comenzamos el análisis recordando cómo funcionaba el *Handshake* inicial en la versión anterior TLS 1.2. Para poder compararla luego con la nueva, utilizaremos un intercambio de claves Diffie-Hellman Efímero (DHE).

A diferencia de lo que hicimos al principio de este trabajo, vamos a hacer uso de un sistema de figuras más conceptual y menos detallista. En ellas, se mostrará por la izquierda el *Key Schedule*, es decir, el mecanismo de derivación de claves que gobierna todo el proceso.

En la figura siguiente puede notarse que el *handshake* se inicia con el envío por parte del Cliente de su mensaje CLIENTHELLO, el cual incluye los conjuntos de cifrados que soporta y propone.

El Servidor responde entonces con su mensaje SERVERHELLO, seleccionando un conjunto de los propuestos que incluyan un intercambio de claves de Diffie-Hellman.

Por lo tanto, esa respuesta va a sumar su certificado, una firma digital sobre su clave pública de Diffie-Hellman y la clave pública efímera del Servidor que llamaremos DHs.

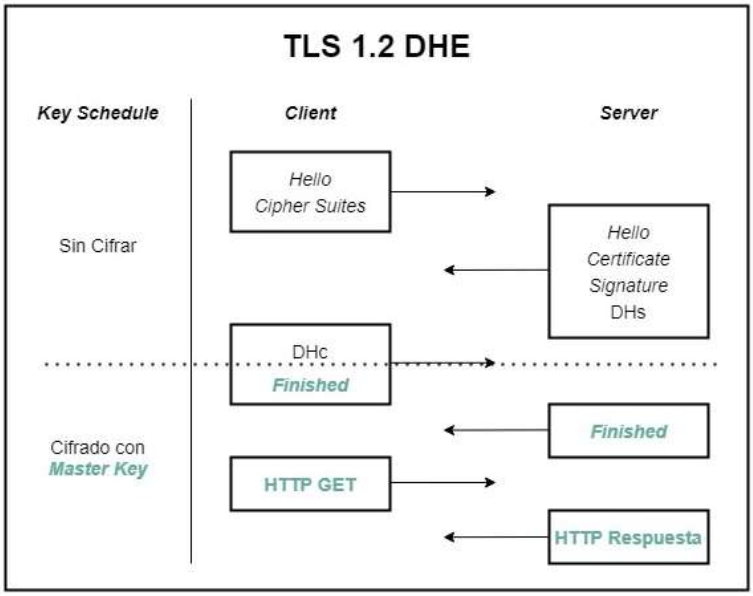

Figura 3.1 - El *Handshake* de TLS 1.2 con DHE.

Al recibir la respuesta del Servidor, el Cliente verificará el certificado contra la autoridad correspondiente (CA), comprobará la firma digital y estará en condiciones de enviar su propia clave pública efímera DHc.

En este punto, el Cliente ya cuenta con toda la información que necesita para cifrar las comunicaciones utilizando la clave *Master Key* simétrica, y el Servidor también. Por lo tanto, se realiza el intercambio de los dos mensajes FINISHED que ya viajan protegidos con esa clave.

Nótese que el proceso requirió de dos viajes de ida y vuelta entre el Cliente y el Servidor, lo que se conoce como 2-RTT (*Round Trip Time*). Y además, que la primer mitad sucedió sin cifrado, mientras que la segunda ya hace uso de la *Master Key*. Finalizado el 2-RTT, se procede a intercambiar datos, también cifrados con esa misma clave simétrica.

Ahora veremos entonces cómo hace TLS 1.3 para reducir el tiempo de negociación a tan sólo una ida y vuelta. Esto se denomina 1-RTT.

Como se ve en la próxima figura, el Cliente envía su mensaje CLIENTHELLO tal como antes, pero incluyendo una o más claves públicas de Diffie-Hellman que el Servidor puede llegar a soportar. En cierto sentido, el Cliente está adivinando de antemano los parámetros del intercambio.

Si sucede que el Servidor está de acuerdo con esos parámetros, lo que generalmente va a ocurrir, entonces ya en el mensaje SERVERHELLO cuenta con toda la información que necesita para comenzar el cifrado simétrico.

Responderá entonces con su clave pública, el certificado y la firma, y ya podrá enviar su mensaje FINISHED cifrado. Acto seguido, el Cliente enviará su propio mensaje FINISHED y ya se puede intercambiar datos. Con estos cambios en el proceso, se logró eliminar un viaje de ida y vuelta (*Round Trip*), lo que permite completarlo en 1-RTT.

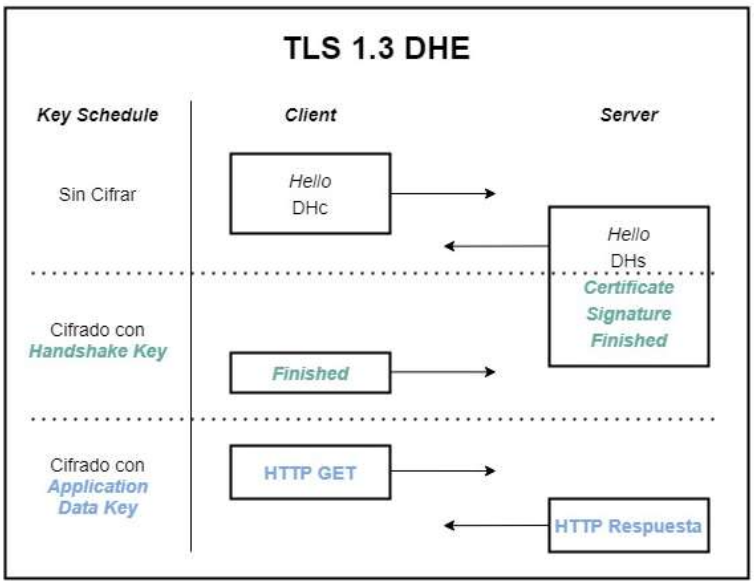

Figura 3.2 – El *Handshake* de TLS 1.3 con DHE.

Es importante señalar, que el *Key Schedule* ahora contempla tres etapas. Se genera una clave que sirve solo para el apretón de manos, que llamaremos por el momento *Handshake Key,* y otra que sirve para el envío de datos, a la que nos referiremos también en forma provisoria como *Application Data Key*. Es decir, ahora el *Handshake* se desarrolla en tres fases.

En adelante entonces, la primera negociación que realice un Cliente con un Servidor se hará en modalidad 1-RTT. Sin embargo, TLS 1.3 ofrece todavía más opciones. Antes de analizarlas, es preciso exponer cómo funciona el *Key Schedule*.

No debemos olvidar que este modo 1-RTT depende de que el Servidor acepte los parámetros propuestos por adelantado por el Cliente. Si por algún motivo no los acepta, entonces el Servidor deberá enviar un mensaje HELLORETRYREQUEST el cual incluye una *cookie* para descargar el estado al Cliente, quien responderá con un nuevo CLIENTHELLO que contenga esa *cookie* y parámetros adecuados.

En este último caso, el *handshake* pasa a requerir 2-RTT y es equivalente en su desempeño al de TLS 1.2 completo.

3.3.2 *Key Schedule*

El mecanismo de generación y derivación de claves simétricas para el cifrado de los datos ha cambiado considerablemente. Ya vimos en el punto anterior que en la modalidad 1-RTT se utilizan al menos dos de ellas, una para una parte del subprotocolo ***Handshake*****,** y otra para el ***Application Data***.

Este proceso conlleva cierta complejidad, aún cuando se trabajó durante el desarrollo del estándar para simplificarlo. Como ya dijimos, aquí se expone un esquema conceptual de su funcionamiento, evitando entrar en todos sus detalles los cuales están completos en la sección 7.1 de la RFC.

Todos los nombres que hemos asignado a las distintas claves son diferentes a los verdaderos, para mayor claridad.

La figura exhibe el esquema mencionado. En él puede notarse que en realidad, el *Key Schedule* incluye la generación de cuatro juegos de claves que se utilizarán a lo largo de todas las posibles versiones del *Handshake*, según el caso.

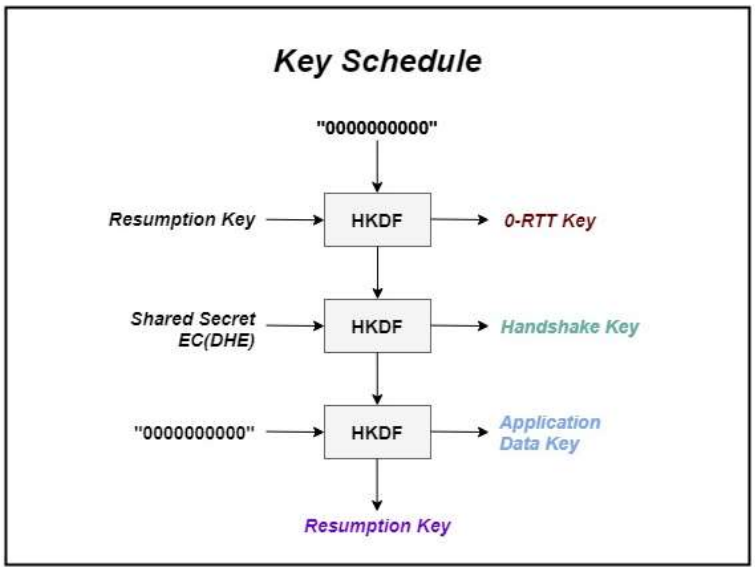

Figura 3.3 - El *Key Schedule* de TLS 1.3.

El proceso comienza al iniciar cualquier tipo de apretón de manos, ya sea el completo donde el Cliente se conecta con el Servidor por primera vez, o si se trata de una reanudación (*resumption*), donde el Cliente ya se había conectado anteriormente.

En el primer caso, la primera conexión, se arranca a partir de una cadena de caracteres nulos, que se ven en la figura como “0000000000”. Si se trata de una reanudación, se utiliza la clave que por ahora llamaremos *Resumption Key*, negociada en la conexión anterior.

Esa clave inicial con caracteres nulos o bien la *Resumption Key* alimentan a la función de derivación de claves HKDF (*Hash Key Derivation Function*) que produce como primer salida, otra clave que por el momento identificaremos como *O-RTT Key*, y sobre la que hablaremos luego.

Una vez finalizado el intercambio de claves de Diffie-Hellman Efímero en el *handshake*, ambas partes calculan la misma clave simétrica que designaremos como *Shared Secret*.

Esa clave más la salida de la función HKDF se combinan para generar la clave que realmente se utiliza para el cifrado de la segunda parte del *Handshake* y que ya mencionamos como *Handshake Key*.

Con ella se protegen el certificado del servidor, la firma digital, y ambos mensajes FINISHED.

Terminada la fase del *handshake* se comienza con la transmisión de datos, y para ello se alimenta una vez más a la función HKDF con otra cadena de caracteres nulos.

La HKDF produce entonces dos salidas, que son la clave para transmitir datos que hemos llamado *Application Data Key*, y una nueva *Resumption Key*, para una próxima reanudación.

Nótese que durante todo el proceso, las sucesivas aplicaciones de la función HKDF se van alimentando del resultado anterior, por arriba, y de alguna otra clave o la cadena nula por la izquierda.

3.3.3 Reanudación

El objetivo de esta modalidad de negociación es el de aprovechar la información generada en la conexión ya establecida anteriormente entre el Cliente y el Servidor para acelerar el proceso.

Para lograrlo, hace falta utilizar una clave preexistente, a la que llamamos *Resumption Key* y que es producto del *Key Schedule*. El problema es, cómo hacemos para almacenarla hasta su utilización?

Una opción como siempre es que el Servidor mantenga un registro, pero para implementaciones de gran porte con muchas conexiones, esto significa un gran consumo de recursos. La otra opción es descargar esta información en el equipo del Cliente.

Para habilitar la reanudación entonces, primero se establece una clave simétrica para el Servidor a la que llamaremos *Server Key*. Esta clave no surge del *Key Schedule*, es simplemente un parámetro que le asignamos al Servidor y que será desconocido para el Cliente.

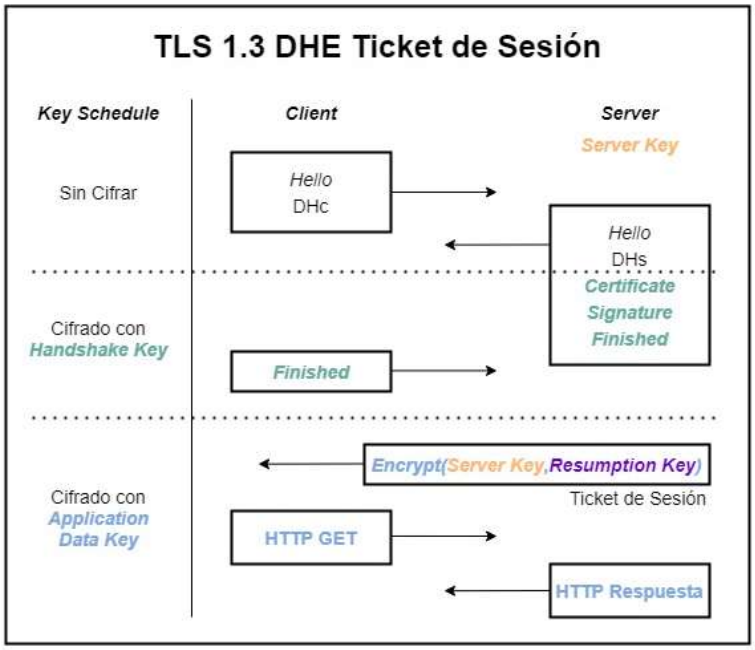

Figura 3.4 - Ticket de sesión en TLS 1.3.

Al finalizar el *Handshake* completo, el Servidor enviará al cliente la clave *Resumption Key* cifrada con la *Server Key*, tal como muestra la siguiente expresión conceptual:

```
Session Ticket = Encrypt(Server Key, Resumption Key)
```

De la cual se puede deducir que lo que envía el Servidor, es efectivamente un ticket de sesión, similar al que se utilizaba en TLS 1.2, pero con ciertas diferencias. Esta nueva forma de operar reemplaza a lo que se hacía en la versión anterior con los ID de sesión y los tickets.

El Cliente no tiene forma de saber qué información contiene ese ticket, solo lo almacena por el momento.

En la siguiente conexión, se produce la reanudación. En ella, el Cliente enviará el *Session Ticket* como parte del mensaje CLIENTHELLO. El Servidor descifra ese ticket con su *Server Key*, y obtiene la clave *Resumption Key*, a partir de la cual puede derivar claves nuevas para cifrar su mensaje FINISHED.

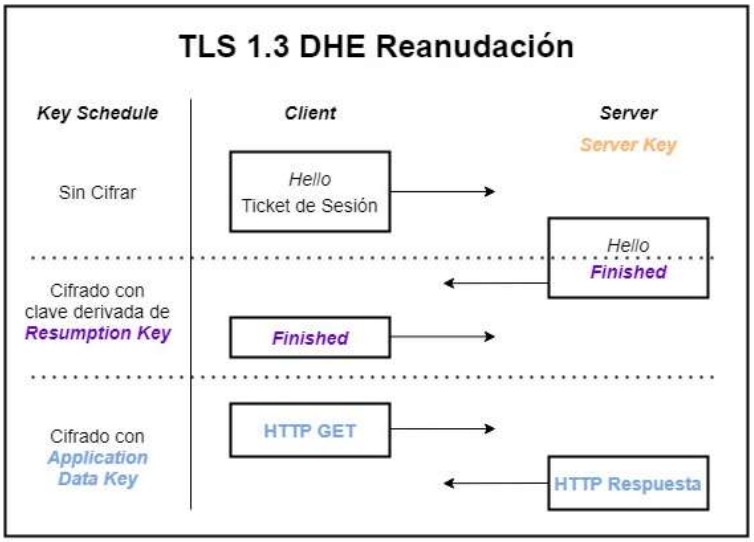

Figura 3.5 – Reanudación en TLS 1.3.

Como se puede observar en la figura anterior, el Servidor ya no necesita enviar su certificado, ni la firma, información que es muy voluminosa. De hecho, es la de mayor tamaño en todo el *handshake*.

Un problema de la reanudación tal como la estamos describiendo, es que su *Key Schedule* no contempla la creación de una *Shared Secret*, como en la conexión original. Por lo tanto, todas las claves derivadas dependerían únicamente de la *Resumption Key,* la cual a su vez había sido enviada en forma de ticket cifrado con la *Server Key*. De ser así, el esquema no sería muy diferente de un intercambio de claves estáticas RSA.

Así funciona la reanudación en TLS 1.2, en donde opera mediante tickets de sesión, en conjunto con el de ID de sesión. En este caso, quien puede hacerse de la clave con que se cifró el ticket, ya sea en el momento o en el futuro analizando de manera pasiva una grabación del intercambio, tiene la posibilidad de descifrar todos los datos, tanto de la sesión reanudada, como de la sesión original y las posteriores.

Esto se resuelve como se ve en la figura, agregando al CLIENTHELLO de la reanudación, un nuevo juego de claves de Diffie-Hellman. De esta manera, el cifrado del mensaje Finished depende ahora tanto de la clave *Resumption Key* como del nuevo *Shared Secret*.

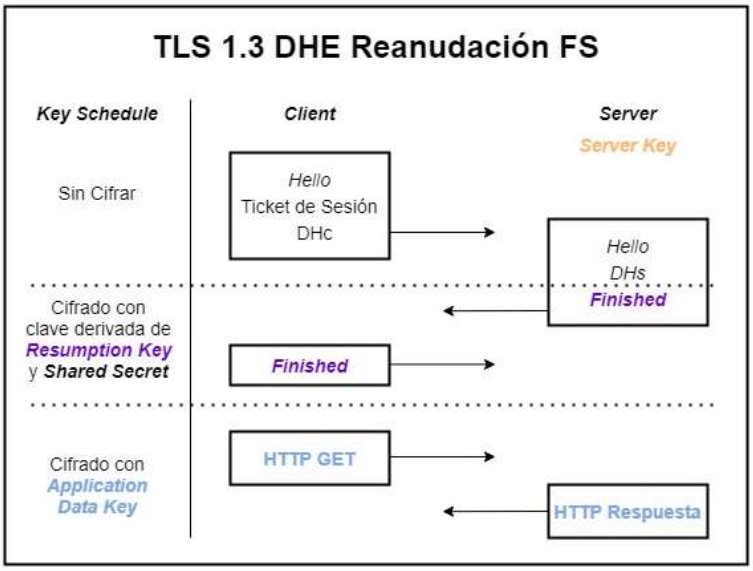

Figura 3.6 - *Forward Secrecy* en la reanudación de TLS 1.3.

Y es así como se logra que las reanudaciones tengan también la propiedad de “secreto hacia adelante” *(Forward Secrecy)*.

Dos recomendaciones surgen entonces para quien deba implementar un servicio con TLS 1.3. La primera, siempre negociar nuevas claves efímeras. La segunda, renovar la clave simétrica del o de los servidores, de manera frecuente, aunque ello implique el invalidar a los tickets de sesión anteriores.

3.3.4 Modo 0-RTT

Cómo se podría mejorar aún más el tiempo de negociación? La solución que ofrece TLS 1.3 es esta modalidad adicional, cuyo nombre sugiere el hacerla en “cero idas y vueltas”.

La realidad es un tanto diferente. Esta opción permite simplemente comenzar a enviar datos en el mensaje CLIENTHELLO. Para que esto ocurra, debe hacerse durante una reanudación, y esos datos tempranos *(Early Data)* estarán protegidos por una nueva clave derivada de la *Resumption Key* que ya mencionamos como parte del *Key Schedule*, la *0-RTT Key*.

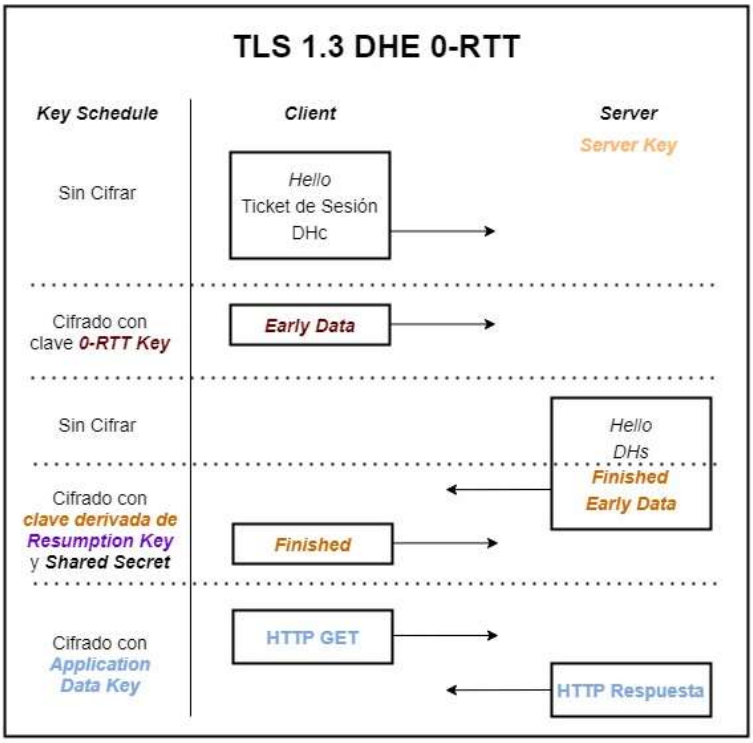

Figura 3.7 - 0-RTT en TLS 1.3.

Una vez más, se presenta el inconveniente que habíamos señalado para la reanudación, la falta de *Forward Secrecy* dado que en este caso, todavía no se han negociado las nuevas claves efímeras. Esta vez, el problema no tiene solución.

Por lo tanto, vale tener bien presente que el modo 0-RTT no cuenta con esa propiedad, que sí tienen el modo 1-RTT y la reanudación normal.

La falta de *Forward Secrecy* en esta modalidad genera toda una discusión sobre su conveniencia y su seguridad. Qué tipo de datos tempranos se pueden enviar sin problemas en el CLIENTHELLO?

Por otra parte, se presenta un segundo problema que consiste en la posibilidad de enviar varias veces el mismo mensaje CLIENTHELLO con idénticos datos tempranos, lo que se conoce como repetición (*Replay*). Y esto se agudiza aún más, si en el otro extremo existen varios Servidores, como suele ocurrir en instalaciones de importancia.

Las medidas de protección que se pueden implementar, incluyen el agregar un tiempo máximo de vida útil a los tickets de sesión, lo que a veces no funciona del todo bien. Un atacante podría enviar el mismo mensaje todas las veces posibles hasta que tiempo expire.

También se propone agregar una señal de *no replay* al mensaje FINISHED del Cliente.

Otra solución es almacenar los datos tempranos en un *buffer*, estableciendo también un tamaño máximo para ellos, y procesarlos realmente al terminar el *handshake*. Con esta opción, los datos ya no son "tan tempranos", pero se mantiene algo del beneficio.

Lo que sí es recomendable, es que la solicitud que se envíe en forma temprana, posea la propiedad de idempotencia. Es decir, en palabras más sencillas, que no haga cambios en el servidor. El ejemplo más claro, en el contexto de la navegación vía HTTP, es un comando GET que solamente solicite descargar una página, lo que no supone ningún riesgo si se reitera.

La sección número 8 y adicionalmente el Apéndice E5 del estándar incluyen recomendaciones relacionadas con los ataques de *replay* y el modo 0-RTT. Y una última es que no se implemente este modo sin un perfil que defina claramente su uso.

3.4 Reseña Final

En el punto 3.1 presentamos una tabla que resume los cambios más significativos que introduce TLS 1.3, basada en la lista que expone la propia RFC 8446 [44] al inicio.

Ya hemos analizado las dos cuestiones de mayor relevancia en esa lista, que son la redefinición del concepto de *Cipher Suites* y las mejoras en la *performance*, describiéndolas en general. En esta sección, haremos una reseña que detalla las modificaciones resultantes de esos dos cambios fundamentales, más otras que no están directamente relacionadas.

3.4.1 Conjuntos de Cifrado

La decisión de reducir notablemente la lista de *Cipher Suites* disponibles a solo cinco de ellos, donde todos combinan el cifrado exclusivamente AEAD con uno de *hashing* ya sea SHA-256 o SHA-384, implica una fenomenal limpieza de algoritmos simétricos débiles.

Esa poda abarca a cifradores de bloque y de flujo, más los algoritmos de *hashing* antiguos. Entre ellos se encuentran los siguientes, con las vulnerabilidades relacionadas a cada uno que han sido resueltas entre paréntesis:

- RC4 (RC4NoMore 2015 [52]).
- 3DES (Sweet32 2016 [53]).
- MD5 y SHA-1 (Sloth 2016 [54]).
- AES en modo CBC (Vaudenay 2002 [55], Boneh / Brumley 2003 [56], Beast 2011 [57], Lucky13 2013 [58], Poodle 2014 [59], Lucky Microseconds 2015 [60]).

Y podemos agregar un problema encontrado en AES modo GCM, relacionado con la mala configuración de sus *nonces* [61]. En TLS 1.3, estos parámetros deben ser explícitamente definidos, siendo su valor función de los números de secuencia de los mensajes intercambiados. Para mayor detalle, ver la sección 5.3 de la RFC.

3.4.1 0-RTT

Ya hemos dedicado un punto completo en este capítulo a las mejoras sobre la *performance*. La lista de cambios que presenta la RFC 8446 [44] a su inicio menciona este nuevo modo de negociación, pero curiosamente no hace referencia en esa lista al cambio más importante, que es la reducción de la negociación inicial completa a 1-RTT.

En referencia a este nuevo modo, en donde se puede enviar datos tempranos directamente en el mensaje CLIENTHELLO, caben recordar las recomendaciones sobre su seguridad disminuida, la falta de *Forward Secrecy* y los problemas de *replays*.

3.4.2 *Forward Secrecy*

Se ha visto en el punto anterior, que la propiedad de “secreto hacia adelante” es un requerimiento de diseño de TLS 1.3.

Es por ello por lo que esta versión del protocolo ha eliminado completamente a los modos RSA y Diffie-Hellman estáticos para el intercambio de claves.

Si bien comentamos al inicio de este capítulo que TLS 1.3 mejora sustancialmente los tiempos de negociación y hace más viable por ese motivo su implementación en las instalaciones internas de las organizaciones, esta eliminación de los modos RSA y DH estáticos presenta un serio problema.

Ese contratiempo está relacionado con la inspección del tráfico, tanto en línea como en modo pasivo (*Out of Band*). La propiedad de secreto hacia adelante o bien impide el seguirlo realizando de la manera en que se venía haciendo, o lo dificulta en forma considerable.

Paradójicamente entonces, es quizás una de las razones por las que instituciones tales como bancos y entidades financieras están tardando en adoptar el nuevo estándar.

Como ejemplo de esa cuestión, podemos mencionar que casi al final de la tarea de elaboración de la RFC 8446 [44], autoridades de las principales entidades financieras de los Estados Unidos reclamaron que el quitar el modo RSA estático les complicaría enormemente las tareas críticas de inspección de tráfico entre sus equipos internos.

En respuesta a ese reclamo, sin restituir RSA al nuevo protocolo pero como solución de compromiso no recomendada, se publicó un documento de carácter informativo únicamente, describiendo el uso de Diffie-Hellman estático sobre TLS 1.3 [62].

3.4.3 Cifrado del *Handshake*

El nuevo estándar incorpora el cifrado de los mensajes del *handshake* mucho antes que en las versiones anteriores. La propia negociación ha sido subdividida en tres fases, que son el intercambio de claves, los parámetros del servidor y la autentificación. Y hemos visto que el mecanismo de generación de claves también ha sido rediseñado, de modo tal que cada una de estas fases utiliza juegos diferentes de ellas.

Todo lo que ocurre en la negociación después del mensaje SERVERHELLO está cifrado con las claves correspondientes a cada etapa. Adicionalmente, se introduce en esta versión de TLS un nuevo tipo de mensaje denominado ENCRYPTEDEXTENSIONS, que envía el Servidor en la segunda fase.

Su función es la de enviar en forma cifrada, algunas extensiones que antes se transmitían sin protección. En la sección 4.3.1 del estándar, se las define y se hace referencia a la tabla que las enumera a todas.

3.4.4 HDKF

En el punto dedicado a la *performance* hemos visto que TLS 1.3 introduce un nuevo mecanismo para la derivación de claves, y que se utilizan varios tipos de ellas para las distintas fases del *Handshake*.

Recordemos también que en el desarrollo de ese punto, hicimos uso de nombres conceptuales para las claves derivadas y no de sus nombres reales, que pueden ser revisados en la sección 7.1 del estándar.

Ese proceso implementa la función *HMAC-based Extract-and-Expand Key Derivation Function* (HKDF) como primitiva, la cual se describe en la RFC 5869 [63]. A partir de ella, construye otras funciones propias del estándar.

El algoritmo de *hashing* que se utiliza en este esquema es el que se negocia como parte del conjunto de cifrado, es decir, SHA-256 o SHA-384. Y cada vez que se necesita derivar una clave en base a un *hash*, se aplica la HKDF de manera consistente.

3.4.5 Máquina de Estados

Desde el diseño original del protocolo SSL 3.0, y hasta TLS 1.2 con ciertos cambios, tal como vimos en los respectivos capítulos, la coordinación de los estados durante el *handshake* tanto del Cliente como del Servidor, actuales y pendientes de lectura y de escritura, se realizaba manteniendo un registro de parámetros tanto para las sesiones como para las conexiones.

En ese contexto, el subprotocolo ***Change Cipher Spec*** funcionaba como un disparador para el cambio de parámetros, activando los pendientes y haciéndolos actuales.

En TLS 1.3, se ha realizado una reestructuración de la máquina de estados, con el fin de que sea más consistente, y eliminar los mensajes CHANGECIPHERSPEC a los que considera superfluos. Ello significa una modificación sustantiva al diseño del protocolo, que pasa a tener solo tres componentes en su capa superior. Cabe notar que estos mensajes se pueden seguir utilizando con fines de compatibilidad.

Por primera vez en los documentos que definen a las distintas versiones de TLS, se incluye un Apéndice A que expone una verdadera máquina de estados con su correspondiente diagrama y nomenclatura tanto para el Cliente como para el Servidor.

3.4.6 Curvas Elípticas y Algoritmos de Firma

La lista de cambios proporcionada por el propio estándar y a partir de la cual estamos describiendo cada uno en ese orden, expone que en esta versión las curvas elípticas (EC) son parte de la "especificación de base" (*base spec*), es decir, que se han incorporado directamente al protocolo.

Si recordamos nuestro paso por TLS 1.2, en esa versión las curvas elípticas habían sido introducidas vía la RFC 4492 [30]. Hoy esa RFC ha sido sustituida por la 8422 [64].

También debemos tener presente que las curvas se utilizan para dos fines en TLS. Uno es el intercambio de claves para la negociación inicial, y otro es la firma digital para la autentificación de las partes.

En TLS 1.3, el Cliente envía la extensión `supported_groups` para acordar la curva elíptica o el campo finito que se utilizará para el intercambio de claves. Es por ello por lo que su nombre es más genérico.

Un dato interesante es que en esta versión, el Servidor también puede responder con una extensión `supported_groups`. Sin embargo, esa respuesta, tiene como objetivo que el Cliente pueda prepararse mejor para el próximo *Handshake*, enviando en ese momento una extensión `key_share` con grupos más adecuados a las preferencias del Servidor.

De hecho, el Cliente puede enviar un vector de grupos vacío, para que el Servidor le informe todos los grupos que soporta. Esto tiene un costo de un viaje adicional.

Lo mismo ocurre al generarse un mensaje HELLORETRYREQUEST, por parte del Servidor, si el Cliente envió grupos con parámetros no soportados.

Las curvas elípticas que se pueden utilizar para el intercambio de claves están definidas en el estándar FIPS 186-4 [65] y en la RFC 7748 [66]:

```
secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019)
            x25519(0x001D), x448(0x001E)
```

Como se puede ver, la lista tiene pocas opciones, lo que en criptografía significa mejores opciones. Además, solo se acepta un formato de punto para cada curva. Anteriormente, como describía la RFC 4492 [30] en su sección 5.1.2, se podía elegir entre varios formatos.

Más detalles técnicos sobre los parámetros de las curvas en la negociación de claves pueden encontrarse en la sección 4.2.8.2 del estándar. A su vez, la sección 7.4.2 detalla la forma en que se debe calcular la clave compartida que llamamos *Shared Secret* en el capítulo sobre *performance*, haciendo referencia a recomendaciones de la RFC 7748 [66] relacionadas con las dos últimas curvas.

El uso de curvas elípticas para la autentificación, está relacionado con los algoritmos de firma digital. Ya en el inicio del estándar se indica que ella es obligatoria para el Servidor, y opcional para el Cliente.

Se puede utilizar criptografía asimétrica con curvas elípticas ECDSA (*Elliptic Curve Digital Signature Algorithm*) [67], y la versión especial de ellas EdDSA (*Edwards-Curve Digital Signature Algorithm*) [68] que es una novedad en TLS 1.3.

Curiosamente, si bien estamos hablando ahora de autentificación y no de intercambio de claves, la selección del algoritmo de firma se realiza ya desde el mensaje CLIENTHELLO.

En ese mensaje inicial, el Cliente utiliza una extensión denominada `signature_algorithms` y opcionalmente otra de nombre similar:

`signature_algorithms_cert`

La primera sirve para establecer el algoritmo de firma en el mensaje CERTIFICATEVERIFY mientras que la segunda es exclusiva para las firmas en certificados. Si la segunda no se utiliza entonces la primera cumple con ambas funciones.

Recordemos que la selección del algoritmo de firma, se negocia en forma independiente del *Cipher Suite*. Y también que existen certificados que incluyen una clave para usar con cierto algoritmo de firma, que están firmados con otro. Por ejemplo, uno con clave RSA firmado con ECDSA.

En TLS 1.3 se pueden utilizar dos algoritmos de firma digital que incluyen curvas elípticas. El primero es ECDSA, y el segundo que es nuevo EdDSA definido en la RFC 8032 [68].

El mensaje CERTIFICATEVERIFY tiene dos propósitos. Probar que el que lo envía – en general el Servidor – es el dueño de la clave privada de su certificado, y también comprobar la integridad de todo el *handshake* hasta ese punto. Por lo tanto, utiliza el algoritmo de firma digital negociado a tales efectos. Esto es diferente de la firma que comprueba el certificado, y es por ello por lo que pueden llegar a usarse dos extensiones.

3.4.7 Mejoras Criptográficas

Revisaremos aquí cuatro de las mejoras en criptografía más relevantes anunciadas al inicio de la RFC 8446 [44]. Desde ya se aclara que hay otras adicionales que se describen en el resto del documento.

La primera de ellas se expone como la implementación del *padding* sobre RSA utilizando el esquema RSASSA-PSS (*RSA Probabilistic Signature Scheme*) en las firmas digitales.

Lo que realmente significa este cambio, es que se actualiza el uso del estándar PKCS #1 publicado por la empresa *RSA Laboratories* a la versión 2.2, descrita en la RFC 8017 [69]. La versión anterior de este estándar, la 1.5, incluía el esquema RSASSA-PKCS1-v1_5, relacionado con las siguientes vulnerabilidades:

- Bleichenbacher 1998 [70].
- Jager 2015 [71].
- Drown 2016 [72].

Lo que se ha conseguido es solucionar un problema que existe referido a cómo se implementaba el *padding* en las firmas RSA con el estándar anterior. Con el nuevo estándar, se soluciona haciendo uso del esquema PSS (*Probabilistic Signature Scheme*).

El esquema antiguo de la versión 1.5, se sigue utilizando pero solo para certificados digitales ya existentes, por cuestiones de compatibilidad. RSASSA-PSS es obligatorio en TLS 1.3, para los mensajes CERTIFICATEVERIFY.

La segunda mejora corresponde a la eliminación de la compresión, relacionada con la vulnerabilidad conocida como CRIME 2012 [73].y en cierta medida de forma indirecta con TIME [74] y BREACH [75].

No hay mucho más para comentar sobre la compresión, salvo indicar que ha sido eliminada por obvias razones de seguridad, y que el campo correspondiente en el CLIENTHELLO se incluye solo por razones de compatibilidad con versiones anteriores de TLS y en estas, se transmite como un *byte* de valor cero.

También sirve como una marca a partir de la cual comienzan las extensiones, cuya existencia se detecta verificando si hay *bytes* de información a continuación del campo `compression_methods`.

En tercer lugar, se encuentra la eliminación de las firmas digitales DSA. Si bien las mismas no están relacionadas con alguna vulnerabilidad específica, existen cuestiones originadas por el tamaño de sus claves, su *performance*, la preferencia por los algoritmos con curvas elípticas (ECDSA, EdCSA), y ciertas advertencias sobre implementaciones tales como la CVE-2008-0166 para generadores de números aleatorios, y la CVE-2018-0734 para un ataque de canal lateral.

Finalmente, la cuarta mejora implica la eliminación de grupos DHE personalizados para campos finitos. Lo que se ha decidido hacer, es utilizar grupos bien definidos y probados que ya habían sido establecidos en la RFC 7919 [76] incluso para versiones previas de TLS.

Con respecto a este cambio, se han generado interesantes discusiones en el sitio *StackExchange* [77] [78] sobre el dejar de usar estos grupos personalizados, y la vulnerabilidad LOGJAM [79], a partir de la cual se recomendaba hacer exactamente lo contrario. Una lista fija de grupos es mucho más valiosa para un atacante, que sabe que al quebrar uno de ellos puede tener acceso a millones de implementaciones.

Como en TLS 1.3 quien propone los grupos esta vez es el Cliente, no es completamente seguro el dejarle asignado el trabajo de comprobar la consistencia de un grupo personalizado. En las versiones anteriores, este trabajo lo hacía el servidor. Entonces, si bien se utiliza una lista fija de grupos, dado que son probados y seguros, conviene más seguir las indicaciones de la RFC 7919 [76].

Negociación de Versión

Dado que TLS 1.3 supone cambios tan significativos al protocolo, no todos los interesados en él están de acuerdo con su nombre. Muchos sostienen que debería llamarse TLS 2.0, entre otras propuestas. De hecho, hoy en día, todavía subsiste el nombre SSL y se sigue utilizando para fines comerciales.

Existen cuestiones técnicas y de compatibilidad que se han tomado en cuenta. En el mensaje CLIENTHELLO, el campo `ProtocolVersion` queda con el valor fijo "3.3" [0x0303] que corresponde a TLS 1.2. Esto se hace para prevenir un problema conocido como intolerancia de versión (*version intolerance*) donde muchos servidores abortan la conexión si se intenta usar el valor "3.4" [0x0304] que corresponde a TLS 1.3.

La verdadera negociación de versión se realiza a través de la extensión obligatoria denominada `supported_versions`. Los numerosos detalles sobre la misma se encuentran en la sección 4.2.1 del estándar.

3.4.8 Pre-Shared Key

En esta versión de TLS, se ha concretado la convergencia entre elementos de diseño del protocolo muy similares tales como la reanudación, los IDs y tickets de sesión de la versión anterior, y el modo 0-RTT, hacia un único modelo en el que su utiliza la clave precompartida.

Una *pre-shared key* (PSK) puede surgir de un *handshake* anterior, o ser provista por un canal separado (*out of band*). La selección del método de intercambio de claves al inicio de la negociación, permite usarla en forma aislada, o combinada con DHE para producir *Forward Secrecy*.

Este nuevo modelo reemplaza la forma en que se reanudaba la conexión en TLS 1.2, con o sin estado por parte del Servidor.

La clave PSK se relaciona también con el envío de datos tempranos en el modo 0-RTT. Esos datos se cifran con un derivado de esta clave, y la mejora en la velocidad se produce a costa de no contar con secreto hacia adelante, y la posibilidad de que se produzcan repeticiones (*replays*).

El nuevo modelo reemplaza además a todos los *Cipher Suites* que se utilizaban anteriormente en TLS 1.2 con clave precompartida como método de intercambio de claves.

El inicio del *handshake* y su modalidad son determinados por las dos extensiones que se incluyen en el CLIENTHELLO, relacionadas con la PSK.

La denominada `psk_key_exchange_modes` define si se va a utilizar una clave precompartida, y si se la va a combinar o no con DHE. Si se elige ese camino, la extensión `pre_shared_key` contiene la estructura que incluye la clave. Por último, si se desea obtener la propiedad de secreto hacia adelante, se agrega la extensión `key_share`.

3.4.9 Otros Cambios

Una de las características que ha sido eliminada del protocolo en el nuevo estándar es la correspondiente a la renegociación, asociada a las siguientes vulnerabilidades:

- Ataque de Renegociación (*Marsh Ray Attack* 2009 [80]).
- Denegación de Servicio (*Renegotiation DoS* 2011 [81]).
- *Triple Handshake* 2014 [82].

Y la misma, ha sido reemplazada por un mecanismo de actualización de claves sencillo *(Easy Key Update*).

La actualización de claves es necesaria porque existen límites criptográficos para la cantidad de información que se puede cifrar utilizando un juego de claves determinado. Esos límites para algoritmos AEAD han sido estudiados por Atul Luykx and Kenneth G. Paterson en una publicación del año 2017 [83].

Los detalles sobre el procedimiento de actualización de claves se referencian en las secciones 4.6.3 y 7.2 del estándar. El mecanismo en cierta forma reemplaza lo que se lograba con el mensaje CHANGECIPHERSPEC.

TLS 1.3 resuelve también un problema existente en la versión anterior con características de falacia circular, relacionado con la verificación del *handshake*. En esa versión, se comprueba toda la negociación computando un MAC en el mensaje FINISHED, pero el algoritmo MAC utilizado en ese mensaje se define en la propia negociación.

Las siguientes vulnerabilidades aprovechan esta retroalimentación degradando (*downgrade*) las conexiones con algoritmos más débiles:

- FREAK, mencionado en [84].
- LOGJAM ya referenciado en [79].
- CurveSwap descrito en [85].

Para poner fin a tal dilema, en el estándar el Servidor firma digitalmente todo el contenido del *handshake*. En TLS 1.2 la firma digital no incluía a los *Cipher Suites* negociados.

Uno de los objetivos de esta versión de TLS, en adición a los de *performance* y seguridad, es el de aumentar la privacidad. Por ello, se ha trabajado sobre el subprotocolo ***Record*** para lograr que revele menos información al análisis de tráfico.

El relleno (*padding*) se puede utilizar en distintas ubicaciones del protocolo. Sin embargo, hay casos de uso con diferentes propósitos.

En principio, existe la posibilidad de utilizar el *padding* en el *handshake*, pero no con fines de proteger la información sino para resolver errores causados por el tamaño del mensaje CLIENTHELLO. Su implementación se describe en la RFC 7685 [86].

Para la privacidad, se realiza el relleno en el subprotocolo ***Record***, añadiendo *bytes* de valor cero al texto plano, previo al cifrado. Cabe notar que en esta nueva versión, también se protegen el tamaño tanto del texto plano como del *padding* agregado, y ciertos campos enviados sin protección tales como el tipo de registro y la versión del protocolo carecen de sentido.

También está permitido el enviar relleno solamente, sin datos, lo que dificulta el análisis del tráfico.

El propio estándar aclara que la estrategia del *padding* es un tema complejo, que excede su alcance, y la sección 5.4 meramente expone su posibilidad de utilización, comentando también que en ciertos casos el relleno puede ser más conveniente si se lo implementa a nivel de la aplicación.

4. Conclusiones

TLS es sin duda uno de los protocolos criptográficos utilizados para la protección de las comunicaciones en red más importantes, y a juicio personal, el más importante de todos.

La versión 1.3 de este protocolo ofrece la mejor combinación de velocidad, seguridad y privacidad, y se trata de un enorme logro producto de más de 20 años de mejoras realizadas al diseño original propuesto por Netscape.

La gran pregunta que aflora a partir de tales afirmaciones, es porqué en Marzo de 2022, y estando vigente la última versión desde Agosto de 2018, todavía no ha sido generalmente adoptada.

Una respuesta simple pero en parte válida, surge de comparar esta transición con la ocurrida al presentarse la versión anterior 1.2. Desde su publicación hasta su adopción general, también transcurrieron varios años, y esto es natural en todo lo que refiere a la implementación de nuevas tecnologías.

Sin embargo, para TLS 1.3, la motivación de la demora es un tanto diferente.

A la fecha, muchos de los grandes protagonistas de la industria tales como los navegadores Chrome, Firefox y Edge, las plataformas Java y OpenSSL, el servidor Apache, los servicios la nube de AWS y Google, y hasta el mismo sistema operativo Windows 10 ya se encuentran actualizados y listos para operar con el nuevo estándar.

Las que están quedando todavía en el camino, son aquellas organizaciones que han invertido gran cantidad de recursos tanto en dinero como en esfuerzo, para ajustar su infraestructura privada a la realidad presentada por la versión anterior, y ahora se ven enfrentadas a los desafíos citados a continuación.

TLS 1.3 introduce varios cambios que mejoran la *performance*, la seguridad y la privacidad de las conexiones, eliminando complejidades y haciendo a la pila del protocolo más simple. Pero todo ello tiene consecuencias para quienes han desplegado soluciones de seguridad de red en sus instalaciones para el cumplimiento, el análisis de riesgos y la búsqueda de amenazas. Entre ellas, figuran principalmente las empresas miembros de la industria financiera, aunque el problema abarca a muchos otros casos.

Entre esos cambios, uno de los más significativos es que los métodos de intercambio de claves RSA y DH estáticos han sido reemplazados por DH Efímero, resultando en la propiedad de “secreto perfecto hacia adelante”. Esta propiedad PFS que es altamente deseable en la red pública, no es tan relevante en el contexto de un centro de datos.

A causa de esta importante modificación al protocolo, se presentan complicaciones en las dos técnicas mediante las cuales se realiza la inspección del tráfico y su descifrado, el modo pasivo y el modo activo.

El modo pasivo es de muy frecuente aplicación en organizaciones de gran tamaño fuertemente reguladas y con considerables requerimientos en lo que concierne al cumplimiento. En ellas, se utiliza el intercambio de claves estático, principalmente con RSA, se comparte la clave privada de cada servidor propio con el dispositivo utilizado para la inspección del tráfico fuera de línea, y se trabaja típicamente sobre las conexiones entrantes.

Con TLS 1.3 dicha modalidad de monitoreo pasivo ya no será posible, y ello trae aparejado que estas organizaciones van a tener que readecuar su arquitectura de seguridad.

El modo activo por su parte consiste en general en la inserción de uno o varios dispositivos que operan en el propio camino de las comunicaciones, tales como un Firewall o un IPS entre muchos otros.

Aquí debemos distinguir entre el descifrado de las comunicaciones entrantes (*inbound*) para proteger contra intrusiones, y el que ocurre en las salientes (*outbound*) con el fin de detectar código maligno y exfiltración de

datos. En el primer caso, dado que se cuenta con la clave privada del servidor y su certificado, el mecanismo es sencillo.

En el segundo, es más complejo dado que las soluciones de seguridad deben simular el certificado del servidor externo con uno privado propio, y distribuirlo a todos los clientes para prevenir las molestas alertas de cada navegador. Y para las aplicaciones SaaS y los dispositivos móviles, donde hay autentificación mutua, esta forma de descifrado se torna inviable.

La adopción del nuevo estándar introducirá nuevas complejidades para este modo activo, incluyendo los ya mencionados cambios en la arquitectura, pero además agregará una mayor latencia, mayores costos y el correspondiente efecto en la performance de toda la instalación.

Si bien en el caso del flujo de datos “Norte-Sur” la utilización de estos dispositivos que por ahora podrán seguir realizando el descifrado es casi inevitable, la inspección del tráfico entre los componentes internos de la red conocido como “Este-Oeste” puede ser cada vez más dificultosa e incluso descartada. En todos los casos, existe un límite a la cantidad de dispositivos de monitoreo activo que se pueden instalar sin afectar tanto el rendimiento como el diseño de las instalaciones.

Cabe comentar que en este sentido, la IETF rechazó propuestas tales como las que mencionamos en el punto 3.4.2, para incluir dentro del estándar opciones que facilitasen el monitoreo pasivo de los datos en tránsito. Y esto compone el requerimiento básico de la actividad de detección de amenazas en un centro de cómputos.

Otra área de impacto potencial radica en que, con TLS 1.3, todos los mensajes del handshake a partir del SERVERHELLO viajan cifrados, y esto incluye a los certificados. Las soluciones de seguridad de red que, operando en modo pasivo inspeccionan los certificados para encontrar situaciones anómalas ya no podrán hacerlo. Incluso desaparece la inspección sin descifrar de los certificados, a partir de la cual se obtienen datos invaluables sobre el tráfico.

Las mejoras aplicadas sobre la latencia del *handshake* en la nueva versión del protocolo son también importantes, e inciden directamente en la experiencia positiva del usuario. Pero la posibilidad de reanudar una sesión con 0-RTT trae aparejadas serias preocupaciones sobre su seguridad y es muy probable que muchas organizaciones directamente no activen tal funcionalidad.

La cuestión redunda entonces en que la adopción general en toda la industria va a seguir tomando su tiempo. Es posible que las versiones 1.2 y 1.3 coexistan durante un largo período, dado que la última no hace que la primera sea insegura, ni la torna obsoleta en términos prácticos. En TLS 1.2 es posible implementar PFS, y operar en forma segura si la configuración es correcta.

Por otra parte, todavía existen implementaciones de TLS 1.0 y 1.1 aún en servicio, y hasta de SSL V3 y V2 en casos muy puntuales.

Las organizaciones que todavía no están al día, van a necesitar entender cómo lograr el ajuste tanto de sus políticas de cumplimiento como de su postura de seguridad. En el caso de la plataforma IoT, es factible incluso que sus dispositivos no se actualicen nunca.

Se presentan en este contexto entonces dos dilemas de interés para el análisis. Uno, la puja entre aquellos que apoyan la mayor privacidad brindada por el nuevo estándar, y la necesidad de monitoreo de las organizaciones. El otro ciertamente relacionado, es la necesidad de implementar cuanto antes la mejor versión del protocolo, versus la complejidad técnica de lograrlo.

En ambos dilemas se ven involucrados el cumplimiento de las regulaciones tales como PCI-DSS, HIPAA y GDPR. Y no es seguro que las mismas exijan que se habilite TLS 1.3 con tanto apremio.

Es importante el mantener presente que la tendencia general está orientada hacia la navegación cifrada, conformando hoy más del 90% del tráfico en Internet [87] y según Gartner más del 54% de las amenazas son

encontradas en ese tráfico protegido [88]. Resuena a través de muchas publicaciones la frase “la red se está volviendo oscura”.

La actualización a TLS 1.3 tiene como protagonista al término “*middleboxes*”, que engloba a todos los dispositivos usados para la inspección activa del tráfico. Esta expresión aparece por primera vez en un estándar de TLS en esta versión en el apéndice D4, donde ya se aborda el problema de compatibilidad de los mismos.

Habiendo transcurrido ya casi cuatro años desde la publicación de la RFC 8446, el proceso de actualización general en la industria se encuentra según mi opinión en un grado intermedio de avance.

Ya mencionamos a algunos de los principales actores que han activado su funcionalidad, y por supuesto muchos de los conocidos fabricantes de hardware de red especializados tales como F5, Cisco, Extrahop y Gigamon ya proveen equipamiento que permite operar con TLS 1.3.

En cuanto a las publicaciones, hay disponibles hoy en la red una cantidad de artículos de investigación y opinión fundada que abordan el problema de esta migración, y realizan recomendaciones al respecto. Desde ya que los mismos fabricantes ofrecen sus propios informes en los cuales resaltan las bondades de sus nuevas soluciones y la “urgencia” de adoptarlas.

Un interesante - aunque expirado en el 2018 - borrador de internet [89] ya habla sobre el impacto de TLS 1.3 en la seguridad de la red. Este documento propone el analizar múltiples problemas en conexiones salientes y entrantes y presenta distintos casos de uso, por lo que es muy aconsejable el revisarlo.

Entre las numerosas recomendaciones que han surgido para hacer frente a la transición, se destaca un informe pago ofrecido por Gartner que refiere a las alternativas factibles para la inspección pasiva [90].

En él sugiere principalmente en cuanto a la seguridad, una primera opción que es muy considerada en general y consiste en utilizar TLS 1.3

para el tráfico externo, y TLS 1.2 para poder seguir monitoreando el interno. Este tipo de implementación es relativamente sencilla, aunque se debe cuidar de no degradar demasiado la seguridad de la conexión, y se trata de una solución para el corto plazo.

Como segunda opción, propone la instalación de agentes en los servidores propios, que se ocupen de reenviar las claves de cada sesión al dispositivo utilizado para el análisis. Y como tercera una similar, que consiste en agregar un ADC (*Application Delivery Controller*) que realice el trabajo de los agentes. Esta técnica es conocida como *session key forwarding*.

Las recomendaciones de Gartner van un paso más allá, sugiriendo la adopción de un mecanismo conocido como DOIM (*Decrypt Once – Inspect Many*), en donde un hardware de red especializado, tal como el ya ofrecido por la empresa Gigamon [91], realiza una sola vez el descifrado, y copia su resultado a múltiples artefactos dedicados al análisis pasivo del tráfico.

Esto se realiza en una "zona de descifrado", creada por el propio aparato. También se aconseja el instalar terminaciones de TLS en los WAF, los *proxies* y los *gateways* de internet.

Las sugerencias incluyen a la creación de políticas de descifrado internas, el uso posible de agentes, y guías para la implementación de mecanismos de seguridad para DNS y HTTPS que afecten a la intercepción de TLS.

Por último, instan a bloquear directamente todo el tráfico no descifrable que ingrese o egrese de la organización, salvo casos específicamente aprobados.

Aprovechando el comentario realizado por Gartner sobre el interceptar a TLS, cabe mencionar un artículo técnico de Symantec que aborda cómo hacerlo de forma "responsable" [92], y en él son relevantes los ejemplos de posibles casos de degradación de versión, y la referencia a las protecciones diferentes contra la misma que implementan la versión 1.2 y la 1.3. En este sentido, también hay un informe de US-CERT [93] que alerta sobre la reducción de la seguridad que la intercepción implica.

La preocupación general sobre los peligros de interceptar TLS de forma inadecuada ya había sido presentada en un importante trabajo académico del año 2017 [94].

Un informe [95] de un prominente analista de Forrester David Holmes, se arriesgó a vaticinar en Agosto de 2020 que restan dos años a partir de entonces para preparar las herramientas de seguridad con el fin de adecuarse a la nueva era.

Aquí combina los efectos de TLS 1.3 con los de DNS sobre HTTPS (DoH) que ya es un estándar [96] aunque muy criticado por el propio Paul Vixie referido como el padre de DNS, y el cifrado de la extensión SNI todavía en estudio [97] pero ya implementada por ejemplo por el proveedor Cloudflare.

Y resalta que las tres tecnologías sumadas, hacen mucho más dificultosa la visibilidad sobre el tráfico que necesitan por ejemplo, los ya mencionados actores de la comunidad financiera, en donde uno de sus representantes alerta que el costo por conexión de actualizar las plataformas en servicio podría ser cercano a un millón de dólares, dado que para su tipo de tráfico en línea no pueden existir demoras.

En el informe también se recomienda el terminar TLS 1.3 y utilizar TLS 1.2 para la inspección del tráfico entrante, ya sea en instalaciones propias o en la nube. Actualizar los cortafuegos para que se pueda inspeccionar TLS 1.3 en el tráfico saliente e invertir en la técnica de *session key forwarding* para poder seguir realizando el monitoreo pasivo.

A eso le suma el considerar el uso del aprendizaje automatizado (*machine learning*) sobre los metadatos restantes en las comunicaciones, y lo que llama “recuperar” la visibilidad sobre DNS, estableciendo alguna solución personalizada que permita obtener y analizar una copia del tráfico de consultas derivados a terceros cuando se aplica DoH.

Como técnicas adicionales, sugiere también la captura de claves de sesión desde la memoria de los servidores y los clientes, aplicar inteligencia artificial al análisis del tráfico cifrado (ETA), y la ya conocida técnica de

fingerprinting [98]. Además, se recomienda el aumentar la categoría de riesgo para las comunicaciones con SNI cifrado.

Otra novedad adicional recomendada, que la empresa Cisco ya ofrece en sus productos de la línea Stealthwatch y en el sistema operativo FTD, para las conexiones salientes consiste en pausar el tráfico sospechoso y hacer una conexión paralela con el fin de descifrar el certificado del servidor externo y verificar su autenticidad.

La empresa citada la llama "*TLS Server Identity Discovery*", Forrester la denomina "*side-band checking*" y puede lograrse también mediante la configuración programática (*scripts*) de muchos dispositivos y herramientas de seguridad existentes.

El hecho de tener que aplicar el descifrado para TLS 1.3 tan pronto como sea posible, es desfavorable para la experiencia del usuario, donde si la conexión no necesitaba de tal inspección, se debe cortar y ser establecida nuevamente sin intermediación.

Todas estas sugerencias deben ser acompañadas de pruebas internas previas y simulaciones, que permitan no sólo comprobar las soluciones a implementar sino comparar su efectividad y eficiencia con respecto a lo que ya se encuentra en servicio.

Por su parte, la empresa Karspersky [99] comenta que, a medida que el tráfico cifrado con TLS 1.3 prevalece, se debe poner mayor foco en la seguridad de las estaciones de trabajo, desplegando sistemas de protección tales como el propio, que ofrezcan un entorno centralizado de detección. Pero también indican que esto no es suficiente, sin el adecuado entrenamiento del personal interno.

A través de todo lo expuesto, hemos podido constatar que la migración hacia TLS 1.3 es tan compleja como necesaria. Nos encontramos en un período de transición de duración todavía indefinida, de tamaña relevancia y sin duda más que fascinante.

La “Era de TLS 1.3” recién comienza, con una versión elegante, eficiente y extensible. Esto último de gran relevancia por ejemplo, para la futura aplicación de algoritmos post-cuánticos.

El conocimiento sobre el funcionamiento de este protocolo tanto en su versión actual como en la anterior nunca ha sido tan importante para el profesional de TI y de Seguridad Informática como lo es hoy.

En particular, el largo viaje que exigió la confección de este trabajo me ha brindado la grata posibilidad de observar con admiración y reverencia la aplicación de la criptografía que conocimos en forma teórica en esta Maestría – y por ende de las matemáticas - a problemas prácticos reales en el protocolo que protege nada menos que a todas nuestras comunicaciones en la red Internet.

En palabras del gran Taher El Gamal, padre de SSL: “La criptografía es el uso más bello de las matemáticas que he visto”.

5. Bibliografía

[I1] CloudFlare, “A Detailed Look at RFC 8446 (a.k.a. TLS 1.3)”, https://blog.cloudflare.com/rfc-8446-aka-tls-1-3/, (Consultada el 6/3/2022).

[I2] Qualys, “SSL Pulse”, https://www.ssllabs.com/ssl-pulse/, (Consultada el 6/3/2022).

[1] Freier A., Karlton P., Kocher P., “The Secure Sockets Layer (SSL) Protocol Version 3.0”, RFC 6101, https://datatracker.ietf.org/doc/html/rfc6101 (Consultada el 1/3/2021).

[2] R. Oppliger, “SSL and TLS Theory and Practice Second Edition”, Artech House, Norwood, MA, 2016.

[3] Fortezza, https://wikioes.icu/wiki/Fortezza. (Consultada el 15/2/2022).

[4] Krawczyk H., Bellare M., Canetti R., “HMAC: Keyed Hashing for Message Authentication”, RFC 2104, https://datatracker.ietf.org/doc/html/rfc2104, (Consultada el 1/3/2021).

[5] Dierks T., Allen C., “The TLS Protocol Version 1.0”, RFC 2246, https://datatracker.ietf.org/doc/html/rfc2246, (Consultada el 1/3/2021).

[6] Dierks T., Rescorla E., “The Transport Layer Security (TLS) Protocol Version 1.1”, RFC 4346, https://datatracker.ietf.org/doc/html/rfc4346, (Consultada el 1/8/2021).

[7] Dierks T., Rescorla E., “The Transport Layer Security (TLS) Protocol Version 1.2”, RFC 5246, https://datatracker.ietf.org/doc/html/rfc5246, (Consultada el 1/8/2021).

[8] Moriarty K., Farrell S., “Deprecating TLS 1.0 y TLS 1.1”, RFC 8996, https://datatracker.ietf.org/doc/html/rfc8996, (Consultada el 1/8/2021).

[9] Moriai S., Kato A., Kanda M., “Addition of Camellia Cipher Suites to Transport Layer Security (TLS)”, RFC 4132, https://datatracker.ietf.org/doc/html/rfc4132, (Consultada el 5/8/2021).

[10] Kato A., Kanda M., Kanno S., “Camellia Cipher Suites for TLS”, RFC 5932, https://datatracker.ietf.org/doc/html/rfc5932, (Consultada el 5/8/2021).

[11] Kanda M., Kanno S., “Addition of the Camellia Cipher Suites to Transport Layer Security”, RFC 6367, https://datatracker.ietf.org/doc/html/rfc6367, (Consultada el 5/8/2021).

[12] Kim W., Lee J., Park J., Kwon D., “Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)”, RFC 6209, https://datatracker.ietf.org/doc/html/rfc6209, (Consultada el 5/8/2021).

[13] Chown P., “Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)”, RFC 3268, https://datatracker.ietf.org/doc/html/rfc3268, (Consultada el 7/8/2021).

[14] Medvinsky A., Hur M., “Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)”, RFC 2712, https://datatracker.ietf.org/doc/html/rfc2712, (Consultada el 7/8/2021).

[15] McGrew D., “An Interface and Algorithms for Authenticated Encryption”, RFC 5116, https://datatracker.ietf.org/doc/html/rfc5116, (Consultada el 7/8/2021).

[16] Saloway J., Choudhury A., McGrew D., “AES-GCM Cipher Suites for TLS”, RFC 5288, https://datatracker.ietf.org/doc/html/rfc5288, (Consultada el 7/8/2021).

[17] Rescorla E., “TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode”, RFC 5289, https://datatracker.ietf.org/doc/html/rfc5289, (Consultada el 9/8/2021).

[18] McGrew D., Bailey D., “AES-CCM Cipher Suites for Transport Layer Security (TLS)”, RFC 6655, https://datatracker.ietf.org/doc/html/rfc6655, (Consultada el 9/8/2021).

[19] Eronen P., Tschofenig H., “Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)”, RFC 4279, https://datatracker.ietf.org/doc/html/rfc4279, (Consultada el 10/8/2021).

[20] Badra M., “Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode”, RFC 5487, https://datatracker.ietf.org/doc/html/rfc5487, (Consultada el 10/8/2021).

[21] Badra M., Hajjeh I., “ECDHE_PSK Cipher Suites for Transport Layer Security (TLS)”, RFC 5489, https://datatracker.ietf.org/doc/html/rfc5489, (Consultada el 12/8/2021).

[22] Blumenthal U., Goel P., “Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)”, RFC 4785, https://datatracker.ietf.org/doc/html/rfc4785, (Consultada el 12/8/2021).

[23] Eastlake D., “Transport Layer Security (TLS) Extensions: Extension Definitions”, RFC 6066, https://datatracker.ietf.org/doc/html/rfc6066, (Consultada el 20/8/2021).

[24] Delignat-Lavaud A., Bhargavan K. , “Virtual Host Confusion: Weaknesses and Exploits”, https://bh.ht.vc/vhost_confusion.pdf, (Consultada el 20/8/2021).

[25] Pettersen Y., “The Transport Layer Security (TLS) Multiple Certificate Status Request Extension”, RFC 6961, https://datatracker.ietf.org/doc/html/rfc6961 (Consultada el 20/8/2021).

[26] Santesson S., Medvinsky A. , Ball J., “TLS User Mapping Extension”, RFC 4681, https://datatracker.ietf.org/doc/html/rfc4681, (Consultada el 20/8/2021).

[27] Santesson S., “TLS Handshake Message for Supplemental Data”, RFC 4680, https://datatracker.ietf.org/doc/html/rfc4680, (Consultada el 20/8/2021).

[28] Brown M., Housley R., “Transport Layer Security (TLS) Authorization Extensions”, RFC 5878, https://datatracker.ietf.org/doc/html/rfc5878, (Consultada el 29/8/2021).

[29] Mavrogiannopoulos N., Gillmor D., “Using OpenPGP Keys for Transport Layer Security (TLS) Authentication”, RFC 6091, https://datatracker.ietf.org/doc/html/rfc6091, (Consultada el 2/9/2021).

[30] Blake-Wilson S. y otros, “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)”, RFC 4492, https://datatracker.ietf.org/doc/html/rfc4492, (Consultada el 2/9/2021).

[31] Nir Y., Josefsson S., Pegourie-Gonnard M., “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier”, RFC 8422, https://datatracker.ietf.org/doc/html/rfc8422, (Consultada el 2/9/2021).

[32] Taylor D., Wu T., y otros, “Using the Secure Remote Password (SRP)”, RFC 5054, https://datatracker.ietf.org/doc/html/rfc5054, (Consultada el 10/9/2021).

[33] The Stanford SRP Homepage, http://srp.stanford.edu/, (Consultada el 10/9/2021).

[34] Seggemann R., Tuexen M., Williams M., “Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension”, RFC 6520, https://datatracker.ietf.org/doc/html/rfc6520, (Consultada el 10/9/2021).

[35] Friedl S. y otros, “Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension”, RFC 7301, https://datatracker.ietf.org/doc/html/rfc7301, (Consultada el 10/9/2021).

[36] Laurie B., Langley A., Kasper E., “Certificate Transparency”, RFC 6962, https://datatracker.ietf.org/doc/html/rfc6962, (Consultada el 10/9/2021).

[37] Certificate Transparency, https://certificate.transparency.dev/, (Consultada el 10/9/2021).

[38] Wouters P. y otros, “Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)”, RFC 7250, https://datatracker.ietf.org/doc/html/rfc7250, (Consultada el 10/9/2021).

[39] Gutmann P., “Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)”, RFC 7366, https://datatracker.ietf.org/doc/html/rfc7366, (Consultada el 15/9/2021).

[40] Bhargavan K. y otros. “Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension”, RFC 7627, https://datatracker.ietf.org/doc/html/rfc7627, (Consultada el 15/9/2021).

[41] Salowey J. y otros. “Transport Layer Security (TLS) Session Resumption without Server-Side State”, RFC 5077, https://datatracker.ietf.org/doc/html/rfc5077, (Consultada el 15/9/2021).

[42] Rescorla E. y otros. “Transport Layer Security (TLS) Renegotiation Indication Extension”, RFC 5746, https://datatracker.ietf.org/doc/html/rfc5746, (Consultada el 15/9/2021).

[43] Hollenbeck S., “Transport Layer Security Protocol Compression Methods”, RFC 3749, https://datatracker.ietf.org/doc/html/rfc5746, (Consultada el 15/9/2021).

[44] Rescorla E., “The Transport Layer Security (TLS) Protocol Version 1.3”, RFC 8446, https://datatracker.ietf.org/doc/html/rfc8446, (Consultada el 1/8/2021).

[45] Rescorla E., “TLS Wish List”, https://www.ietf.org/proceedings/87/slides/slides-87-tls-5.pdf, (Consultada el 1/8/2021).

[46] IETF TLS Working Group, “IETF Tutorial”, https://www.youtube.com/watch?v=9kB2ZJUVUuE&t=2313s, (Consultada el 1/8/2021).

[47] Balducci A., “What's New in TLS 1.3”, https://www.youtube.com/watch?v=eTJjMGClri4, (Consultada el 5/8/2021).

[48] Valsorda F., Sullivan N., “Deploying TLS 1.3: the great, the good and the bad”, 33c3, https://media.ccc.de/v/33c3-8348-deploying_tls_1_3_the_great_the_good_and_the_bad, (Consultada el 3/8/2021).

[49] Nir Y., Langley A., “ChaCha20 and Poly1305 for IETF Protocols”, RFC 8439, https://datatracker.ietf.org/doc/html/rfc8439, (Consultada el 3/8/2021).

[50] McGrew D., Bailey D., “AES-CCM Cipher Suites for Transport Layer Security (TLS)”, RFC 6655, https://datatracker.ietf.org/doc/html/rfc6655, (Consultada el 3/8/2021).

[51] NIST, “Secure Hash Standard (SHS)”, FIPS 180-4, https://csrc.nist.gov/publications/detail/fips/180/4/final, (Consultada el 3/8/2021).

[52] RC4 NOMORE, “Numerous Occurrence MOnitoring & Recovery Exploit”, https://www.rc4nomore.com, (Consultada el 3/8/2021).

[53] Sweet32, “Birthday attacks on 64-bit block ciphers in TLS and OpenVPN”, https://sweet32.info, (Consultada el 3/8/2021).

[54] SLOTH, “Security Losses from Obsolete and Truncated Transcript Hashes”, https://www.mitls.org/pages/attacks/SLOTH, (Consultada el 3/8/2021).

[55] Vaudenay S., “Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS...”, https://link.springer.com/chapter/10.1007/3-540-46035-7_35, (Consultada el 3/8/2021).

[56] Boneh D., Brumley D., “Remote timing attacks are practical”, https://crypto.stanford.edu/~dabo/abstracts/ssl-timing.html, (Consultada el 5/8/2021).

[57] Banach Z., “How the BEAST Attack Works”, https://www.netsparker.com/blog/web-security/how-the-beast-attack-works, (Consultada el 10/8/2021).

[58] Alfardan N., Patterson K., “Lucky Thirteen: Breaking the TLS and DTLS Record Protocols”, http://www.isg.rhul.ac.uk/tls/Lucky13.html, (Consultada el 10/8/2021).

[59] Möller B., Duong T., Kotowicz K., “This POODLE Bites: Exploiting The SSL 3.0 Fallback”, https://www.openssl.org/~bodo/ssl-poodle.pdf, (Consultada el 12/8/2021).

[60] Albrecht M., Paterson K., “Lucky Microseconds: A Timing Attack on Amazon’s s2n Implementation of TLS”, https://www.iacr.org/archive/eurocrypt2016/96650136/96650136.pdf, (Consultada el 8/8/2021).

[61] Böck H. y otros, “Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS”, https://eprint.iacr.org/2016/475.pdf, (Consultada el 8/8/2021).

[62] Green M. y otros, “Data Center use of Static Diffie-Hellman in TLS 1.3”, https://tools.ietf.org/id/draft-green-tls-static-dh-in-tls13-01.html, Internet Draft. (Consultada el 12/8/2021).

[63] Krawczyk H., Eronen P., “HMAC-based Extract-and-Expand Key Derivation Function (HKDF)”, RFC 5869, https://tools.ietf.org/pdf/rfc5869.pdf, (Consultada el 20/7/2021).

[64] Y. Nir, S. Josefsson, M. Pegourie-Gonnard, “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier”, RFC 8422, (Consultada el 10/4/2021).

[65] NIST, “Digital Signature Standard (DSS)”, FIPS 186-4, https://csrc.nist.gov/publications/detail/fips/186/4/final, (Consultada el 12/4/2021).

[66] A. Langley, M. Hamburg, S. Turner, “Elliptic Curves for Security”, RFC 7748, https://datatracker.ietf.org/doc/html/rfc7748, (Consultada el 1/4/2021).

[67] ANSI, “Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)”, https://standards.globalspec.com/std/1955141/ANSI%20X9.62, (Consultada el 2/6/2021).

[68] Josefsson S., Liusvaara I, “Edwards-Curve Digital Signature Algorithm (EdDSA)”, RFC 8032, https://tools.ietf.org/pdf/rfc8032.pdf, (Consultada el 2/6/2021).

[69] Moriarti K. y otros, “PKCS #1: RSA Cryptography Specifications Version 2.2”, RFC 8017, https://datatracker.ietf.org/doc/html/rfc8017, (Consultada el 10/8/2021).

[70] Bleichenbacher D., “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PCKS #1”, https://link.springer.com/content/pdf/10.1007%2FBFb0055716.pdf, (Consultada el 10/8/2021).

[71] Jager T., Schwenk J., Somorovsky J., “On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption”, https://www.nds.ruhr-uni-bochum.de/media/nds/veroeffentlichungen/2015/08/21/Tls13QuicAttacks.pdf, (Consultada el 10/8/2021).

[72] Aviram N. y otros., “DROWN: Breaking TLS using SSLv2”, https://drownattack.com/drown-attack-paper.pdf, (Consultada el 10/8/2021).

[73] Ristic I., “CRIME: Information Leakage Attack against SSL/TLS”, https://blog.qualys.com/product-tech/2012/09/14/crime-information-leakage-attack-against-ssltls, (Consultada el 10/8/2021).

[74] Be’ery T., “A Perfect CRIME? TIME Will Tell”, https://www.imperva.com/docs/adc/ADC_2013_A_Perfect_CRIME-TIME_Will_Tell.pdf, (Consultada el 17/8/2021).

[75] Gluck Y., Harris N., Prado A., “BREACH: Reviving the CRIME attack”, http://breachattack.com/, (Consultada el 17/8/2021).

[76] D. Gillmor, “Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)”, RFC 7919, https://datatracker.ietf.org/doc/html/rfc7919, Agosto 2016. (Consultada el 1/8/2021).

[77] StackExchange, “Why does TLS 1.3 deprecate custom DHE groups?”, https://security.stackexchange.com/questions/181820/why-does-tls-1-3-deprecate-custom-dhe-groups, (Consultada el 1/8/2021).

[78] StackExchange, “Why is Mozilla recommending predefined DHE groups?”, https://security.stackexchange.com/questions/149811/why-is-mozilla-recommending-predefined-dhe-groups, (Consultada el 1/8/2021).

[79] Adrian D. y otros, “Weak Diffie-Hellman and the Logjam Attack”, https://weakdh.org/, (Consultada el 1/8/2021).

[80] Saloway J., Rescorla E., “TLS Renegotiation Vulnerability”, https://www.ietf.org/proceedings/76/slides/tls-7.pdf, IETF-76, (Consultada el 10/5/2021).

[81] Ristic I., “TLS Renegotiation and Denial of Service Attacks”, https://blog.qualys.com/product-tech/2011/10/31/tls-renegotiation-and-denial-of-service-attacks, (Consultada el 10/5/2021).

[82] miTLS, “Triple Handshakes Considered Harmful: Breaking and Fixing Authentication over TLS”, https://mitls.org/pages/attacks/3SHAKE, (Consultada el 10/5/2021).

[83] Luykx A., Paterson K., “Limits on Authenticated Encryption Use in TLS”, https://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf, (Consultada el 14/8/2021).

[84] Beurdouche B. y otros. “A Messy State of the Union: Taming the Composite State Machines of TLS”, https://cacm.acm.org/magazines/2017/2/212438-a-messy-state-of-the-union/fulltext, (Consultada el 15/8/2021).

[85] Valenta L. y otros. “In search of CurveSwap: Measuring elliptic curve implementations in the wild”, https://eprint.iacr.org/2018/298.pdf, Marzo 2018. (Consultada el 15/8/2021).

[86] Langley A., “A Transport Layer Security (TLS) ClientHello Padding Extension”, RFC 7685, https://datatracker.ietf.org/doc/html/rfc7685, Octubre 2015. (Consultada el 16/8/2021).

[87] InfoTech News, “HTTPS encryption traffic on the Internet has exceeded 90%”, https://meterpreter.org/https-encryption-traffic/, (Consultada el 5/2/2022).

[88] ExtraHop, “Gartner Report on Handling Challenges with TLS 1.3 and Passive Decryption”, https://www.extrahop.com/company/blog/2020/gartner-report-on-handling-challenges-with-tls-1.3/, (Consultada el 5/2/2022).

[89] Andreasen F. y otros, “TLS 1.3 Impact on Network-Based Security”, https://tools.ietf.org/id/draft-camwinget-tls-use-cases-00.xml, (Consultada el 5/2/2022).

[90] Gartner Research, “Demystifying the Impact of TLS 1.3 on TLS Inspection”, https://www.gartner.com/en/documents/3978669/demystifying-the-impact-of-tls-1-3-on-tls-inspection, (Consultada el 5/2/2022).

[91] Gigamon, “Tools Challenged by SSL Decryption?”, https://www.gigamon.com/products/optimize-traffic/traffic-intelligence/gigasmart/ssl-tls-decryption.html, (Consultada el 5/2/2022).

[92] Symantec, “Responsibly Intercepting TLS and the Impact of TLS 1.3”, https://docs.broadcom.com/doc/responsibly-intercepting-tls-and-the-impact-of-tls-1.3.en, (Consultada el 12/2/2022).

[93] TechTarget, “HTTPS interception, middlebox models under fire”, https://www.techtarget.com/searchsecurity/news/450415345/HTTPS-interception-middlebox-models-under-fire, (Consultada el 12/2/2022).

[94] Durumerik Z. y otros, “The Security Impact of HTTPS Interception”, https://jhalderm.com/pub/papers/interception-ndss17.pdf, (Consultada el 12/2/2022).

[95] Holmes D, “Maintain Security Visibility In The TLS 1.3 Era”, https://resources.softwaretrends.com/resources/122418/forrester-maintain-security-visibility-in-the-tls-13-era, (Consultada el 12/2/2022).

[96] Hoffman P., McManus P., “DNS Queries over HTTPS (DoH)”, RFC 8484, https://datatracker.ietf.org/doc/html/rfc8484, (Consultada el 12/2/2022).

[97] Rescorla E. y otros, “TLS Encrypted Client Hello”, Internet Draft, https://datatracker.ietf.org/doc/draft-ietf-tls-esni/?include_text=1, (Consultada el 19/2/2022).

[98] Althouse J., ““TLS Fingerprinting with JA3 and JA3S”, https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967?gi=314fde49f7eb, (Consultada el 19/2/2022).

[99] Karpersky, “Migrating to TLS 1.3: What are the cybersecurity challenges?”, https://www.kaspersky.com/blog/secure-futures-magazine/tls-1-3-network/28278/, (Consultada el 6/3/2022).