University of Castilla-La Mancha

A publication of the

Department of Computer Science

Verificación de TLS 1.0 y extensiones para redes inalámbricas

by

Llanos Tobarra and Diego Cazorla

Technical Report **#DIAB-04-11-02** November 2004

DEPARTAMENTO DE INFORMÁTICA
ESCUELA POLITÉCNICA SUPERIOR
UNIVERSIDAD DE CASTILLA-LA MANCHA
Campus Universitario s/n
Albacete - 02071 - Spain
Phone +34.967.599200, Fax +34.967.599224

Verificación de TLS 1.0 y extensiones para redes inalámbricas

Llanos Tobarra Diego Cazorla

{mtobarra,dcazorla}@info-ab.uclm.es

Resumen

El protocolo TLS v1.0 es el protocolo desarrollado por el Internet Engineering Task Force (IETF) para garantizar la seguridad en una red. Junto con la versión básica del protocolo, se ha publicado un mecanismo de extensión que intenta adaptar TLS a las limitaciones asociadas a las redes inalámbricas. Verificaremos TLS básico y aquellas extensiones que influyen en la seguridad del protocolo.

1. Introducción

1.1. El problema clásico de la criptografía

La aplicación de métodos formales para protocolos criptográficos comienza centrándose básicamente en un problema de distribución de claves. Se centraba en dos entidades, A y B, que querían hablar de forma segura pero eran molestados por un intruso M que podía capturar, modificar y eliminar mensajes del tráfico en la red (ver fig. 1).

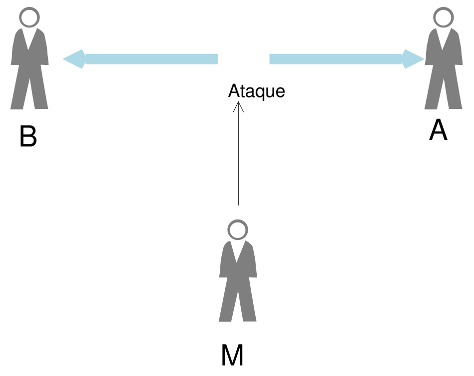


Figura 1: Sistema criptográfico abstracto

Este modelo conocido como modelo Dolev-Yao ([**?**]) puede adaptarse para diversas formas de comunicación en las redes actuales donde se requiera un intercambio de información entre dos entidades A y B y se pretendan garantizar las propiedades de autentificación y de confidencialidad. La autentificación es la propiedad que nos permiten identificarnos en una conversación. Hace referencia al proceso de verificar la identidad de alguien. Por otra parte la confidencialidad es el concepto consistente en evitar que la información sensible sea accesible a personas no autorizadas. Clásicamente la criptografía ha intentado desarrollar protocolos que solo garantizaran estas dos propiedades básicamente basándose en el modelo de Dolev-Yao.

Sin embargo debido a la evolución en las redes y la gran cantidad de usos que se les dan aparecen nuevos problemas:

- Protocolos de e-comerce que permiten a un cliente realizar compras en un determinado mercado y se encargan de transacciones financieras, que dependen tanto de las propiedades clásicas de seguridad garantizadas por la criptografía como de la disponibilidad y confianza del sistema.
- Protocolos de autentificación y acceso a recursos con una infraestructura distribuida, donde se comparten los recursos distribuidos en comunidades de usuarios dinámicas, llamadas organizaciones virtuales; las políticas de control de acceso se basan en pertenecer a una organización o papeles en las organizaciones; y las comunicaciones involucran a tres entidades: el recurso, el usuario que requiere el recurso y el servicio de pertenencia a la organización (que certifica la pertenencia del usuario o el papel del usuario en la red).
- Protocolos de gestión de derechos para el acceso a determinados recursos, como archivos o bases de datos, que requieren su ejecución distribuida en diversos procesadores.

Las redes no solo han de defenderse contra intrusos que intentan obtener los secretos o participantes deshonestos, sino que deben ser robustas contra los ataques de denegación de servicio, y, en algunos casos, contra los analistas de tráfico.

Por eso conforme se complican las necesidades, se complican los protocolos para garantizar esas necesidades, es más difícil detectar fallos en los protocolos de forma manual y se imponen métodos complejos y efectivos para encontrar esas faltas. Así aparecen dos propiedades nuevas que deseamos para nuestras redes, la integridad y la disponibilidad (vivacidad). La disponibilidad determina que los recursos estén disponibles para las entidades autorizadas y la integridad incide en que los recursos de un sistema sean modificados solo por entes autorizados. La modificación incluye escribir, cambiar, cambiar de estado, suprimir y crear.

Además, en las redes inalámbricas se añaden a las necesidades de seguridad otras cuestiones a tener en cuenta. Se deben diseñar protocolos de seguridad robustos que tengan en cuenta el tipo de dispositivos que se emplean en las comunicaciones y en ocasiones el medio de transmisión. Los dispositivos inalámbricos suelen tener un espacio limitado de memoria que no les permite manejar gran cantidad de información para ejecutar un protocolo, suelen depender de la duración de baterías como fuentes de alimentación, y en algunos casos la capacidad de computación es reducida, o se tiende a disminuir. Hay tecnologías inalámbricas de redes personales que tienen un ancho de banda reducido. Todas estas limitaciones tienen que tenerse en cuenta cuando se diseñan algoritmos para estas redes.

Y aunque aquí nos centramos en una criptografía orientada a las redes, también hay otros tipos de sistemas que emplean la criptografía hoy en día como son los sistemas de protección contra la piratería en CDs y DVDs. Estos sistemas también colaboran al desarrollo de protocolos seguros para almacenar la información en dispositivos y su transmisión segura.

1.2. Criptografía y verificación automática

Con la necesidad de enviar mensajes confidenciales aparece la criptografía. La criptografía recoge un conjunto de técnicas que se emplean para conseguir la confidencialidad y autentificación en una comunicación. Estas técnicas suelen basarse en cifrar o codificar un mensaje desordenando los datos de forma que el mensaje quede incomprensible siguiendo un patrón. Gracias a este patrón o clave luego podrá recuperarse el mensaje.

Los protocolos de criptográfica son protocolos que usan la criptografía para publicar claves y autentificar entidades y datos en una red. Se asume que el medio de comunicación es hostil, ya que hay en ella intrusos que pueden leer, modificar y borrar el tráfico o pueden controlar a una o más entidades suplantando su identidad en la red. El protocolo debe haber sido diseñado para que aunque existan estas amenazas la conversación entre dos entidades deba producirse de forma segura, garantizando la confidencialidad y la autentificación de los participantes.

Para probar que un protocolo garantiza las propiedades de seguridad se suelen validar usando diferentes técnicas y viendo como se comporta frente a una serie de ataques que se pueden clasificar en dos tipos de agresiones, los ataques pasivos y los ataques activos.

Una de las técnicas de validación serían los **métodos formales**. Los métodos formales son una combinación de un modelo bien matemático o bien lógico de un sistema y sus requisitos, junto con un procedimiento efectivo para determinar cuando una prueba de que un sistema satisface sus requerimientos es correcta. Consideramos la prueba completa, aunque en la definición incluimos técnicas incompletas como los analizadores de modelos de estado finitos. Estos sistemas no dan una prueba completa de seguridad para todo el espacio de estados, pero permiten una descripción precisa de las condiciones en las cuales sostiene sus conclusiones, junto con un procedimiento efectivo para comprobarlos. Hay tres aproximaciones principales para el análisis de protocolos (ver fig, 2):

- **Lógicas de creencias**. Consisten en un conjunto de operadores modales que describen las relaciones entre los participantes y los datos, un conjunto de creencias que pueden ser sostenidas por los participantes y un conjunto de reglas de inferencia para derivar nuevas creencias a partir de las viejas. Al operar en un nivel de abstracción son herramientas mas débiles que las siguientes aunque ofrecen la ventaja de su simplicidad y son decidibles y a menudo computables de forma eficiente. A pesar de su simplicidad permitieron señalar defectos graves en protocolos de seguridad.

- **Exploración del espacio de estados** (model checking). Son técnicas basadas en el modelo de Dolev-Yao donde se define un espacio de estados que es explorado mediante una herramienta para determinar si hay algún camino a través del espacio correspondiente donde un ataque del intruso tenga éxito.

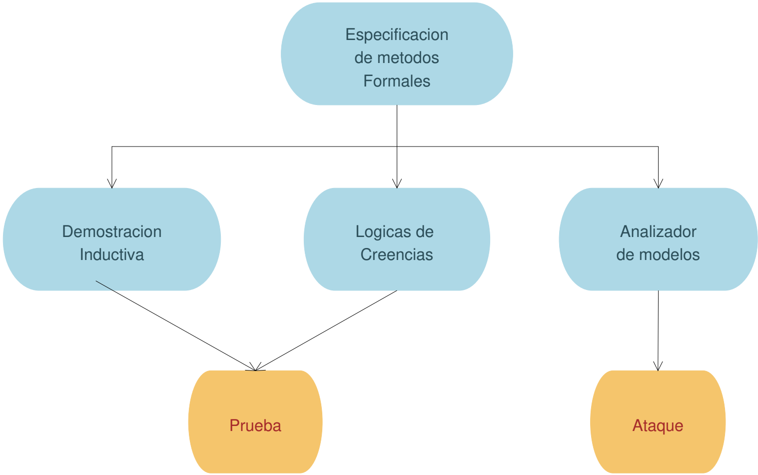


Figura 2: Clasificación de los métodos formales

- **Demostraciones inductivas**. Son de naturaleza más interactiva y se basan en la inferencia usando una serie de teoremas para la demostración de las propiedades de un protocolo.

La primera aproximación y la última tienen como objetivo probar la corrección del protocolo con respecto a las propiedades de la seguridad, mientras que el segundo busca los ataques exitosos.

Las herramientas basadas en los métodos formales han resultado muy útiles dentro del campo de la seguridad. Hasta la aceptación de estos métodos, los protocolos de seguridad eran verificados por analistas humanos. En muchas ocasiones no se comprobaban casi todas las trazas posibles del protocolo, omitiendo fallos. Gran parte del éxito de los métodos formales se ha debido a la detección de defectos en protocolos bien conocidos.

Pero además de verificar protocolos reales, los métodos formales se pueden aplicar en tiempo de diseño, ahorrando tiempo y esfuerzo. El diseñador, a partir solo de la especificación puede comprobar si el protocolo cumple unos requisitos , y en caso de no cumplirlo, la mayoría de las herramientas suelen dar contraejemplos que ayudan a mejorar el protocolo. No es necesaria la implementación, como ocurría con los analistas humanos, para analizar el comportamiento del protocolo. Otra ventaja es que son imparciales, que no se ven condicionadas como un analista humano por presiones externas.

Sin embargo, las herramientas formales dependen de la especificación que se de tanto del protocolo como de los requisitos a verificar. Si la especificación no es correcta o no se ciñe a la realidad o bien los requisitos que verifican no son los mismos que se desean comprobar, lo lógico es que el análisis no permita afirmar la seguridad del protocolo.

Por otra parte, los protocolos no se ejecutan aislados. Además de especificar el protocolo, es necesario especificar el entorno de ejecución. Y generalmente, los entornos de ejecución suelen ser variables. Así que, a pesar de las grandes ventajas de los métodos formales, no nos permiten asegurar la completa seguridad de un protocolo.

Hasta hace poco, el principal grueso del trabajo se ha dedicado al análisis de sistemas discretos. De hecho, muchos problemas de seguridad en el análisis de protocolos pueden formularse en términos de las propiedades de un sistema discreto. Varias entidades desean intercambiar datos de forma segura, pero son objeto del ataque de un intruso que puede ejecutar un conjunto finito de operaciones. Luego, los fallos del sistema son explotados por el intruso, y si esos ataques en principio no son intuitivos puede ayudar un análisis formal discreto a encontrarlos y evitarlos así como especificar el entorno restrictivo en que tienen lugar. Por esta razón, hace tiempo que se vienen utilizando los métodos formales para análisis de la seguridad de los protocolos criptográficos.

1.3. Casper

Casper es una herramienta basada en el algebra de procesos CSP (1985,[?]) y su herramienta de model Checking FDR (1994, [?]) que ha sido desarrollada por Gavin Lowe, Philippa Bradfoot y Mei Lin Hui del Oxford University Computing Laboratory (ver [?, ?, ?]). Casper en sí no es más que una adaptación de la herramienta FDR de propósito general a los protocolos de seguridad. Realmente Casper lo que hace es traducir la especificación del protocolo de forma que:

- Cada agente que toma parte en el protocolo se modela como un proceso CSP.
- El intruso más general que puede interactuar con el protocolo también se modela como un proceso CSP.
- El resultado del sistema se comprueba contra las especificaciones que representan las propiedades de seguridad deseables, como "conseguir autentificarse correctamente" o "asegurar la privacidad"; FDR busca el espacio de estados a investigar donde pueda tener lugar trazas inseguras (es decir, secuencias de mensajes).

Casper sólo es una capa superior que simplifica la tediosa tarea de traducir el sistema a CSP, facilitándolo para los protocolos criptográficos y así evitando errores.

FDR opera mediante la enumeración explícita del número de estados posibles en un sistema. Por ello no tiene capacidad para verificar sistemas con infinitos estados y el sistema a representar debe ser un sistema muy concreto y particular. Típicamente será un sistema con un solo emisor y un solo receptor, en vez de un sistema con un número arbitrario de procesos emisor y receptores. Otra restricción impuesta por FDR es que sólo podemos contar con un número finito de tipo de datos atómicos, aunque podamos crear tipos de datos más complejos sobre ellos. El archivo de entrada para Casper tendrá dos partes distintas:

- Una definición del modo en el que el protocolo opera, describiendo los mensajes intercambiados entre los agentes, las comprobaciones que llevan a cabo los agentes, los tipos de datos para los elementos utilizados, el conocimiento inicial de los agentes, y una especificación de qué se supone que se logra con el protocolo.
- Una definición del sistema real que se analiza, definiendo los agentes que toman parte en el sistema y los roles que ejecutan, los tipos de datos reales a utilizar, y las habilidades del intruso.

Estas dos grandes partes se subdividen en secciones. La parte dedicada a la definición del protocolo tiene cinco secciones:

- La sección de variables libres declara los tipos de las variables libres y las funciones que se usan en la definición del protocolo.
- La sección de procesos declara los agentes que toman parte en el protocolo, y da información acerca de su estado inicial.
- La sección de descripción del protocolo define el protocolo, dando la secuencia de mensajes que corresponden a una ejecución normal del protocolo.
- La sección de especificación define los requisitos del protocolo.
- La sección de equivalencias define las propiedades algebraicas de interés de las funciones criptográficas usadas en el protocolo. Esta sección es opcional.

Por otro lado, la parte dedicada a la definición del sistema cuenta con cuatro secciones:

- La sección de variables reales declara los tipos de datos atómicos que se utilizan en el sistema para ser analizado por FDR.
- La sección de funciones define las funciones que se utilizan en el protocolo. Esta sección es opcional.
- La sección del sistema define el sistema que se analiza, en términos de número y tipos de agentes.
- La sección del intruso determina la identidad y el conocimiento inicial del intruso.

1.4. TLS

El Internet Engineering Task Force (IETF) comenzó a desarrollar TLS , basándose en el existente SSL 3.0, con el propósito de desarrollar un protocolo de seguridad sobre redes que se recoge en [?].

El principal objetivo de TLS es permitir que dos aplicaciones establezcan una conexión segura para intercambiar datos a través de una red. Se compone de dos capas: *TLS Record* y *TLS Handshake*.

TLS Record se ejecuta sobre el protocolo de transporte, generalmente TCP,(ver fig. 3. Es el encargado de encapsular los mensajes de las capas superiores. Además proporciona una conexión segura que garantiza dos propiedades

- La conexión es privada. Cada conexión tiene asociada un par de claves simétricas generadas a partir de ciertos dato secretos intercambiados mediante TLS Handshake con las que se cifran los mensajes
- La conexión es fiable. Cada mensaje incluye un campo de verificación de la integridad del mismo utilizando un código de autentificación de mensaje (MAC. Se emplean funciones hash seguras para los cálculos.

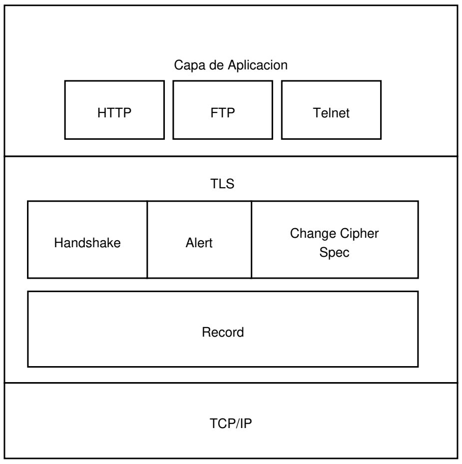


Figura 3: Pila de protocolos con TLS

Por otra parte, TLS Handshake se encarga de autentificar a ambos extremos y negociar un algoritmo de cifrado y claves criptográficas antes de que se transmitan los datos de aplicación. TLS Handshake nos ofrece tres propiedades de seguridad

- La identidad de cada extremo se puede autentificar tanto con clave simétrica como asimétrica. Es opcional aunque suele exigirse al menos a un extremo (el servidor).
- La negociación de los datos secretos es segura. No es posible averiguar el valor de estos datos espiando el flujo de mensajes.
- La negociación es fiable. Ningún atacante podrá modificar el flujo de mensajes sin que sea detectado

Además existen dos subprotocolos más, el protocolo **Alert** y el protocolo **Change Cipher Suite**, que introduciremos en el apartado 6.

1.5. Organización

En este documento resumimos el análisis que hemos llevado a cabo del protocolo TLS mediante la herramienta Casper. En la siguiente sección se introducen las suposiciones que hemos considerado cuando hemos descrito el protocolo para verificarlo. La sección 3 introduce brevemente como se describen las conexiones en TLS. Después hay dos partes, una dedicada a TLS básico y otra dedicada a las extensiones sobre TLS.

En la primera parte las secciones 4 y 5 resumen el funcionamiento de las capas Record y Handshake, así como los resultados de la verificación que hemos realizado sobre ellos. La sección 6 resume las capas Alert y Change Cipher Spec.

Después, en la segunda parte la sección 7 describe el mecanismo general de extensión de TLS y la sección 8 explica las extensiones concretas para redes inalámbricas y la verificación que hemos llevado a cabo sobre ellas.

2. Consideraciones sobre la verificación

El sistema sobre el que vamos a verificar el protocolo puede verse representado en la figura 4. Básicamente tenemos tres tipos de entidades: un servidor TLS, varios clientes TLS y un intruso que intenta alterar la comunicación.

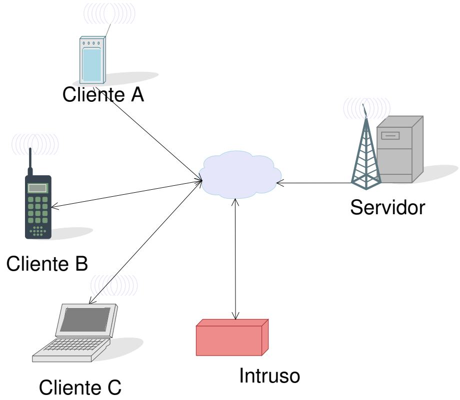


Figura 4: Descripción del sistema donde verificamos el protocolo

Nuestro intruso podrá realizar las siguientes acciones:

- Escuchar e interceptar todos los mensajes de la red.
- Modificar los mensajes añadiendo bytes, borrando bytes o cambiando el valor de los datos.
- Generar nuevos mensajes en base a su conocimiento inicial y el adquirido de los mensajes capturados.
- Reenviar cualquier mensaje nuevo o capturado previamente a otra entidad del sistema.

Sin embargo, no podrá realizar criptoanálisis. Además suponemos que todas las funciones y algoritmos criptográficos son seguros y, por tanto, no podrá corromperlos. Existirán autoridades de certificados (CA) fiables que proporcionaran mecanismos para generar certificados seguros que no pueden ser modificados por el intruso. Todas las variables son atómicas, de forma que el intruso no podrá tener conocimiento parcial de su valor.

3. Estado de la conexión

En TLS tenemos cuatro posibles estados en un determinado momento

- el estado actual de lectura
- el estado actual de escritura
- el estado pendiente de lectura

- el estado pendiente de escritura.

Todos los mensajes se procesan bajo los estados actuales de lectura y escritura. Los parámetros de los estados pendientes se establecen mediante el TLS Handshake. Se puede hacer que los estados pendientes pasen a ser los actuales mediante TLS Change Cipher Spec. Entonces los estados pendientes pasan a ser vacíos.

Los parámetros que configuran cada par de estados lectura/escritura son

Extremo de la conexión. Indica el papel asumido por el extremo: cliente o servidor.

Algoritmo de cifrado simétrico. Se usa para cifrar los mensajes. En concreto se nos ofrecen las siguientes opciones: *null, rc4, rc2, des, triple des (3des), des40.* Además en esta especificación se incluyen varios parámetros más, como el tamaño de las claves de sesión, el tipo de algoritmo de cifrado (por bloques o de flujo),el tamaño del bloque de cifrado si el algoritmo de cifrado es por bloques , y si el algoritmo de cifrado es de tipo "export".

Algoritmo MAC. El algoritmo que usamos para el cálculo del código de autentificación de mensaje y el tamaño de los datos que resultan del cálculo. En concreto solo se nos ofrecen dos opciones *sha* y *md5*

Algoritmo de Compresión. El algoritmo que se usa para comprimir los datos y toda la información asociada al mismo en TLS Record.

Datos secretos (Master Secret). 48 bytes de datos secretos que se intercambian ambos extremos.

Valores aleatorios. Cada extremo (cliente y servidor) generan 32 bytes de datos aleatorios que se llaman *nonce* para cada conexión.

Todos estos parámetros se usan para generar las claves de la sesión y ciertos datos secretos necesarios para el cifrado y el cálculo de los campos MAC, que se describen en el apéndice A.

Una vez que se han establecido los parámetros de seguridad y se han generado las claves, se comienzan a dar valores a los elementos de los estados de conexión y pasando a ser los estados actuales. Cada estado de conexión incluye los siguientes elementos

Estado de compresión. El estado actual del método de compresión.

Estado de cifrado. Es un resumen de la información necesaria para cifrar y descifrar los mensajes. Consistirá en las claves asociadas a la conexión, junto con los vectores de inicialización (IV) si se usa un algoritmo de cifrado por bloques.

MAC_Secret. Datos secretos que se generan para los cálculos MAC.

Número de Secuencia. Cada extremo de la conexión guarda un par de contadores. Uno de los contadores le indica el número de mensaje que espera recibir, y el otro le indica el número de mensaje del próximo mensaje que debe enviar. Cuando se inicia el TLS Handshake ambos contadores comienzan con 0, y cada vez que se envía o recibe un mensaje se actualiza su valor.

4. TLS Handshake

Además del propio TLS Handshake hay dos subprotocolos que permiten al cliente y al servidor negociar los parámetros de sesión. Estos protocolos son el protocolo *Alert* y el protocolo *Change Cipher Spec* (ver sección 6). Pero en esta sección nos centraremos solo en TLS Handshake.

TLS Handshake es el responsable de negociar los elementos que configuran el estado de la sesión: el identificador de sesión, el certificado del participante, un método de compresión, los datos secretos que generan las claves secretas *MasteSecret*, el modo de cifrado *CipherSpec* y un flag que indica si podemos o no restaurar la sesión.

Dentro de TLS Handshake tenemos dos posibles situaciones de partida. La primera es cuando un cliente desea iniciar una nueva sesión con el servidor y debe negociar por primera vez todos los parámetros de sesión. La segunda es cuando ya tenemos una sesión previa y queremos restaurarla.

4.1. Establecer una nueva sesión

La primera opción viene representada por la figura 5. Los mensajes en cursiva son aquellos opcionales y dependen de la situación mientras que los otros son los deben enviarse obligatoriamente.

Cliente		**Servidor**
ClientHello	$\longrightarrow$	
		ServerHello
		Certificate
		ServerKeyExchange
		CertificateRequest
	$\longleftarrow$	ServerHelloDone
Certificate		
ClientKeyExchange		
CertificateVerify		
ChangeCipherSpec		
Finished	$\longrightarrow$	
		ChangeCipherSpec
	$\longleftarrow$	*Finished*
Application Data	$\longleftrightarrow$	**Application Data**

Figura 5: Flujo de mensajes de TLS para establecer una sesión nueva

Inicialmente, el cliente envía al servidor un mensaje *hello* que el servidor debe contestar con su propio mensaje *hello* o con un error fatal y la conexión fallará. Estos mensajes se usan para establecer la combinación más segura entre las posibilidades de ambos participantes. Los mensajes *hello*(ver fig. 6) establecen los siguientes parámetros: la versión del protocolo del emisor, identificador de sesión, *CipherSuite* y método de compresión. Opcionalmente, se generarán dos valores aleatorios que se intercambian: *ClientHello.random* y *ServerHello.random*. En este caso el campo de identificador de sesión en el mensaje del cliente estará vacío y el servidor tendrá que generar uno nuevo.

Después del intercambio de parámetros de sesión el servidor envía su certificado si es necesario autentificarlo. En el mensaje *Certificate* el servidor envía una cadena de certificados desde su propio certificado hasta el certificado de la autoridad de certificados raíz del árbol de certificados correspondiente. Estos certificados suelen seguir el formato X.509v3 descrito en [?]. Junto con el mensaje *Certificate* el servidor puede enviar un mensaje *ServerKeyExchange* si su tipo de certificado lo exige (ver fig. 7). Tras la autentificación del servidor, este podría pedirle al cliente que se autentifique mediante un mensaje de petición de certificado *CertificateRequest* (ver fig. 8), si la *CipherSuite* seleccionada lo indica. En la petición de certificado se incluye una lista de los tipos de certificados que son admisibles y una lista de las autoridades de certificados que se consideran fiables por parte del servidor. Y una vez enviada la petición de certificado el servidor envía su mensaje *ServerHelloDone* (ver fig. 6) para indicar que está listo para comenzar la siguiente parte del Handshake. Cuando el cliente recibe este mensaje sabe que debe verificar el certificado del servidor y comenzar con el intercambio de datos secretos.

Además de la autentificación, también se realiza el intercambio de los datos necesarios para la generación segura de claves. Para ello usamos cuatro mensajes: *Certificate* del servidor, *ServerKeyExchange* si se envía, *Certificate* del cliente y *ClientKeyExchange*. Si se desea añadir un nuevo método de intercambio de claves se debe alterar el formato de estos mensajes. Y la cantidad de datos secretos intercambiados deben ser suficientemente grande como para garantizar la seguridad de las claves. Actualmente se usan entre 48 y 128 bytes.

Generalmente el intercambio de datos secretos comienza con el mensaje que contiene el certificado del cliente, si el cliente ha recibido un mensaje *CertificateRequest*. Después se envía el mensaje *ClientKeyExchange* cuyo formato varía dependiendo del tipo de algoritmo de intercambio de claves seleccionado (ver fig. 8). Por último, si el cliente ha enviado un certificado con posibilidad de realizar firmas digitales además se envía otro mensaje *CertificateVerify* (ver fig. 8) que reafirma su identidad. En este mensaje se genera una firma digital sobre un dato llamado *mensajes_handshake*, que son todos los mensajes intercambiados entre el cliente y el servidor hasta ese punto incluyendo los campos de longitud y tipo de mensaje handshake.

En este punto se considera que la negociación de parámetros ha finalizado y se intercambian los mensajes *ChangeCipherSpec*. Entonces el cliente envía su mensaje *Finish*(ver fig. 9) y como respuesta el servidor le envía otro mensaje de este tipo. Este mensaje tiene el mismo formato tanto para el cliente como el servidor. Lo único que los diferencia es que al generar los datos de verificación se usa como etiqueta *ClientFinish* si lo envía el cliente o *ServerFinish* si lo envía el servidor. Después de estos mensajes el protocolo TLS Handshake ha finalizado y comienza el intercambio de datos de aplicación.

4.2. Verificación del protocolo para establecer una nueva sesión

Para verificar este protocolo hemos adoptado algunas simplificaciones (ver fig. 10). En principio no son necesarias pero como añadiremos a esta traza las extensiones y eso complicará mucho el protocolo, hemos decidido incluirlas desde el principio. La primera es que se han omitido los mensajes de ChangeCipherSpec y ServerHelloDone, ya que son solo constantes que no aportan ningún elemento

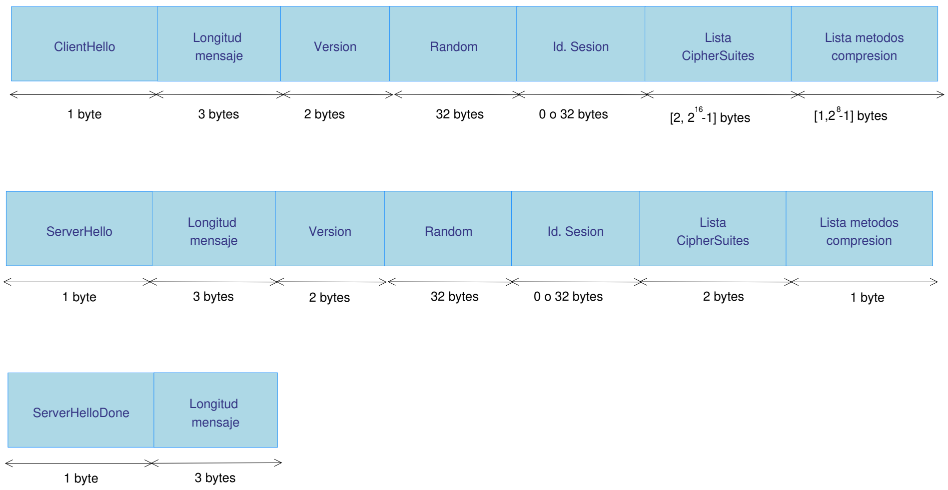


Figura 6: Formato de los mensajes *Hello*

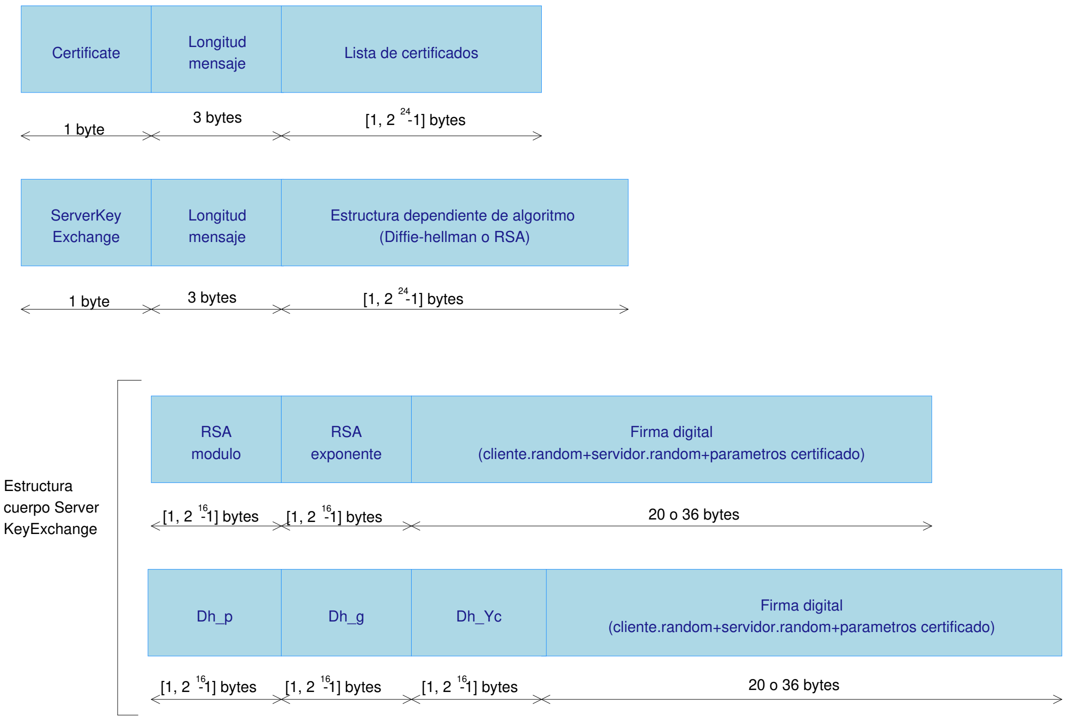


Figura 7: Formato de los mensajes de autentificación del servidor

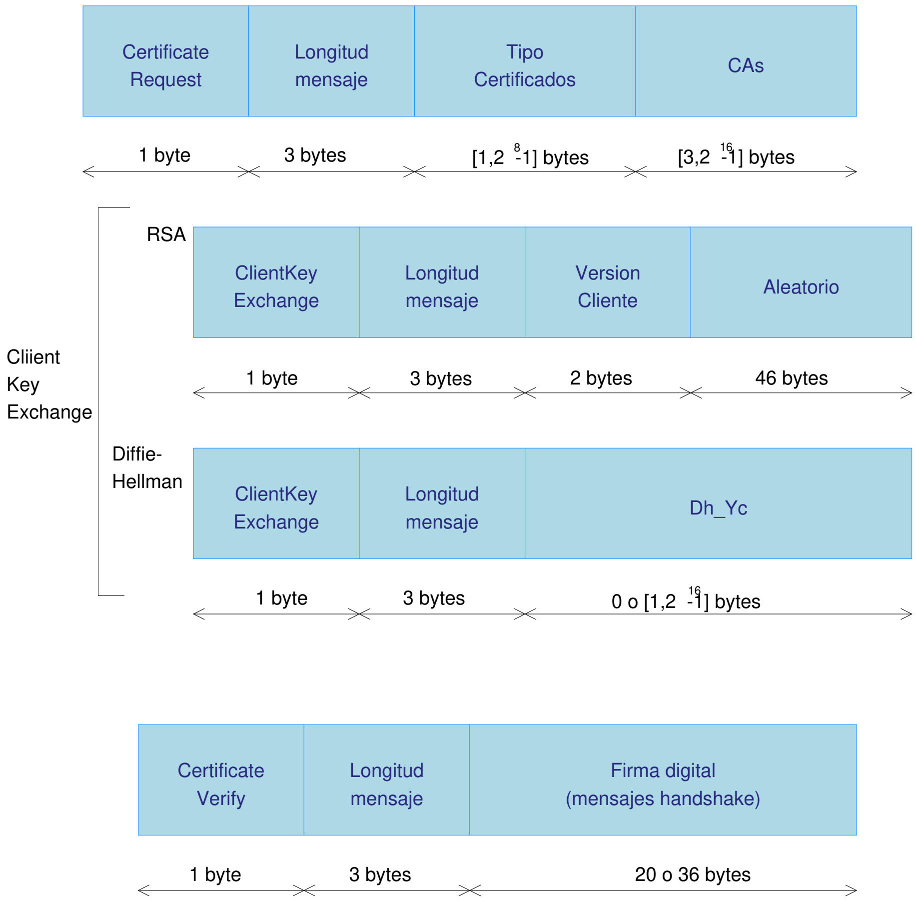


Figura 8: Formato de los mensajes de autentificación del cliente e intercambio de datos secretos

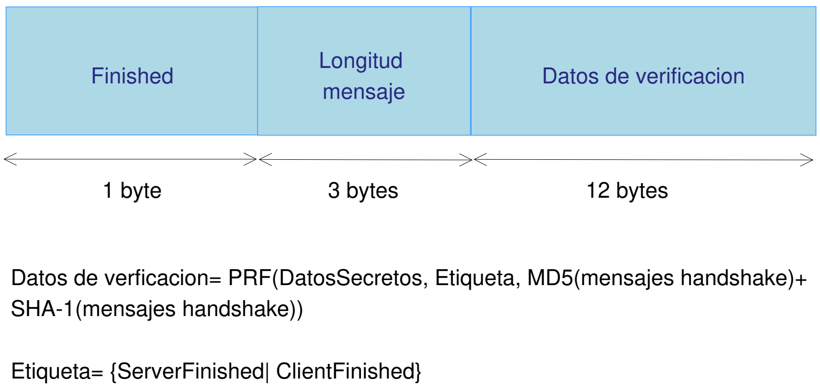
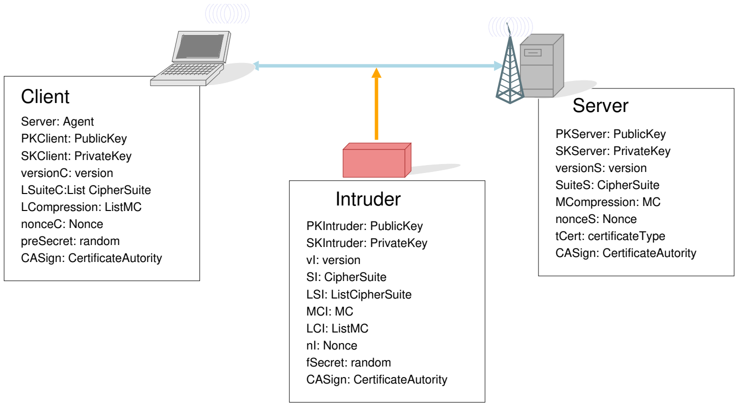


Figura 9: Formato de los mensajes *Finish*

importante para establecer la sesión y solo indican que se puede realizar ciertos procedimientos. Después, hemos supuesto que el identificador de sesión y el nonce del servidor reciben un tratamiento similar y los podemos agrupar en una única variable de forma que si recibe un ataque lo reciben ambos datos. Así se reduce el espacio de estados que la herramienta tiene que verificar.

Por otra parte, representamos los certificados como una lista con dos elementos: el identificador del propietario del certificado y la clave pública asociada (ver fig. 10). Están firmados por una clave simétrica que suponemos que es de la Autoridad de Certificados. Como se suponen que son fiables y no pueden ser alterados, se comprueban con una función *ValidCertificate*, que devuelve false si ha sido alterado y finaliza la ejecución. De esta forma evitamos tener que introducir otro agente que represente a la autoridad de certificados.

Por último, sólo indicar que el proceso de cálculo de las claves de sesión está representada por las funciones CWS para el cliente y SWS para el servidor.



Protocol Description:

```
ClientHello.  C -> S : C, verC, NonceC, SuiteC, LCompression
[C!=S]
ServerHello.  S -> C : verS, NonceS, SuiteS, MCompression
CertificateS.  S -> C : {S,PKServer}{CASign}
[ValidCertificate(S,PKServer)]
CertificateRequest. S -> C : tCert
CertificateC.  C -> S : {C, PKClient}{CASign}
[ValidCertificate(C,PKClient)]
ClientKeyExchange. C -> S : {preSecret}{PKServer}
CertificateVerify.  C -> S : {MD5(preSecret, NonceC,NonceS)}{SKClient}
FinishedC1.   C -> S : {MD5(SuiteC, verC, LCompression)}{CWS(preSecret, NonceC, NonceS)}
FinishedC2.  C -> S : {MD5(SuiteS, verS, MCompression)}{CWS(preSecret, NonceC, NonceS)}
FinishedS1.  S -> C : {MD5(SuiteC, verC, LCompression)}{SWS(preSecret, NonceC, NonceS)}
FinishedS2.  S -> C : {MD5(SuiteS, verS, MCompression)}{SWS(preSecret, NonceC, NonceS)}
```

Figura 10: Representación del sistema y descripción del protocolo para restaurar una sesión

Las propiedades que le pedimos a Casper que verifique son:

A. `Secret(C, preSecret,[S])`

B. `Agreement(C,S, [preSecret, SuiteS, MCompression, PKServer, NonceC, NonceS, verS, verC])`

C. `Agreement(S,C, [preSecret, SuiteS, MCompression, PKServer, NonceC, NonceS, verS, verC])`

La primera propiedad nos permite comprobar si el intercambio de datos secretos es seguro, el intruso no logra conocer su valor y las claves de sesión son realmente secretas.

Por otra parte, las dos últimas propiedades indican si el cliente ejecuta con éxito el protocolo con el servidor y ambos conocen el mismo valor para las variables de la lista en la definición de la propiedad. De esta forma comprobamos que la autentificación ha sido correcta, el intruso no suplanta a ningún agente, y que no consigue modificar la información intercambiada entre el cliente y el servidor. Casper no ha encontrado ningún ataque sobre esta parte del protocolo.

4.3. Restaurar una sesión previa

Al principio decíamos que existían dos situaciones iniciales posibles. Ya hemos explicado el proceso para establecer una nueva sesión. Ahora introduciremos los pasos que se siguen para restaurar una sesión previa, representados en la figura 11.

Cliente		Servidor
ClientHello	$\longrightarrow$	
		ServerHello
		ChangeCipherSpec
	$\longleftarrow$	Finished
ChangeCipherSpec		
Finished	$\longrightarrow$	
Application Data	$\longleftrightarrow$	**Application Data**

Figura 11: Flujo de mensajes de TLS para restaurar una sesión previa

En este caso el cliente incluye el identificador de la sesión previa en el mensaje *ClientHello*. Cuando el servidor lo recibe comprueba que este identificador esté en su caché. Si lo encuentra y el servidor está en condiciones de restablecer la sesión bajo la configuración previa enviará un mensaje *ServerHello* con el mismo identificador de sesión. Entonces intercambian mensajes *Finished* y mensajes *ChangeCipherSpec*. Así se completa la restauración de sesión y ambos extremos comienzan a intercambiar datos de aplicación. Por otra parte, si el servidor no encontrará el identificador de sesión en su caché se tendría que ejecutar el protocolo handshake completo como está en la figura 5. En este caso el servidor contestaría con un mensaje *ServerHello* con un nuevo identificador de sesión distinto al que envió el cliente.

4.4. Verificación del protocolo para restaurar una sesión previa

En este caso, si incluimos el identificador de sesión porque se trata de una variable con un tratamiento diferente al del nonce del servidor. En la representación del protocolo (ver fig. 12) se han vuelto a omitir los mensajes del protocolo Change Cipher Spec.

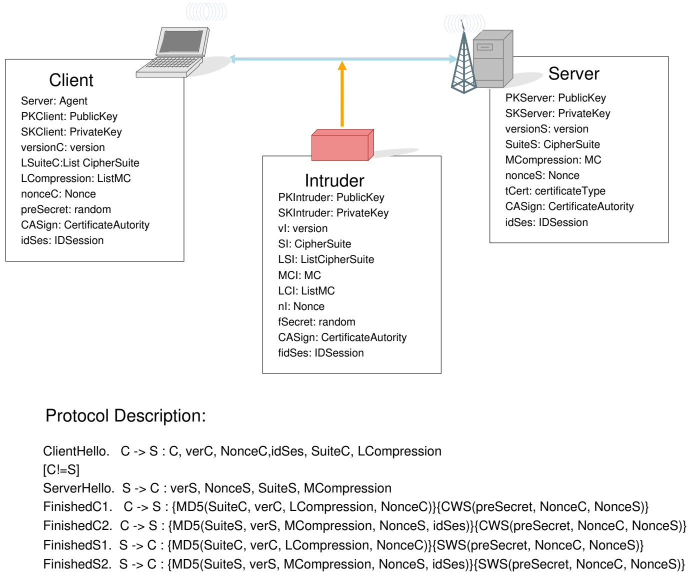


Figura 12: Representación del sistema y descripción del protocolo para restaurar una sesión

Comprobamos que se cumplen las siguientes propiedades:

A. `Agreement(C,S, [preSecret, SuiteS, MCompression, PKServer, NonceC, NonceS, verS, verC])`

B. `Agreement(S,C, [preSecret, SuiteS, MCompression, PKServer, NonceC, NonceS, verS, verC])`

Como no tiene lugar intercambio de secretos, solo verificamos que el intruso no suplante a ningún agente ni altere ningún mensaje. Casper no encuentra ataques en esta traza.

5. TLS Record

TLS Record es un protocolo formado por diversas capas. En cada capa, se incluye en los mensajes campos descriptivos, de tamaño del mensaje y el propio contenido del mensaje. El proceso es simple

- Para enviar datos se dividen en bloques de un tamaño predeterminado, se comprimen, se calcula un MAC, se cifran y se transmite el resultado.
- Cuando se reciben datos se descifran, se verifica el mensaje con el MAC, se descomprimen y se ensamblan para pasarlos a las capas superiores.

5.1. Funcionamiento de TLS Record

Primero, TLS Record recibe datos de las capas superiores y divide los datos de aplicación en fragmentos de 2^{14} bytes. Incluso se mezclan datos de diferentes aplicaciones pero del mismo tipo para llegar al tamaño mínimo. Cada fragmento se le da formato como *records* y se incluye en un mensaje *TLSPlainText*(ver fig. 13) y hay cuatro tipos de datos: TLS Handshake, TLS Alert, TLS Change Cipher Spec y datos de aplicación.

Después, todos los records son comprimidos y descomprimidos con el método negociado en el Handshake. Cuando comprimimos los datos de aplicación pasamos de un record del tipo *TLSPlainText* a uno *TLSCompressed*(ver fig. 13). Ambos records tienen el mismo valor para el campo tipo y para el campo versión del protocolo.

Por último, se protegen los datos del record pasando de un record *TLSCompressed* a un record *TLSCipherText*. El cuerpo del record *TLSCipherText* depende de si usamos un algoritmo de cifrado por bloques o uno de flujo, pero en ambos casos, sobre los datos de aplicación comprimidos se calcula el MAC y se cifra el contenido.

El cálculo del MAC se realiza antes de cifrar el mensaje, mediante la siguiente función:

$$\begin{aligned} &HMAC\_hash(MAC\_secreto\_escritura, numero\_secuencia + \\ &\quad TLSCompressed.Tipo + TLSCompressed.Version + \\ &\quad TLSCompressed.Longitud + TLSCompressed.Datos) \end{aligned} \tag{1}$$

En caso de usar un algoritmo de cifrado por bloques, además de los datos y el MAC, se añade un campo de relleno *padding* y un campo que indica la longitud del campo anterior. Este campo de relleno se inserta para que el tamaño de los datos sea múltiplo del tamaño de bloque. Se rellena repitiendo el valor del campo de longitud de padding. Por ejemplo, si necesitamos 6 bytes de padding, el campo padding será ”06 06 06 06 06 06”.

5.2. Verificación de TLS Record

Para verificar TLS Record partimos del momento en que se ha realizado con éxito el TLS Handshake y cada extremo ha calculado ya las claves de sesión asociadas. Luego, representaremos las claves de sesión como constantes. Además, suponemos que los datos de aplicación que intercambiamos, *msg1* y *msg2*, son datos ya fragmentados y comprimidos. Solo intercambiamos records del tipo *TLSCipherText* para un algoritmo de cifrado de bloques. Por eso incluimos el campo de padding con su longitud. Las variables tipo *header* agrupan a todos los campos del encabezado del record para simplificar: el tipo de mensaje, la versión y la longitud del mensaje.

Básicamente la especificación se corresponde con el intercambio de dos mensajes representado en la figura 14.

Sobre este protocolo verificamos las siguientes dos propiedades:

A. `Secret(C, msg1,[S])`

B. `Secret(S, msg2,[C])`

Ambas nos permiten comprobar que los datos de aplicación intercambiados son secretos y no han sido modificados por el camino. Al verificar estas propiedades en Casper vemos que se cumplen y no se encuentra ningún ataque.

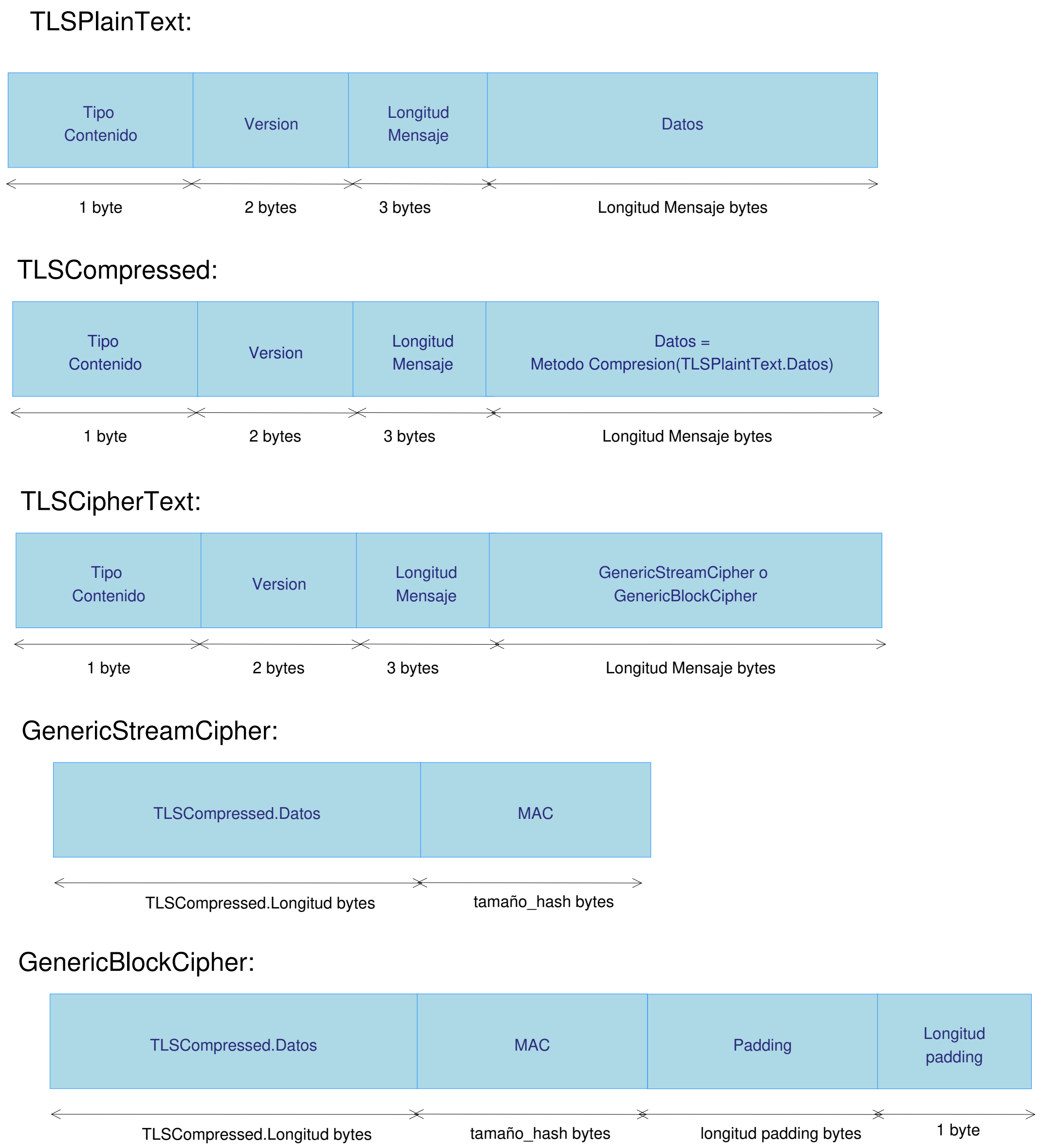
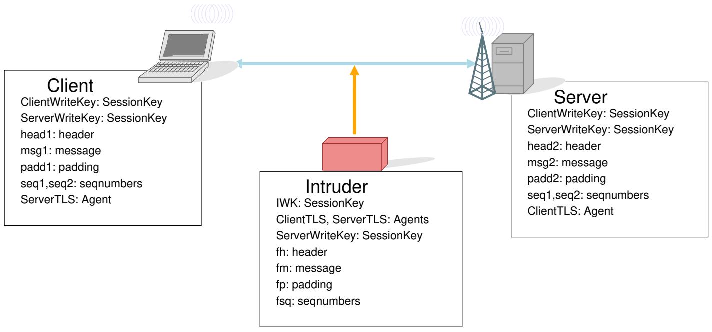


Figura 13: Formatos de los mensajes de TLS Record



Protocol Description:

RecordCS1. C -> S : head1, {msg1, padd1}{ClientWriteKey}
RecordCS2. C -> S :{mac(ClientWriteKey,seq1,seq2, head1,msg1)}{ClientWriteKey}
RecordSC1. C -> S : head2, {msg2, padd2}{serverWriteKey}
RecordSC2. C -> S :{mac(ServerWriteKey,seq1,seq2, head2,msg2)}{ServerWriteKey}

Figura 14: Representación del sistema y descripción del protocolo para verificar TLS Record

6. Otros mensajes

6.1. Protocolo Change Cipher Spec

Simplemente es un protocolo que indica el cambio entre estrategias de cifrado. Consiste en un único mensaje (ver fig. 15) que recibe el mismo nombre que el protocolo que es cifrado y comprimido bajo el estado de conexión actual, no el pendiente.

Lo envían tanto el cliente como el servidor cuando desean que los records que se envían a partir de ese momento se cifren bajo la configuración recién negociada. El receptor del mensaje debe copiar la configuración pendiente a la configuración actual.

Se envía junto después de la negociación del TLS Handshake pero antes de enviar los mensajes *finished*.

6.2. Protocolo Alert

Los mensajes del protocolo *Alert* sirven para informar de un error. Son mensajes que se cifran y comprimen con la configuración actual. Y cada mensaje consta de dos campos: el nivel del error y un campo constante con una descripción (ver fig. 15). En el apéndice B se encuentra un listado de los tipos de errores y una breve descripción de los mismos.

Cada tipo de error tiene asociado un nivel de importancia. Dentro del nivel de importancia tenemos dos posibilidades:

1. *fatal*, que son errores irrecuperables y conllevan la finalización de la conexión.

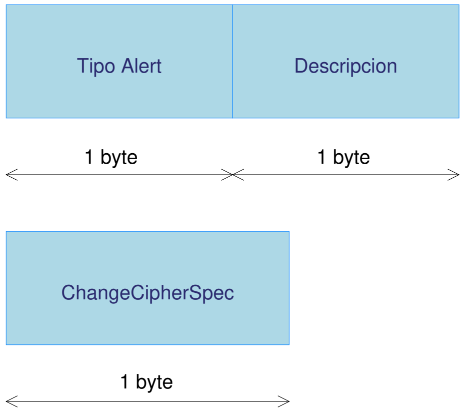


Figura 15: Mensajes de los protocolos *ChangeCipherSpec* y *Alert*

2. *warning*, son errores menores, que permiten que la sesión continue, pero el identificador de sesión se invalida, para evitar que se establezcan más sesiones que conlleven otros errores.

Sólo destacar el mensaje *Alert* del tipo *close_notify*. Este mensaje lo usan ambos extremos de una conexión para indicar que finalizan una sesión y evitar cualquier posible ataque. Cualquier parte debe enviar un mensaje de este tipo antes de cerrar el lado de escritura de la conexión, y el receptor debe contestar con un mensaje *close_notify* y cerrar la conexión, descartando cualquier escritura pendiente.

7. Mecanismo general de extensión

TLS ha sido dotado de mecanismos de extensión para adaptarse a los diversos entornos en los que puede operar [?]. Estos mecanismo son genéricos y a partir de ellos podemos obtener extensiones especificas. De hecho, en [?], ya vienen definidas unas extensiones específicas para redes inalámbricas.

Las extensiones se refieren a modificaciones que afectan al formato de ciertos mensajes, ya que la definición propia de TLS está preparada para añadir nuevas *CipherSuites*.

Los mecanismos generales de extensión de TLS simplemente contemplan el modificar los mensajes Hello del protocolo Handshake para permitir a ambas entidades negociar el uso de extensiones específicas. Tanto en el mensaje *ClientHello* como en el mensaje *ServerHello* se modifica el formato original del mensaje y añade un nuevo campo llamado *Lista extensiones* de hasta $2^{16} - 1$ bits (ver fig. 16).

La lista de extensiones está compuesta por elementos *extension*. Cada elemento *extension* está compuesto por un nombre que define el tipo de extensión y unos parámetros para negociar las condiciones de la extensión. Para diferenciar los mensajes *hello* normales a los que llevan extensión nos referiremos a estos últimos como *hello extendidos*.

El funcionamiento de las extensiones es muy sencillo. El cliente realiza una

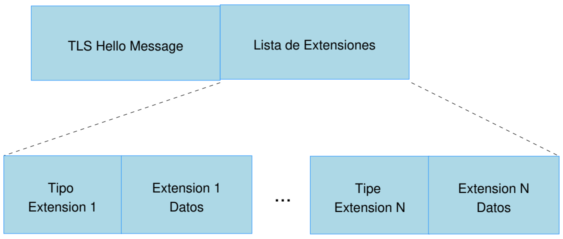


Figura 16: Representación de la extension de los mensajes hello

petición de algún tipo de funcionalidad extraordinaria enviando un mensaje *hello extendido* al servidor. Si el servidor no soportara la petición del cliente, éste podría decidir terminar la ejecución del handshake. Si el servidor la soporta, debe contestarle al cliente con su correspondiente mensaje *hello* extendido. Un cliente puede solicitar tantas extensiones como pueda especificar dentro de los $2^{16} - 1$ bits del campo aunque no puede repetir varias extensiones del mismo tipo.

Un cliente finalizará la ejecución si recibe un mensaje *hello* extendido que contiene o bien un tipo de extensión que el no ha pedido o bien incluye una extensión cuando el cliente no ha solicitado ningún tipo de funcionalidad adicional. De momento solo se han considerado extensiones solicitadas por el cliente.

Por último indicar que se incrementan los tipos de mensajes Alert para dar cobertura a los posibles errores referidos a extensiones. Para ver estos nuevos mensajes de error ver el apendice C.

En la verificación nos centraremos sólo en las extensiones sobre redes inalámbricas, que conllevan otras limitaciones además de las propias de los medios cableados. Estas limitaciones son principalmente la capacidad de computación de los dispositivos, reducido espacio de memoria, la duración de batería y limitaciones en el ancho de banda. Los posibles tipos de extensiones para redes inalámbricas que se han especificado vienen descritos en el cuadro 1.

Tipo de Extensión	Código	Uso
server_ name	0	Indicar el nombre del servidor en redes virtuales
max_fragment_lenght	1	Establecer un tamaño máximo de mensaje
client_certificate_url	2	Usar una URL de certificado en vez el certificado
trusted_ca_keys	3	Indicar al servidor cual es la raíz de CA del cliente
truncated_hmac	4	Usar MACs truncadas
status_request	5	El servidor envía información a cerca del estado del certificado del cliente

Cuadro 1: Tabla resumen de las extensiones específicas consideradas

8. Extensiones Específicas para Wireless

8.1. URLs de Certificados del cliente

Por cuestiones de espacio de memoria, si un cliente debe autentificarse podría preferir enviarle una dirección URL a un servidor donde se encuentra su certificado. De esta forma el cliente no tiene que almacenar el certificado completo de forma local. Para ello envía un mensaje *hello* con la extensión *client_certificate_url* y el campo de datos vacío.

Una vez el servidor recibe este mensaje *hello* extendido indicará al cliente que espera su certificado enviando el correspondiente mensaje *hello* extendido incluyendo el mismo tipo de extensión.

Así queda negociado el uso de URL de certificados, y el cliente envía un mensaje *CertificateURL* (ver fig. 17) en lugar de un mensaje *Certificate* normal.

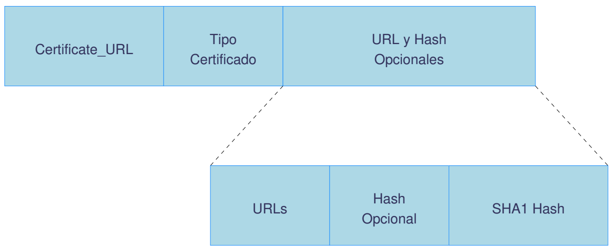


Figura 17: Mensaje CertificateURL para la extensión de uso de URL de certificados

El campo *URL y Hash opcional* contiene una lista de URLs y una lista opcional de hashes. Si *hash presente* es true entonces se incluye la lista de hashes, si por el contrario valiera false esta lista estaría vacía.

El campo *Tipo Certificado* puede tener dos valores: *individual_certs* ó *pkipath*. Si usamos certificados X.509 nos podemos encontrar con dos posibilidades:

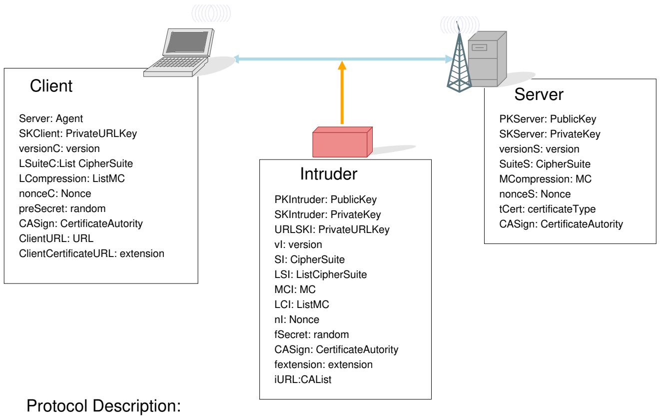
1. Cada URL se refiere a un solo certificado DER-encoded X.509v3 con una dirección al certificado del cliente primero (caso del tipo *individual_certs*).

2. La lista de URLs contiene una única URL que se refiere a la cadena de certificados DER-encoded usando el tipo *pkipath*.

Sea cual sea el formato, cuando el servidor recibe el mensaje debe recuperar la cadena de certificados y comprobarlos de forma análoga a como lo hacia cuando recibía un mensaje *Certificate*.

Un cliente puede seleccionar enviar o un mensaje *Certificate URL* o un mensaje *Certificate* después de negociar con éxito el uso de URL de certificados. Así se permite que el cliente posea múltiples certificados.

Para verificar esta extensión (ver fig. 18), consideramos las URLs como una variable que se envía en lugar de los certificados en el mensaje *Certificate*. Cuando el servidor la recibe ejecuta la función *CheckURLCorrect* que asocia cada URL con un agente. Si la URL se corresponde con el cliente entonces devolverá *true* y de esta forma indicamos que el servidor ha recuperado correctamente la cadena de certificados.



Protocol Description:

ClientHello. C -> S : C, verC, NonceC, SuiteC, LCompression, ClientCertificateURL
[C!=S]
ServerHello. S -> C : verS, NonceS, SuiteS, MCompression, ClientCertificateURL
CertificateS. S -> C : {S,PKServer}{CASign}
[ValidCertificate(S,PKServer)]
CertificateRequest. S -> C : tCert
CertificateC. C -> S : {ClientURL}{CASign}
[CheckURLCorrect(C,ClientURL)]
ClientKeyExchange. C -> S : {preSecret}{PKServer}
CertificateVerify. C -> S : {MD5(preSecret, NonceC,NonceS)}{SKClient}
FinishedC1. C -> S : {MD5(SuiteC, verC, LCompression)}{CWS(preSecret, NonceC, NonceS)}
FinishedC2. C -> S : {MD5(SuiteS, verS, MCompression, ClientCertificateURL)}{CWS(preSecret, NonceC, NonceS)}
FinishedS1. S -> C : {MD5(SuiteC, verC, LCompression)}{SWS(preSecret, NonceC, NonceS)}
FinishedS2. S -> C : {MD5(SuiteS, verS, MCompression, ClientCertificateURL)}{SWS(preSecret, NonceC, NonceS)}

Figura 18: Representación del sistema y descripción del protocolo para la extensión para URLs de Certificados

Sobre esta versión verificamos las mismas tres propiedades de seguridad que en la sección de inicio de nueva sesión (ver 4.2). Aquí modificamos levemente la propiedad B y añadimos que se verifique el tipo de extensión. Casper no encuentra ataques en el protocolo.

8.2. Indicación de CA fiables

Dado que los clientes tienen limitaciones de memoria, sólo pueden poseer un limitado número de CA fiables y sería recomendable que pudieran indicarle cuales poseen al servidor con el fin de evitar fallos en el Handshake. Para ello se usa un mensaje *hello* extendido con el tipo *trusted_ca_authorithy* y una lista de las CA fiables que posee el cliente (ver fig. 19). Por cada CA fiable se añade un registro a la lista que depende del tipo de autoridad de certificados.

Los formatos de los registros son:

- *pre_agreed*: no se indica ningún tipo de identidad de la CA raíz de la clave.
- *key_sha1_hash*: contiene el hash SHA-1 de la clave de la CA raíz. Depende de si usamos RSA o DSA.
- *x509_nombre*: contiene el DER-encoded X.509 DistinguishedName de la CA.
- *cert_sha1_hahs*: contiene el hash SHA-1 de un certificado DER-encoded que contiene la clave de la CA raíz.

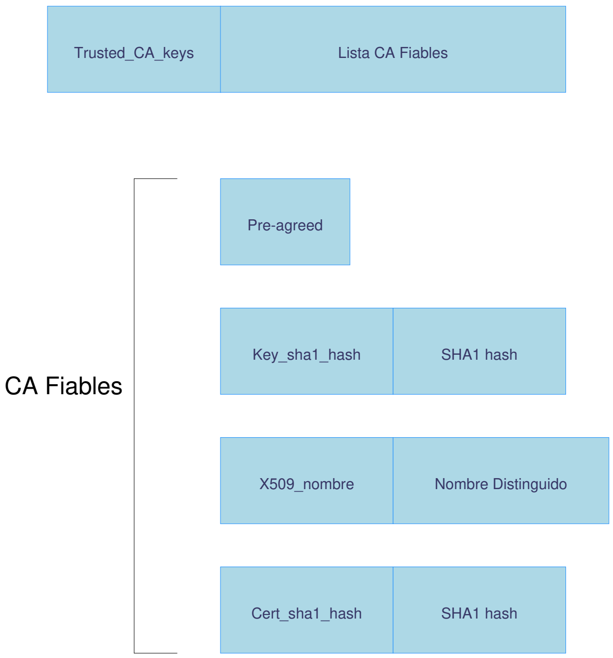


Figura 19: Extensión para indicarle al servidor CA fiables

Un cliente puede incluir alguna clave de raíces de certificados, todas las que posee o ninguna.

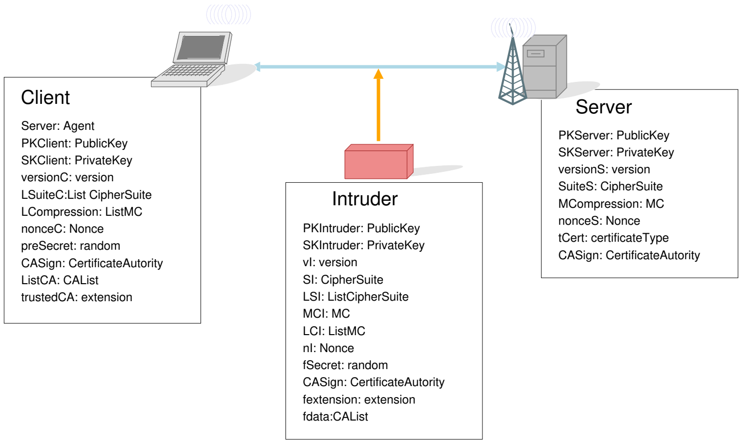
Cuando un servidor recibe un mensaje *hello* extendido con la lista de CA usará esta información para seleccionar la cadena de certificados apropiada para enviarle al cliente. Y cuando envíe su mensaje *hello* extendido enviará una extensión de este tipo aunque con el campo de datos vacío.

Para verificar esta extensión hemos modificado muy poco la descripción del protocolo para establecer una nueva sesión (ver fig. 20). Simplemente hemos especificado los mensajes hello extendidos y en el del cliente además del tipo de extensión hemos incluido una variable que representa una lista de CA fiables para el cliente. Suponemos que una vez que recibe el mensaje el servidor lo procesa adecuadamente y selecciona la CA más conveniente para ambos.

Hemos verificado como siempre las tres principales propiedades de seguridad que ya hemos visto con anterioridad (ver apartado 4.2). Y al igual que antes hemos ampliado la lista de variables de la propiedad B con aquellas relacionadas con la extensión, en este caso concreto la lista de CAs del cliente. Casper no ha encontrado errores.

8.3. Petición del estado del certificado

Algunos clientes, debido a sus limitaciones, prefieren usar protocolos que les ayuden a verificar el estado y la validez del certificado que reciben de un servidor, así se evitan recuperar y verificar una cadena de certificados no válidos y ahorran ancho de banda. TLS incluye una extensión que permite comprobar el estado de un certificado, ahorrando recursos y rodeos.



Protocol Description:

ClientHello. C -> S : C, verC, NonceC, SuiteC, LCompression, trustedCA , ListCA
[C!=S]
ServerHello. S -> C : verS, NonceS, SuiteS, MCompression, trustedCA
CertificateS. S -> C : {S,PKServer}{CASign}
[ValidCertificate(S,PKServer)]
CertificateRequest. S -> C : tCert
CertificateC. C -> S : {C, PKClient}{CASign}
[ValidCertificate(C,PKClient)]
ClientKeyExchange. C -> S : {preSecret}{PKServer}
CertificateVerify. C -> S : {MD5(preSecret, NonceC,NonceS)}{SKClient}
FinishedC1. C -> S : {MD5(SuiteC, verC, LCompression)}{CWS(preSecret, NonceC, NonceS)}
FinishedC2. C -> S : {MD5(SuiteS, verS, MCompression, ListCA)}{CWS(preSecret, NonceC, NonceS)}
FinishedS1. S -> C : {MD5(SuiteC, verC, LCompression)}{SWS(preSecret, NonceC, NonceS)}
FinishedS2. S -> C : {MD5(SuiteS, verS, MCompression, ListCA)}{SWS(preSecret, NonceC, NonceS)}

Figura 20: Representación del sistema y descripción del protocolo para verificar la extensión para CA Fiables

Para ello el cliente envía un mensaje *hello* extendido con el tipo *status_request* y en el campo de datos un campo de petición del estado de un certificado (ver fig. 21).

Sólo tenemos para esta extensión un tipo de petición que es *ocsp*. En el cuerpo de la petición tenemos dos campos. El primero incluye una lista de los identificadores de aquellas entidades que van a contestar a la petición y en las que el cliente confía. Si la lista es vacía es porque estas entidades se consideran implícitas. Por otra parte, el campo de petición de extensiones proviene de [?] y si no hay peticiones es vacío. La petición de OCSP contiene la versión del protocolo, la petición de servicio, identificador del certificado a verificar y ciertas extensiones opcionales.

Los servidores que reciben un mensaje con una petición del estado del certificado deben responder al cliente con un certificado de su estado adecuado. Si se pide OCSP, deben usar la información contenida en la extensión cuando seleccionan una respuesta OCSP y deben incluir las peticiones de extensión en la petición OCSP.

El servidor enviará un mensaje de respuesta *CertificateStatus* (ver fig. 21) después de enviar su mensaje *Certificate*. Además cuando envíe su mensaje *hello* será extendido con este tipo de extensión y el campo de datos de extensión vacío. Un servidor puede decidir no responder a una petición de comprobación del

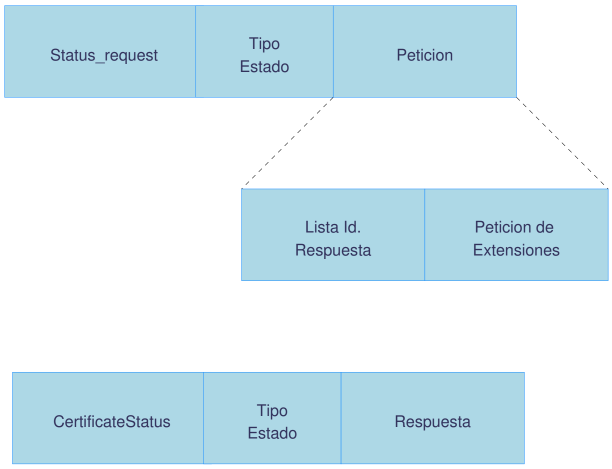


Figura 21: Extensión para la comprobación del estado del certificado

estado de su certificado, pero no puede enviar un mensaje *CertificateStatus* sin una petición del cliente. El campo respuesta generalmente será del tipo OCSP según se especifica en [**?**].

Una respuesta OCSP contiene los siguientes datos:

- Versión de la sintaxis de respuesta.
- Nombre del agente que responde.
- Respuesta compuesta por: el identificador del certificado objetivo, valor del estado del certificado (valido, revocado o desconocido), intervalo de validez y extensiones.
- Extensiones.
- Firma digital calculada siguiendo la siguiente fórmula:

$$hash(version, identificador\ del\ agente, intervalo\ de\ validez, respuesta, extensiones) \tag{2}$$

Cuando el cliente recibe la respuesta del servidor la comprueba y si no le satisface puede abortar la ejecución del handshake.

Para verificar la extensión supondremos que cuando el cliente realiza la petición del estado del certificado no incluye ninguna lista de entidades de las que espera respuesta porque solo le responderá el servidor y no pide ninguna extensión (ver fig. 22). En cuando al mensaje de *CertificateStatus* del servidor solo se incluyen los campos más importantes: el estado del certificado, el intervalo de validez, el identificador del agente que envía la respuesta y la firma digital de los datos anteriores. Cuando el cliente recibe este mensaje ejecuta la función *Verify* que mira si el estado y el intervalo de validez son correctos y entonces se acepta el certificado. Si no fueran valores correctos se finalizaría la ejecución.

Hemos verificado como siempre las tres principales propiedades de seguridad que ya hemos visto con anterioridad (ver apartado 4.2). Y al igual que antes hemos ampliado la lista de variables de la propiedad B con aquellas relacionadas con la extensión. Casper no ha encontrado errores.

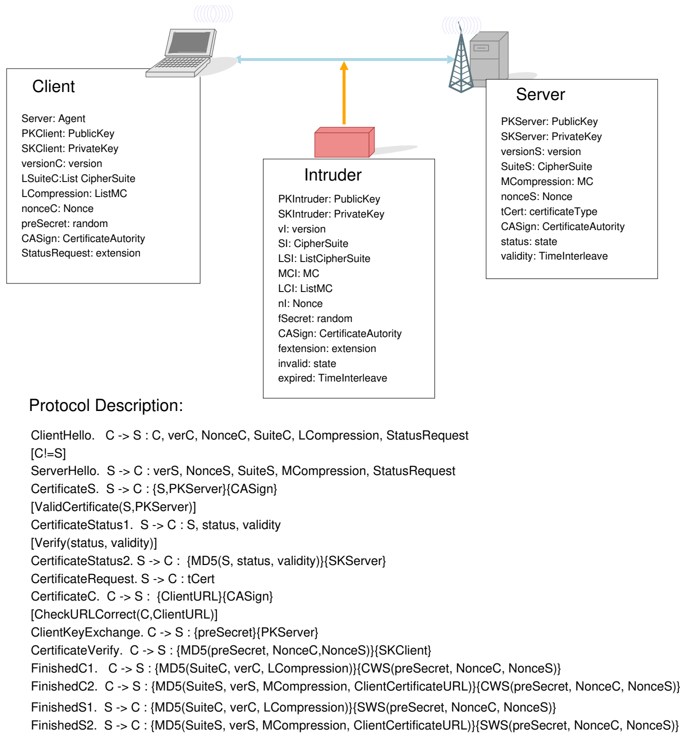


Figura 22: Representación del sistema y descripción del protocolo para verificar la extensión de estado del certificado

9. Extensiones Especificas sin verificar

9.1. Indicación del nombre del servidor

En ocasiones es conveniente poder indicar el nombre del servidor al que un cliente se quiere conectar, para poder usar TLS en redes virtuales sobre una misma dirección de red.

Para indicar el nombre del servidor al que nuestro cliente TLS quiere conectarse debe indicar una extensión del tipo *server_name* junto con un campo *ListaNombreServidores*(ver fig. 23). Esta lista de nombres de servidores puede estar vacía o contener una tupla formada por el tipo de nombre y el nombre del servidor en sí.

En la actualidad, el único tipo de nombre que se ha considerado son los nombres DNS. Pero se pueden añadir otros tipos de nombres de servidores si es necesario en un futuro. En este caso concreto la sintaxis del nombre del servidor debe adaptarse a la convención de los nombre en DNS.

Cuando un servidor recibe un mensaje *hello* extendido con este tipo de petición, puede usar la información contenida en la extensión como guía para seleccionar un certificado adecuado que devolver al cliente o/y aplicar ciertas posibles políticas de seguridad. En su mensaje *hello* incluirá una extensión de este tipo pero sin una lista de nombres de servidores. Si el servidor comprende la extensión pero no reconoce el nombre del servidor enviará un mensaje de error

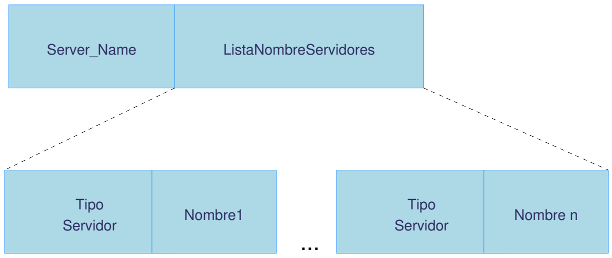


Figura 23: Extensión para indicar el nombre del servidor

del tipo *unrecognized_name*.

Si una aplicación negocia el nombre de un servidor usando un protocolo de la capa de aplicación, entonces TLS envía una extensión del tipo *server_name* con el nombre que le da la capa de aplicación. Si se negocia un determinado nombre con TLS Handshake para una sesión, el cliente no debe intentar pedir un nombre de servidor diferente al que le proporciona la capa de aplicación.

9.2. Tamaño máximo del fragmento

Por regla general, todos los mensajes solo pueden contener 2^{14} bytes de datos de aplicación sin cifrar. Pero dadas ciertas limitaciones de ancho de banda podría ser necesario negociar un tamaño menor de fragmento de datos. Para ello se usa la extensión *max_fragment_lenght* (ver fig. 24), donde en los datos de la extensión se incluye un posible valor n para el nuevo tamaño de fragmento.

Figura 24: Extensión para negociar el tamaño máximo del fragmento

Los posibles valores de n están limitados a 2^9, 2^{10}, 2^{11} y 2^{12} bytes. Si un servidor o un cliente reciben un mensaje *hello* con un valor de longitud de fragmento distinto a los anteriores finalizará la ejecución y enviará un mensaje de error del tipo *IllegalParameter*.

Cuando un servidor recibe una petición de este tipo contesta al cliente si puede aceptarla con otro mensaje *hello* extendido donde la longitud del fragmento es la misma que el client le pidió. Si la negociación tiene éxito a partir de ese momento ambas entidades cambian la longitud máxima del fragmento de datos. Se aplicará este tamaño máximo durante toda la sesión, incluyendo las restauraciones de sesiones que se lleven a cabo.

Sobra decir que tanto el servidor como el cliente deben soportar la fragmentación de los mensajes del handshake. La longitud negociada afecta a los mensajes que la capa record puede admitir sin fragmentación. Pero los mensajes de salida podrían ser mayores.

9.3. HMAC truncados

Para usar un HMAC truncado que solo produzca 80 bits, reduciendo la cantidad de bits a enviar, el cliente envía un mensaje *hello* extendido del tipo *truncated_hmac* y el campo de datos de la extensión vacío. Si el servidor recibe este mensaje y está de acuerdo en usar HMAC truncado responde con un mensaje *hello* extendido incluyendo el mismo tipo de extensión y con el campo de datos de la extensión vacío. De forma que a partir del momento en que tiene éxito esta negociación solo se usan los primeros 10 bytes del cálculo de HMAC sobre el mensaje.

Destacar, que si se usan *CipherSuites* que no permitan esta extensión de funcionalidad, y se selecciona una de esas *CipherSuites* al negociar los parámetros de sesión, esta extensión no tendrá ningún efecto.

Sólo afecta a los cálculos de HMAC de los mensajes del protocolo Record pero no afecta a los cálculos de las claves de sesión. Y se aplica a lo largo de la sesión, incluyendo restauraciones de sesión.

A. Cálculo de claves y otras funciones

A.1. HMAC

El cálculo de campos MAC en TLS se realiza mediante HMAC. HMAC puede usar cualquier algoritmo hash, pero TLS usa en su handshake en concreto dos algoritmos: MD5 y SHA-1, que se denotan como HMAC_MD5(secreto, datos) y HMAC_SHA(secreto,datos).

Además, en el cálculo es preciso expandir los datos secretos a bloques de datos para generar las claves o validación. Para ello se emplea una función llamada *PRF* que toma como parámetros un secreto, una semilla, y una etiqueta de identificación y produce una salida de longitud arbitraria.

Primero, definimos una función de expansión de los datos, *P_hash(secreto, datos)* que usa una única función hash:

$$\begin{aligned} P\_hash(secreto, semilla) &= HMAC\_hash(secreto, A(1) + semilla) + \\ &\quad HMAC\_hash(secreto, A(2) + semilla) + \\ &\quad HMAC\_hash(secreto, A(3) + semilla) + \\ &\quad \ldots \end{aligned} \tag{3}$$

donde + significa la concatenación, y la función A() se define como

$$\begin{aligned} A(0) &= semilla \\ A(i) &= HMAC\_hash(secreto, A(i-1)) \end{aligned} \tag{4}$$

P_hash puede usarse iterativamente cuantas veces sea necesaria para producir la cantidad de datos necesarios.

La función PRF se crea dividiendo los datos secretos en dos fragmentos (S_1 y S_2) y usando una mitad para generar P_MD5 y la otra mitad para generar P_SHA-1, y entonces usar la operación or-exclusiva de la salida de esas dos funciones:

$$\begin{aligned} PRF(secreto, etiqueta, semilla) &= P\_MD5(S_1, etiqueta + semilla) \\ & XOR \quad P\_SHA-1(S_2, etiqueta \\ & +semilla). \end{aligned} \tag{5}$$

La etiqueta es una cadena en ASCII y debe añadirse tal cual, sin caracteres de final de cadena ni ningún byte de longitud.

A.2. Cálculo de claves de sesión

TLS necesita generar claves de sesión y datos relacionados con los cálculos MAC seguros. Para ello se usa unos datos secretos intercambiados en el protocolo TLS Handshake como base para obtener estos datos.

Primero se genera una serie de bytes que denominamos *Key Material*:

$$\begin{aligned} key\_material &= PRF(DatosSecretos, \\ & "keyexpansion", \\ & Servidor.random + Cliente.random) \end{aligned} \tag{6}$$

Esta función devuelve un bloque de bytes que se divide de la siguiente forma:

$$\begin{aligned} MAC\_secret\_cliente &= key\_material[0, th] \\ MAC\_secret\_servidor &= key\_material[th+1, 2th] \\ Clave\_escritura\_cliente &= key\_material[2th+1, 2th+1+tc] \\ Clave\_escritura\_servidor &= [2th+2+tc, 2th+2+2tc] \\ IV\_cliente &= key\_material[2th+3+2tc, 2th+3+2tc+tiv] \\ IV\_servirdor &= key\_material[2th+4+2tc+tiv, 2th+4+2tc+2tiv] \end{aligned}$$

siendo *th* el parámetro de tamaño devuelto por una función hash, *tc* el número de bytes de una clave de sesión e *tiv* el tamaño de los vectores de inicialización para los algoritmos de cifrado por bloques.

B. Tipos de mensajes Alert

Nombre	Código	Tipo	Descripción
close_notify	0	warning	Avisa al receptor que el emisor no va a enviarle ningún mensaje más, y la sesión no se podrá restaurar.
bad_certificate	42	opcional*	El certificado estaba corrupto.
unsupported_certificate	43	opcional*	El certificado es de un tipo no soportado.
certificate_revoked	44	opcional*	El certificado ha sido revocado por su firmante.
certificate_expired	45	opcional*	El certificado no es actualmente valido.
certificate_unknown	46	opcional*	El certificado no se ha podido procesar por una causa desconocida.
decrypt_error	51	opcional*	Una operación criptográfica del handshake ha fallado.
user_canceled	90	warning	El handshake ha sido cancelado por cualquier causa ajena al protocolo.
no_renegotiation	100	warning	Enviado por el cliente como respuesta a un mensaje *HelloRequest* o por el servidor en respuesta a un mensaje *ClientHello* después un handshake inicial.

*: hemos considerado estos mensajes como opcional porque dependen de la consideración del implementador del protocolo.

Nombre	Código	Tipo	Descripción
unexpected_message	10	fatal	Se ha recibido un tipo de mensaje que no se esperaba.
bad_record_mac	20	fatal	Se ha recibido un mensaje record con un campo MAC incorrecto.
decryption_failed	21	fatal	Se ha recibido un TLSCipherText incorrecto: o no era múltiplo del tamaño de bloque o sus valores padding no son correctos
record_overflow	22	fatal	El tamaño del record excede los tamaños especificados.
decompression_failure	30	fatal	La función de decompresión ha recibido un valor no válido
handshake_failure	40	fatal	El emisor es incapaz de negociar un conjunto de parámetros que garanticen la seguridad.
illegal_parameter	47	fatal	Un campo del mensaje está fuera de rango o es inconsistente con el resto del mensaje.
unknown_ca	48	fatal	La cadena de certificados es válida pero se rechaza porque no se ha localizado un certificado de una CA.
access_denied	49	fatal	Se ha recibido un certificado válido pero al aplicar las políticas de control de acceso el emisor ha decido no continuar.
decode_error	50	fatal	Un mensaje no puede decodificarse porque algún campo es erróneo.
export_restriction	60	fatal	Se ha detectado una negociación que no está de acuerdo con las restricciones de *export*.
protocol_version	70	fatal	La versión del protocolo del cliente se reconoce pero no se soporta.
insufficient_security	71	fatal	Una negociación ha fallado porque el servidor requiere un cifrado más seguro que el que el cliente soporta.
internal_error	80	fatal	Se ha producido un error interno que no guarda relación con el extremo o la corrección del protocolo que impide continuar la ejecución.

C. Nuevos Tipos de Mensajes Alert para extensiones

Nombre	Código	Descripción
unsupported_extension	110	Un cliente ha recibido un mensaje *hello* extendido cuyas peticiones no se corresponden con las que él envió en su mensaje *hello*
certificate_unobtainable	111	Lo envía un servidor cuando no es capaz de recuperar la cadena de certificados de una petición de uso de URLs de certificados
unrecognized_name	112	Un servidor ha recibido una lista de nombres de servidores pero no reconoce el nombre de un servidor
bad_certificate_status_response	113	Lo envía un cliente cuando recibe una respuesta de estado de certificado no válida
bad_certificate_hash_value	114	Lo envía un servidor cuando el hash de un certificado no concuerda con el hash que le proporciona el cliente para ese certificado