Universitat Oberta de Catalunya

uoc.edu

TLS 1.3

Autor: Juan Carlos Romero Benito

Plan de Estudios: Máster Interuniversitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones.

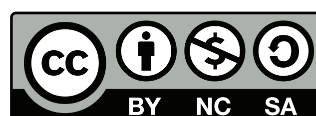
Trabajo fin de máster.

Director del TFM: Pau del Canto Rodrigo

Profesor responsable de la asignatura: Victor Garcia Font

Fecha de entrega: 28 de diciembre de 2020

Este documento esta realizado bajo licencia Creative Commons “Reconocimiento-NoCommercial-CompartirIgual 3.0 España”.



FICHA DEL TRABAJO FINAL

Título del trabajo:	TLSv1.3
Nombre del autor:	Juan Carlos Romero Benito
Nombre del consultor/a:	Pau del Canto Rodrigo
Nombre del PRA:	Víctor García Font
Fecha de entrega:	12/2020
Titulación:	MISTIC
Área del Trabajo Final:	Protocolos y aplicaciones de seguridad
Idioma del trabajo:	Castellano
Palabras clave:	TLS, SSL

Resumen del Trabajo

Este trabajo se enmarca en el estudio del protocolo TLS. En él se aporta una perspectiva histórica y técnica de las versiones que lo han compuesto. Se detalla su funcionamiento y los fallos de seguridad que propiciaron su evolución a lo largo de su historia. Además, se analiza en profundidad su nueva versión, TLSv1.3, con énfasis en las mejoras de seguridad y rendimiento que ha introducido respecto a su anterior versión. Se crea un laboratorio para la demostración empírica de su funcionamiento, así como para analizar las ventajas de una manera práctica. Al final se extraen una serie de conclusiones sobre esta nueva versión y se comentan las posibles líneas de trabajo futuro.

Abstract

This dissertation is a review of the TLS protocol. It provides a historical and technical perspective of the versions that have composed it. It details its operation and the security failures that led to its evolution throughout its history. In addition, it analyses in depth its new version, TLSv1.3, with emphasis on the security and performance improvements it has introduced with respect to its previous version. A laboratory is created for the empirical proof of its operation, as well as to analyze the advantages in a practical way. At the end, a series of conclusions are drawn from this new version and possible lines of future work are discussed.

A Ana, pola túa paciencia
e cariño ao longo
de todos estes anos

Índice general

Página

1. Introducción **1**
1.1. Motivación 1
1.2. Objetivos 2
1.2.1. Objetivo General 2
1.2.2. Objetivos Específicos 2
1.3. Estructura de la memoria 3
1.4. Metodología 4
1.5. Planificación 5

2. Estado del arte **9**
2.1. Origen de SSL/TLS 9
2.1.1. Motivación y origen 9
2.1.2. Historia de las distintas versiones 12

3. Evolución de TLS **15**
3.1. Fundamentos de SSL/TLS 16
3.1.1. Conjunto de cifrados (*Cipher Suite*) 16
3.1.2. Internet PKI: Infraestructura de clave pública 20
3.1.3. Limitaciones del protocolo 21
3.2. SSLv3 23
3.2.1. Características de SSLv3 23
3.2.1.1. Mensajes en SSLv3 23
3.2.1.2. *Handshake* en SSLv3 26
3.2.2. Fallos de seguridad de SSLv2 y SSLv3 30
3.3. TLS 33
3.3.1. Características de TLSv1.0 33

3.3.2. Características de TLSv1.1 37
3.3.2.1. Extensiones 37
3.3.3. Características de TLSv1.2 41
3.3.4. Fallos de seguridad de TLSv1.0-1.2 42
3.3.4.1. BEAST (Browser Exploit Against SSL/TLS) 42
3.3.4.2. CRIME (Compression Ratio Info-leak Made Easy) . . . 45
3.3.4.3. BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext) 46
3.3.4.4. *Heartbeat* 47
3.3.4.5. FREAK (Factoring RSA Export Keys) 48
3.3.4.6. POODLE (Padding Oracle On Downgraded Legacy Encryption) 50
3.4. TLSv1.3 53
3.4.1. Características de TLSv1.3 y diferencias con TLSv1.2 53
3.4.1.1. *Handshake* en TLSv1.3 54
3.4.1.2. Resumen de sesión 57
3.4.1.3. Modos 1-RTT y Zero-RTT 58
3.4.1.4. Protección *Anti-downgrade* 60
3.4.1.5. *Cipher suites* soportadas por TLSv1.3 60

4. Soporte y uso de TLSv1.3 63
4.1. Soporte *software* para TLSv1.3 63
4.2. Grado de adopción de TLSv1.3 64
4.3. Mejoras de rendimiento de TLSv1.3 68

5. Pruebas en laboratorio 71
5.1. Topología y *software* utilizado 71
5.2. Instalación y montaje 73
5.2.1. Configuración de F5 BIG-IP 73
5.2.2. Configuración de los servidores web 79
5.3. Pruebas realizadas 82
5.3.1. Diferencias de *Handshake* entre TLSv1.2 y 1.3 82
5.3.1.1. Comunicación entre cliente y balanceador 82
5.3.1.2. Comunicación entre balanceador y servidores 87
5.3.2. Resumen de sesión en TLSv1.2 vs TLSv1.3 89
5.3.3. Resumen de sesión 0-RTT con TLSv1.3 92
5.3.4. *Downgrade* en TLSv1.3 94

6. Conclusiones **95**
6.1. Líneas de trabajo futuro 96

A. Anexo I: *Cipher suites* obligatorias (SSLv3 y TLSv1.0) **99**
A.1. *Cipher suites* soportadas por SSLv3 99
A.2. *Cipher suites* obligatorias por RFC para TLSv1.0 101

B. Glosario de acrónimos **103**

C. Glosario de términos **105**

Bibliografía **107**

Índice de figuras

Figura **Página**

1.1. Tareas de la planificación del proyecto 6
1.2. Diagrama de Gantt de la planificación 7

2.1. Modelo OSI vs TCP/IP [1] 11
2.2. Línea temporal de aparición de las distintas versiones de SSL/TLS . . . 13

3.1. Ejemplo de cipher suite 16
3.2. Ciclo de vida de PKI [2] 21
3.3. Formato de la *Record Layer* 27
3.4. *Handshake* de SSL utilizando TCP 28
3.5. *Handshake* SSL con autenticación del cliente [3] 30
3.6. Funcionamiento de HMAC [3] 34
3.7. Función *PseudoRandom Function* utilizada en TLSv1.0 [3] 35
3.8. Uso de PRF para crear el *master secret* y el material de claves [3] 36
3.9. Ataque BEAST contra CBC con un IV predecible [2] 44
3.10. Relleno de bloques en SSLv3 vs TLSv1.0 [2] 51
3.11. Handshake en TLSv1.2 y 1.3 [4] 54
3.12. 1-RTT en TLSv1.2 y 1.3 [5] 58
3.13. 0-RTT en TLSv1.3 [5] 59

4.1. Adopción de TLSv1.3 según datos de Mozilla 65
4.2. Adopción en la categoría Ciencia según SANS [6] 66
4.3. Adopción en la categoría Compras según SANS [6] 67
4.4. Adopción en la categoría Negocios según SANS [6] 67
4.5. *Benchmark* TLSv1.2 vs TLSv1.3 69

5.1. Topología del laboratorio. 72

5.2. Definición de un nodo en F5. 74
5.3. Nodos levantados en F5. 74
5.4. Definición de un *pool* en F5. 75
5.5. Definición de un *virtual server* en F5. 76
5.6. *Virtual servers* en F5. 77
5.7. Certificados en F5. 77
5.8. Perfil TLS en F5. 78
5.9. Arquitectura *Full-Proxy* en F5 [7]. 82
5.10. Captura del *handshake* entre cliente y balanceador. 83
5.11. Primera captura del *ClientHello* entre cliente y balanceador. 83
5.12. Segunda captura del *ClientHello* entre cliente y balanceador. 85
5.13. Tercera captura del *ClientHello* entre cliente y balanceador. 85
5.14. Captura del *ServerHello* entre el balanceador y el cliente. 86
5.15. Captura del *handshake* entre el balanceador y un servidor. 87
5.16. Captura del *ClientHello* entre balanceador y servidor. 87
5.17. Captura del *ServerHello* entre servidor y balanceador. 88
5.18. Resumen de sesión mediante el *Session ID*. 89
5.19. Envío de un *Session ticket* al balanceador. 90
5.20. Resumen de sesión utilizando un *ticket*. 91
5.21. Resumen de sesión utilizando una PSK en TLSv1.3. 92
5.22. Resumen de sesión 0-RTT con TLSv1.3. 93
5.23. *Anti-Downgrade* en TLSv1.3. 94

Índice de cuadros

Tabla **Página**

3.1. *Cipher suites* en TLSv1.3 61

A.1. *Cipher suites* en SSLv3 100
A.2. *Cipher suites* en TLSv1.0 102

Capítulo 1

Introducción

Índice general

1.1. Motivación **1**

1.2. Objetivos **2**

1.2.1. Objetivo General 2

1.2.2. Objetivos Específicos 2

1.3. Estructura de la memoria **3**

1.4. Metodología **4**

1.5. Planificación **5**

Este capítulo realiza una introducción al trabajo fin de máster realizado, explicando su motivación, citando sus objetivos, su estructura, la metodología que se ha utilizado y explicando su planificación.

1.1. Motivación

Desde el nacimiento de la propia Internet, la necesidad de nuevas funcionalidades, protocolos y utilidades han superado a la seguridad. Se han creado todos los protocolos sin una conciencia exhaustiva de que la seguridad de las comunicaciones puede ser tan importante como el propio funcionamiento de las mismas.

Hace ya casi tres décadas del nacimiento de la *World Wide Web* y de la publicación de los primeros RFCs (*Request for Comments*) de HTTP (*Hypertext*

Transfer Protocol), pero hasta hace relativamente poco, muchas webs solo cifraban los inicios de sesión o transacciones económicas, pero no así el resto de la navegación. Es interesante recordar casos como el de Facebook que en 2011 [8] habilitaba una opción de perfil para utilizar HTTPS (*HTTP Secure*) más allá del momento del login, recordemos que SSLv3.0 (*Secure Sockets Layer*) se publicó en 1996 y TLSv1.0 (*Transport Layer Security*) en 1999. La adopción de los métodos de cifrado y seguridad en Internet para multitud de protocolos ha sido lenta y tardía.

Este trabajo se enmarca en la aparición de la versión 1.3 de TLS publicada por el IETF (*Internet Engineering Task Force*) como estándar propuesto en el año 2018 [9]. El IETF, cuya misión es realizar estándares para contribuir a la evolución de la arquitectura de Internet [10], llevaba desde el año 2014 [11] trabajando en un nuevo estándar que sustituyese a la versión 1.2 y la mejorase, tanto en aspectos de seguridad como de rendimiento.

1.2. Objetivos

1.2.1. Objetivo General

El objetivo de este trabajo es realizar un estudio de las diferentes versiones de TLS, sus fallos de seguridad, razones de evolución y la evaluación de la nueva versión 1.3. Además, se realizará un estudio práctico que mostrará el funcionamiento empírico de las distintas versiones y evaluará las mejoras tanto de seguridad como de rendimiento.

1.2.2. Objetivos Específicos

- Conocer el origen, uso y evolución de las distintas versiones de SSL/TLS que han existido desde su creación.
- Analizar el funcionamiento de cada versión de TLS, sus problemas de seguridad y mejoras. Se pondrá especial énfasis en las diferencias de las versiones más recientes, TLSv1.2 y TLSv1.3.

- Evaluar el nuevo estándar TLSv1.3 analizando en profundidad su mejoras de seguridad y/o rendimiento.
- Analizar el grado de soporte actual de las distintas versiones y las incompatibilidades de algunos sistemas con TLSv1.3.
- Evaluar distintos *benchmarks* realizados en estudios y analizar las mejoras de rendimiento que supone TLSv1.3 respecto a sus antecesores.
- Examinar en un entorno de laboratorio de manera empírica el funcionamiento de las distintas versiones y sus mejoras de seguridad y rendimiento.
- Crear una simple guía de despliegue de TLSv1.3 con las tecnologías utilizadas en el laboratorio.

1.3. Estructura de la memoria

La presente memoria del proyecto está estructurada en capítulos de la siguiente manera:

- **Introducción**: Se trata el marco en el cual se desarrolla el proyecto y los objetivos que se van a cumplir, se explica la estructura de la propia memoria, la planificación seguida para el desarrollo de este proyecto, así como la metodología aplicada, las tareas que se han definido, la estimación y el seguimiento del proyecto.
- **Estado del arte**: Se ofrece una visión global sobre el tema tratado y se introduce desde un punto de vista histórico la aparición de SSL/TLS y sus aplicaciones.
- **Evolución de TLS**: Se detallan las diferencias técnicas entre las distintas versiones de TLS, así como sus fallos de seguridad, los motivos de su evolución y las mejoras que se fueron introduciendo.
- **Soporte y uso de TLS**: Se detalla el soporte actual de los distintos sistemas operativos y software a las distintas versiones. Se analiza el grado de uso de cada versión en la actualidad en la web y en cada sector. Además, se incluirá un análisis de *benchmarks* realizados en otros estudios.

- **Pruebas en laboratorio**: Se creará una pequeña prueba de concepto para evaluar las mejoras de seguridad y rendimiento de TLSv1.3 en comparación con sus antecesores. Además, se comentará de manera sencilla el despliegue realizado en este trabajo a modo de guía.
- **Conclusiones**: Sobre los resultados obtenidos se desarrollan las conclusiones, se evalúa el trabajo realizado y se exploran las líneas de desarrollo futuro.
- **Apéndices**:
  - **Glosario de acrónimos**: Se detallan todas las siglas utilizados en esta memoria.
  - **Glosario de términos**: Se definen los términos técnicos más importantes utilizados en este documento.
  - **Bibliografía**: Fuentes de información utilizadas para el desarrollo de este proyecto.

1.4. Metodología

El primer paso en la planificación del proyecto es seleccionar la metodología adecuada para llevar a cabo el mismo. La metodología marca el camino a seguir y ayuda a planificar cada una de las etapas del proyecto. Existen multitud de metodologías ampliamente conocidas y probadas que dependen del ámbito en el cual se enmarque el trabajo en cuestión, pero también existen proyectos que pueden no adaptarse a las mismas y los cuales pueden seguir metodologías propias. En este caso por la naturaleza de investigación y revisión de este proyecto se va a seguir una metodología propia.

El enfoque utilizado en este proyecto para dar cumplimiento a los objetivos fijados en la Sección 1.2 es el siguiente:

- **Definición del ámbito del trabajo y su planificación**

  En esta primera etapa se introduce el ámbito del trabajo, se definen los objetivos a cumplir, se cita la metodología seguida y se explica la planificación para la ejecución del proyecto.

- **Investigación sobre las distintas versiones de TLS**

  En esta fase se investigará en la documentación requerida, como pueden ser los RFCs donde se define TLS o bibliografía específica, para poder evaluar y explicar el motivo y las características de cada versión del protocolo que ha existido. Se indagará además en las vulnerabilidades que han afectado a cada versión.

- **Evaluación de la versión 1.3 de TLS**

  En esta fase se definirá el origen, la necesidad, las características y ventajas que introduce la nueva versión del protocolo.

- **Búsqueda de estudios que demuestren el grado de adopción y *benchmarks***

  En esta fase se indagará en los estudios que existen sobre el grado de adopción a día de hoy de la versión 1.3 y además, se buscará conocer cual es la mejora que se está obteniendo en la migración a esta versión.

- **Análisis práctico de TLSv1.3**

  En esta fase se realizarán las pruebas oportunas en laboratorio para ver de manera práctica los beneficios e inconvenientes de TLSv1.3, así como apreciar el propio funcionamiento del protocolo mediante capturas de tráfico. Además, se documentará el propio montaje del laboratorio creado a modo de guía.

- **Evaluación del trabajo y conclusiones**

  En esta fase se determinarán las conclusiones de todo el trabajo de una manera concisa, para así determinar las ventajas de este nuevo estándar y ver si se aprecian problemas derivados del mismo.

1.5. Planificación

En esta sección se exponen los aspectos referentes a la planificación del proyecto. Se identifican todas las actividades a llevar a cabo y la duración de cada una de ellas para alcanzar los objetivos definidos en la Sección 1.2 para este proyecto. Se estima el esfuerzo y tiempo que consume cada tarea para conocer en

cualquier momento si el proyecto sufre algún tipo de desviación respecto al plan inicial previsto y aplicar las correcciones oportunas para cumplir el plan inicial. En la Figura 1.1 se pueden observar todas las tareas e hitos que se han definido con su planificación inicial y fechas estimadas.

DIAGRAMA DE GANTT

	Nombre de tarea	Duración	Comienzo	Fin
1	**Trabajo final de máster**	**79 días**	**mié 16/09/20**	**mar 05/01/21**
2	Estudio previo del tema	2 días	mié 16/09/20	jue 17/09/20
3	Revisión de la documentación de la asignatura	1 día	vie 18/09/20	vie 18/09/20
4	Planificación previa del TFM	1 día	lun 21/09/20	lun 21/09/20
5	**Introducción**	**6 días**	**mar 22/09/20**	**mar 29/09/20**
6	Motivación	1 día	mar 22/09/20	mar 22/09/20
7	Definición de los objetivos	1 día	mié 23/09/20	mié 23/09/20
8	Estructura de la memoria	1 día	jue 24/09/20	jue 24/09/20
9	Metodología	1 día	vie 25/09/20	vie 25/09/20
10	Planificación	2 días	lun 28/09/20	mar 29/09/20
11	**Entrega PEC1**	**0 días**	**mar 29/09/20**	**mar 29/09/20**
12	**Estado del arte**	**6 días**	**mié 30/09/20**	**mié 07/10/20**
13	Motivación para el nacimiento de SSL	2 días	mié 30/09/20	jue 01/10/20
14	Origen de SSL/TLS	2 días	vie 02/10/20	lun 05/10/20
15	Historia de las distintas versiones de SSL/TLS	2 días	mar 06/10/20	mié 07/10/20
16	**Evolución de TLS**	**22 días**	**jue 08/10/20**	**vie 06/11/20**
17	Características de SSLv3	3 días	jue 08/10/20	lun 12/10/20
18	Fallos de seguridad de SSLv3	1 día	mar 13/10/20	mar 13/10/20
19	Características de TLSv1.0	3 días	mié 14/10/20	vie 16/10/20
20	Fallos de seguridad de TLSv1.0	1 día	lun 19/10/20	lun 19/10/20
21	Características de TLSv1.1	3 días	mar 20/10/20	jue 22/10/20
22	Fallos de seguridad de TLSv1.1	1 día	vie 23/10/20	vie 23/10/20
23	**Entrega PEC2**	**0 días**	**mar 27/10/20**	**mar 27/10/20**
24	Características de TLSv1.2	3 días	lun 26/10/20	mié 28/10/20
25	Fallos de seguridad de TLSv1.2	1 día	jue 29/10/20	jue 29/10/20
26	Características de TLSv1.3	3 días	vie 30/10/20	mar 03/11/20
27	Diferencias entre TLSv1.2 y TLSv1.3	3 días	mié 04/11/20	vie 06/11/20
28	**Soporte y uso de TLS**	**6 días**	**lun 09/11/20**	**lun 16/11/20**
29	Investigación de estudios sobre el grado de implantación de TLSv1.3	2 días	lun 09/11/20	mar 10/11/20
30	Documentación de la información del grado de implantación	1 día	mié 11/11/20	mié 11/11/20
31	Investigación de estudios de Benchmarks de TLSv1.3	2 días	jue 12/11/20	vie 13/11/20
32	Documentación de la información de los benchmarks	1 día	lun 16/11/20	lun 16/11/20
33	**Pruebas en laboratorio**	**18 días**	**mar 17/11/20**	**jue 10/12/20**
34	Despliegue del laboratorio	5 días	mar 17/11/20	lun 23/11/20
35	Análisis de las pruebas a realizar	2 días	mar 24/11/20	mié 25/11/20
36	**Entrega PEC3**	**0 días**	**mar 24/11/20**	**mar 24/11/20**
37	Realización de las pruebas	7 días	jue 26/11/20	vie 04/12/20
38	Documentación de los resultados obtenidos	2 días	lun 07/12/20	mar 08/12/20
39	Documentación del montaje del laboratorio y las pruebas	2 días	mié 09/12/20	jue 10/12/20
40	**Conclusiones**	**3 días**	**vie 11/12/20**	**mar 15/12/20**
41	Evaluación del trabajo realizado	1 día	vie 11/12/20	vie 11/12/20
42	Redacción de las conclusiones del trabajo	2 días	lun 14/12/20	mar 15/12/20
43	**Revisión de la memoria del trabajo**	**6 días**	**mié 16/12/20**	**jue 24/12/20**
44	Revisión de la estructura y la maquetación	5 días	mié 16/12/20	mar 22/12/20
45	Revisión y corrección ortográfica y gramatical	1 día	mié 23/12/20	mié 23/12/20
46	**Entrega PEC4**	**0 días**	**jue 24/12/20**	**jue 24/12/20**
47	**Preparación de la defensa**	**8 días**	**jue 24/12/20**	**mar 05/01/21**
48	Preparación de la presentación para la defensa	7 días	jue 24/12/20	vie 01/01/21
49	Grabación del vídeo	1 día	lun 04/01/21	lun 04/01/21
50	**Entrega PEC5**	**0 días**	**mar 05/01/21**	**mar 05/01/21**

Figura 1.1: Tareas de la planificación del proyecto

En la Figura 1.2 se puede apreciar el diagrama de Gantt generado a partir de la planificación de las tareas, así como la relación entre cada una de ellas. No existen tareas que se vayan a desarrollar en paralelo, la relación entre todas ellas es Fin-Comienzo puesto que al existir un único recurso no hay posibilidad de reducir el tiempo del proyecto.

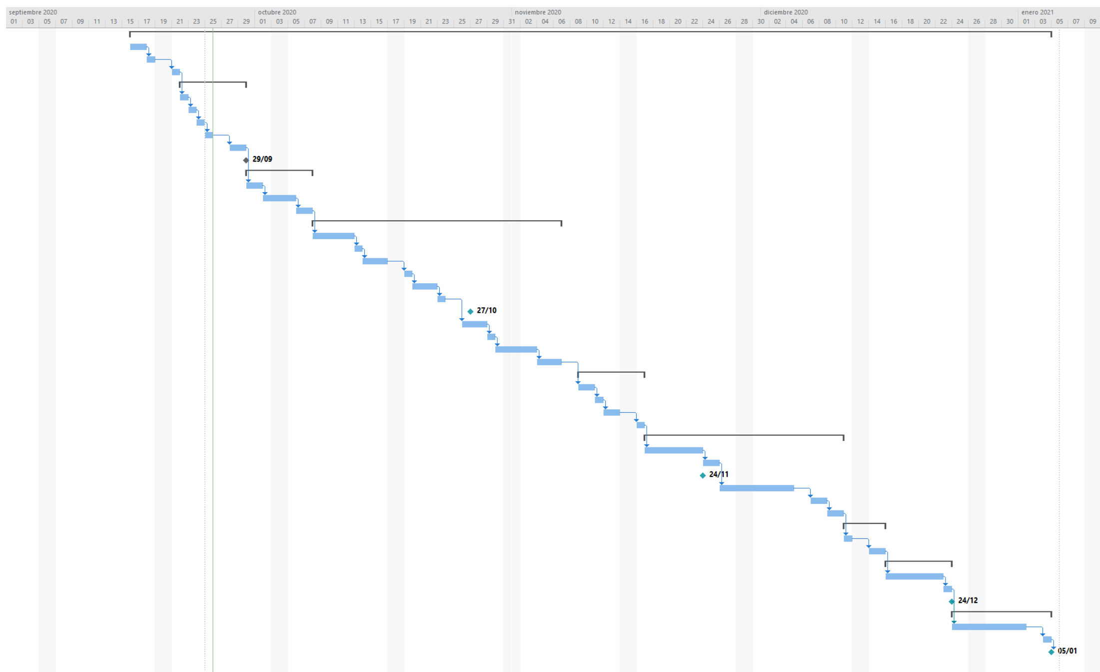


Figura 1.2: Diagrama de Gantt de la planificación

Capítulo 2

Estado del arte

Índice general

2.1. Origen de SSL/TLS **9**

2.1.1. Motivación y origen 9

2.1.2. Historia de las distintas versiones 12

En este capítulo se realiza una breve introducción a SSL/TLS y se da una visión global de su estado actual. Se explican los motivos de su nacimiento, su origen e historia, además de su línea temporal de actualizaciones del protocolo a lo largo de los años. También se citan algunos de sus principales problemas y fallos que han surgido.

2.1. Origen de SSL/TLS

2.1.1. Motivación y origen

Desde la creación de los protocolos que permitieron el nacimiento de ARPANET[1], y posteriormente Internet, el funcionamiento ha sido más importante que la seguridad, y los protocolos no se han diseñado con la suficiente desconfianza

[1]ARPANET [12] (***A****dvanced* ***R****esearch* ***P****rojects* ***A****gency* ***Net****work*) fue una red creada por la *Advanced Research Project Agency*, una agencia de EEUU perteneciente al Departamento de Defensa del mismo país, encargada de desarrollar este proyecto para comunicar las instituciones académicas y del estado. Esta red debía ser capaz de sobrevivir a ataques, como por ejemplo nucleares.

en los integrantes de la misma red. Esto se debe a que en un principio todos los integrantes eran instituciones estatales de EEUU [12] en las cuales se podía confiar. Esto ha producido que desde principios de los años 70 del siglo pasado se hayan producido continuamente problemas relacionados con la seguridad en relación a todos los protocolos. Poco a poco se demostró que los protocolos necesitaban una revisión y actualización para asegurar un comportamiento seguro y, como es conocido, se siguen encontrando fallos tanto en los propios protocolos como en las implementaciones específicas. Se pueden citar protocolos de capa de aplicación que, aún teniendo capacidades de autenticación muy básicas, no aseguraban la confidencialidad de la comunicación, como pueden ser FTP (*File Transport Protocol*), SMTP (*Simple Mail Transport Protocol*) o HTTP, por poner algunos ejemplos. Estos protocolos tienen algo en común, la integración de la seguridad de los mismos ha pasado por integrar SSL/TLS sobre los mismos.

En este momento no existe manera de asegurar el camino entre un navegador y servidor web. Cualquier nodo en la red por donde pasara (o a quién llegara) la comunicación podía ver el mensaje completo. Se debe recordar que un cliente europeo que accediese a un servidor norteamericano podía pasar por varios países, equipamiento de los proveedores de distintas empresas y varias jurisdicciones. Ni el cliente web ni el servidor tenían ningún control sobre el camino que tomaría la información (esta responsabilidad recae en la capa de red). Los datos más privados no pueden viajar de esta manera por ningún medio, ni electrónico, ni escrito. Además, al no existir métodos de autenticación o integridad es posible que el servidor al cual el cliente cree estar accediendo sea falso o se esté realizando un ataque de *Man-in-the-middle*[2].

SSL y TLS son protocolos que pertenecen a la capa de Presentación en el modelo OSI (*Open System Interconnection*). OSI definido en la ISO/IEC 7498-1 es un modelo de referencia para la creación de protocolos de red, este define una serie de capas que dividen las atribuciones de cada una de las mismas con sus funciones [13]. La implementación estándar de facto es TCP/IP que, aunque no utiliza todas las capas definidas en OSI, se basa en el modelo. Aunque en el mode-

[2]Estos tipos de ataques consisten en interceptar una comunicación con cualquier tipo de motivación, tratando de suplantar al destinatario.

lo TCP/IP que se utiliza, SSL y TLS forman parte de la capa de Aplicación igual que los protocolos a los que aseguran, es conocido que sus funciones pertenecen a una capa inferior que, como se ha comentado, es la capa de Presentación.

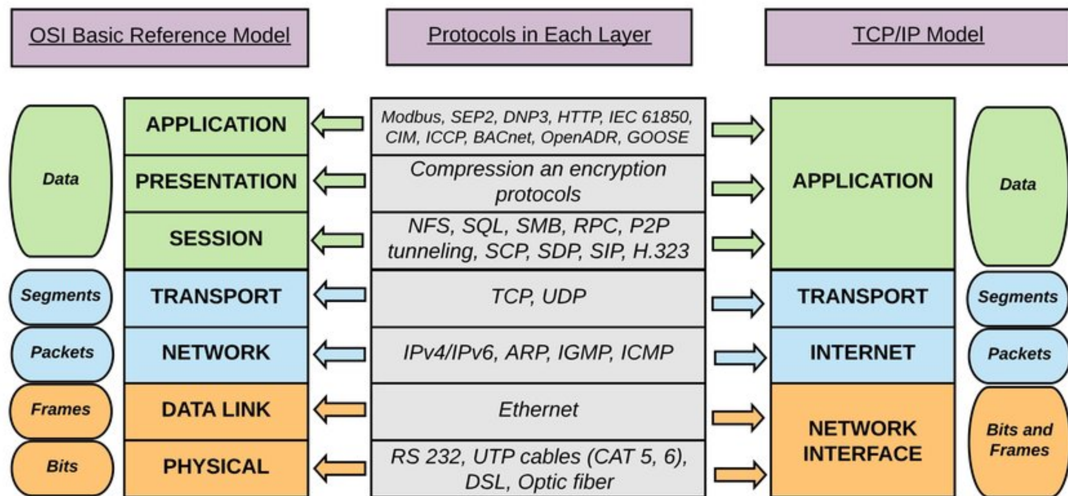


Figura 2.1: Modelo OSI vs TCP/IP [1]

En la Figura 2.1 se puede apreciar una comparación entre ambas pilas y una aproximación a cuales de los protocolos de nuestros días pertenecen a cada capa en el modelo de referencia OSI.

Con la expansión de Internet y la creación de la Web, uno de los líderes en aquel momento de la industria de los navegadores web, Netscape Communications, empezó a considerar el desarrollo de un protocolo que asegurara las comunicaciones y el comercio electrónico. En 1994 comenzaron a trabajar en un protocolo llamado *Secure Sockets Layer* (SSL), cuyo objetivo era añadir seguridad a las comunicaciones HTTP. El navegador Netscape Navigator fue liberado a final del mismo año implementando la versión 2.0 de SSL, puesto que la versión 1.0 fue simplemente una prueba de concepto y nunca llegó a salir al mercado. En esta época Netscape era el navegador más utilizado [3].

2.1.2. Historia de las distintas versiones

Como se ha comentado en la sección anterior, la primera versión de SSL (1.0) nació a mitad de 1994 como una prueba de diseño y no fue integrada en ningún navegador o *software* comercial. La versión 2.0 fue publicada e integrada en Netscape Navigator al final del mismo año y principios de 1995 [3]. En esa época Microsoft desarrollo PCT (*Private Communication Technology*) como una mejora sobre la versión 2.0 de SSL. Debido a las debilidades de esta versión, en las que se entrará en detalle en el siguiente capítulo, se desarrolló la versión 3.0 que fue publicada en 1996 [14]. Netscape en esta época animaba a la industria a participar y realizaba un desarrollo abierto del protocolo, pero aún así este pertenecía a la compañía y acabó obteniendo una patente otorgada por los EEUU. Cuando el proyecto pasó a manos del IETF, como organización internacional de estándares, se modificó el nombre del protocolo para cambiar la apariencia de que este pertenecía a una única empresa. Este paso fue lento y estuvo enmarcado por problemas entre Netscape y Microsoft, como consecuencia de una batalla por dominar la Web. Es interesante ver que además, el cambio de nombre del protocolo fue una manera de que Microsoft aceptara de buen grado el nuevo estándar y así eliminar las referencias al protocolo creado inicialmente por Netscape [2]. Con los años se ha visto que esto no ha servido de mucho, tal vez mucha gente no sepa que el protocolo inicial fue creado por Netscape, o incluso la existencia de esa compañía, pero lo que es un hecho es que el nombre de SSL se continúa utilizando aún en nuestros días de manera intercambiable y natural con TLS.

En 1999 la primera versión oficial de TLS (1.0) vio la luz como el RFC 2246 [15], pero este cambio fue un simple lavado de cara, puesto que era una mínima evolución de SSLv3.0. En 2006 se publicó la versión 1.1 de TLS, que principalmente corregía errores de seguridad, pero además, esta versión fue la primera en introducir las extensiones a TLS. En 2008 fue publicada la versión 1.2 que añadía soporte de autenticación y eliminaba todas las primitivas codificadas directamente en el protocolo, haciendo de este un protocolo más flexible. La versión TLSv1.3 fue publicada en 2018, después de un largo periodo de diseño y que ha prometido mejorar la seguridad y rendimiento como se verá más adelante.

En la siguiente figura se puede apreciar la línea temporal de la aparición de cada versión, con los años de publicación y su RFC en cada caso.

Figura 2.2: Línea temporal de aparición de las distintas versiones de SSL/TLS

Se debe comentar además, que en la última década se han descubierto multitud de vulnerabilidades de SSL/TLS. Alguno de estos fallos de seguridad fueron ya descubiertos en los años 2000 de una manera teórica, como pueden ser ataques de tipo *Padding Oracle*. Sin entrar en detalle se citan las más importantes y se explicarán en el siguiente capítulo:

- **BEAST** (*Browser Exploit Against SSL/TLS*): publicado en 2011, aplica a SSLv3.0 y TLSv1.0, se aprovecha de un fallo en la implementación del modo CBC (*Cipher Block Chaining*) en TLSv1.0. Sección 3.3.4.1.
- **CRIME** (*Compression Ratio Info-leak Made Easy*): publicado en 2012, aplica a versiones hasta TLSv1.2 que utilizan métodos de compresión, un fallo en la compresión sin ofuscar la longitud de los datos no cifrados puede producir la capacidad de un ataque MTM. Sección 3.3.4.2.
- **BREACH** (*Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext*): publicado en 2013, aplica a todas las versiones de SSL/TLS. El objetivo directo no es TLS, si no que se explota la compresión HTTP, pero se tienen que dar unas condiciones determinadas. Sección 3.3.4.3.

- **Heartbleed**: publicado en 2014, aplica a la librería de OpenSSL, específicamente a la extensión *heartbeat*, el atacante puede obtener datos que están en la memoria principal del servidor. Sección 3.3.4.4.
- **POODLE** (*Padding Oracle On Downgraded Legacy Encryption*): publicada en 2014, aplica a los clientes/servidores que todavía soportan SSLv3.0 y con un ataque MTM se realiza un downgrade hacia esa versión. Con esto se ataca la versión SSLv3.0 en modo CBC utilizando un ataque con *Padding*. Sección 3.3.4.6

En el siguiente capítulo se entrará en el detalle de como funcionaba cada versión de los protocolos, así como sus ventajas y fallos demostrados.

Capítulo 3

Evolución de TLS

Índice general

3.1. Fundamentos de SSL/TLS 16

3.1.1. Conjunto de cifrados (*Cipher Suite*) 16

3.1.2. Internet PKI: Infraestructura de clave pública 20

3.1.3. Limitaciones del protocolo 21

3.2. SSLv3 23

3.2.1. Características de SSLv3 23

3.2.2. Fallos de seguridad de SSLv2 y SSLv3 30

3.3. TLS 33

3.3.1. Características de TLSv1.0 33

3.3.2. Características de TLSv1.1 37

3.3.3. Características de TLSv1.2 41

3.3.4. Fallos de seguridad de TLSv1.0-1.2 42

3.4. TLSv1.3 53

3.4.1. Características de TLSv1.3 y diferencias con TLSv1.2 53

Este capítulo examina las distintas versiones de TLS desde un punto de vista más técnico. Se citan las características de varias de sus versiones y los fallos de seguridad que propiciaron su evolución.

3.1. Fundamentos de SSL/TLS

Aunque cada estándar tiene sus características propias, todas las versiones tienen propiedades en común. En las comunicaciones cifradas con SSL/TLS se definen dos roles distintos, un sistema va a actuar como cliente y otro como servidor. Esta diferencia es importante para cualquier comunicación SSL/TLS puesto que las acciones y pasos en la negociación son diferentes dependiendo del rol que desarrolla cada uno. El cliente inicia la comunicación y tiene la responsabilidad de proponer una serie de opciones para la negociación, el servidor escoge una de las opciones propuestas (y por ende soportadas) por el cliente que los dos utilizarán. Aunque la decisión de qué *cipher suite* utilizar corresponde al servidor, este solo podrá utilizar aquellos que el cliente le ha confirmado que soporta, si el servidor no soporta ningún *cipher suite* de los que el cliente le ha enviado podrá cerrar la conexión ante la falta de compatibilidad.

3.1.1. Conjunto de cifrados (*Cipher Suite*)

Un *cipher suite* o conjunto de cifrados, es el conjunto de algoritmos que componen las distintas opciones de autenticación, algoritmos de intercambio de claves y cifrado utilizado en la negociación de SSL/TLS.

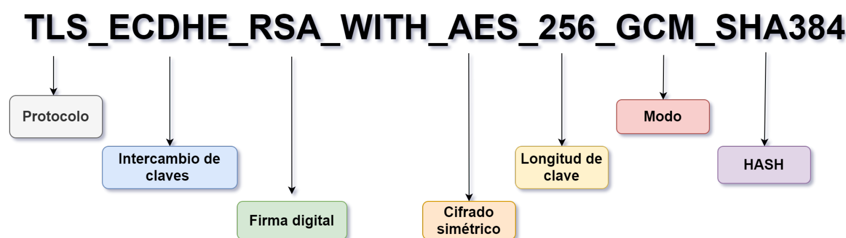


Figura 3.1: Ejemplo de cipher suite

Cliente y servidor tienen un conjunto de *cipher suites* que soportan y que comparten para utilizar el conjunto seleccionado como marco del establecimiento de la conexión cifrada.

- **Protocolo** (*Protocol*): define el protocolo que se utilizará de manera genérica. Será SSL o TLS únicamente.

- **Intercambio de claves** (*Key Exchange*): define el algoritmo de intercambio de claves que se utilizará para llegar a la creación de una clave simétrica compartida para la sesión. Aunque existen más tipos de los aquí comentados, los siguientes son los que realmente están implementados en la práctica, pero existen muchos más que, por su falta de seguridad o inconveniencia, no se utilizan.

  * **RSA** (*Rivest, Shamir y Adleman*): ha sido el estándar de facto como algoritmo de intercambio de claves y universalmente soportado, pero tiene el problema de diseño de que un atacante pasivo que obtenga la clave privada del certificado podrá descifrar todos los datos capturados a lo largo del tiempo, puesto que la clave (simétrica) se envía cifrada con la clave pública del servidor. Por este motivo, aunque se siga utilizando para firmar digitalmente se trata de reemplazar como método de intercambio en favor de otros algoritmos que soportan *forward secrecy*. El intercambio en RSA es un método de transporte de clave, el cliente genera el *premaster secret* y lo envía al servidor cifrado con la clave pública de este, como se ha comentado [16].
  * **DH** (*Diffie-Hellman*): es un protocolo de intercambio de claves por acuerdo que permite a dos partes establecer una clave compartida sobre un canal de comunicación inseguro. Aunque este tipo de negociación está a salvo de ataques pasivos, un atacante podría impersonarse como una de las dos partes, es por este motivo que al intercambio de claves con este método se le añade autenticación. El truco de Diffie-Hellman es utilizar una función matemática que es fácil de calcular en una dirección pero extremadamente difícil de revertir, aunque algunos parámetros sean conocidos. Cuando no se utiliza la versión efímera, para la creación de la clave compartida se utilizan algunos parámetros estáticos que son parte de los certificados del cliente y servidor. Este método no es seguro, al igual que con RSA se podría realizar un ataque pasivo con el tráfico capturado cuando se obtuviesen por cualquier razón los certificados.
  * **DHE** (*Diffie-Hellman Ephimeral*): es el mismo protocolo que DH pero en este caso las claves son efímeras, esto se consigue realizando el

acuerdo sin utilizar parámetros estáticos y se consigue así el *forward secrecy*. Como la clave simétrica compartida es efímera, se evita los ataques pasivos basados en la obtención de los certificados aunque se haya capturado tráfico de una comunicación. Este método es lo mínimo que podemos considerar como seguro.

* **ECDH** (*Elliptic Curve Diffie-Hellman*): este protocolo utiliza el mismo concepto que DH pero cambiando la función matemática utilizada. El intercambio de claves se realiza utilizando una función de curva elíptica definida por el servidor, aunque en teoría el intercambio estático está soportado, en la práctica solo existen implementaciones con la variante ECDHE que crea claves efímeras y habilita el *forward secrecy*.
* **ECDHE** (*ECDH Ephimeral*): como se ha comentado, es la versión de ECDH con claves efímeras que realmente se utiliza en las implementaciones existentes [2].

- **Firma digital** (*Digital Signature*): define el algoritmo que se utilizará para la validación y autenticación de los datos intercambiados entre cliente y servidor.

  El primer paso de la firma digital es obtener y validar el certificado del servidor (en caso de que el cliente no necesite ser autenticado). Durante el intercambio de claves, se utilizará cualquiera de los métodos de autenticación citados a continuación, el cliente habrá generado el *premaster secret* utilizando el propio RSA o enviará los parámetros en el caso de (EC)DH(E) cifrados con la clave privada del servidor. Como se asume que solo este está en posesión de la clave privada, la autenticación está implícita, puesto que solo el servidor podrá conocer los parámetros utilizados:

  - **RSA**
  - **ECDSA** (*Elliptic Curve Digital Signature Algorithm*)
  - **DSS** (*Digital Signature Standard*)

- **Algoritmo de cifrado simétrico y longitud de clave** (*Cipher and key length*): estos parámetros indican el algoritmo de cifrado simétrico a utilizar, así como la longitud de clave ha utilizar del mismo. AES es el estándar de

facto de la industria, considerado seguro, el resto de algoritmos simétricos se dejan de utilizar en favor de este. Algunos de los más conocidos son:

- **AES**
- **3DES**
- **ARIA**
- **CAMELIA**
- **RC4**
- **SEED**
- **IDEA**
- **FORTEZZA**
- **BLOWFISH**

- **Modo de cifrado** (*Encryption Mode*): este parámetro (opcional) indica el modo en el que funciona el algoritmo de cifrado simétrico. Sin entrar en detalle, existen múltiples modos de cifrado para muchos algoritmos como pueden ser:
  - **CBC**
  - **GCM**
  - **ECB**
  - **CFB**
  - **CTR**
  - **PCBC**
  - **OFB**
- **Algoritmo de Hash** (*Hashing Algorithm*): este parámetro indica el algoritmo utilizado para la validación de la integridad de los mensajes para garantizar que no han sufrido alteraciones de ningún tipo. Aunque MD5 fue uno de los más utilizados (ya no se considera seguro desde hace mucho tiempo ya que se ha demostrado que se pueden producir colisiones), SHA es, a día de hoy, el estándar de facto, a partir de su versión SHA2 de más

de 256 bits se considera seguro. Algunos de los algoritmos de *Hash* más utilizados son los siguientes:

- **MD5**
- **AEAD**
- **SHA-1**
- **SHA-256**
- **SHA-384**
- **SHA-512**

3.1.2. Internet PKI: Infraestructura de clave pública

PKI (*Public Key Infrastructure*) es la infraestructura que da soporte a las comunicaciones de confianza a través de Internet, su objetivo es que dos interlocutores que no se conocen puedan autenticarse mutuamente a través de un tercero (autoridades certificadoras) en el cual ambos confían. Se van a citar los componentes más importantes para comprender PKI, y su importancia dentro de las comunicaciones seguras utilizando SSL/TLS, pero no se profundizará en su funcionamiento dado que no es el objetivo de este trabajo.

De modo muy genérico en la Figura 3.2 se pueden apreciar las partes implicadas en el ciclo de vida de PKI. A continuación se citan los principales actores:

- **Suscriptor (*Subscriber*)**: entidad que desea proveer servicios con una cierta seguridad y requiere un certificado.

- **Autoridad de registro (*Registration authority*)**: realiza la función de validar la identidad del suscriptor antes de solicitar a la autoridad certificadora la emisión de un certificado.

- **Autoridad certificadora (*Certification authority*)**: es la encargada de emitir los certificados que confirman la identidad de los suscriptores. También son las encargadas de proveer información de revocación de certificados online actualizada, para que la parte que confía pueda validar si el certificado ha sido revocado por algún problema de seguridad.

- **Consumidor (*Relying party*)**: son los clientes que consumen el certificado. Son programas, navegadores web, sistemas operativos, etc., que deben validar la identidad del suscriptor. Esto lo consiguen utilizando los almacenes de certificados públicos de las principales autoridades certificadoras raíz que tienen los clientes almacenados localmente.

- **Certificado (*CERT*)**: es un documento digital que contiene la clave pública, información del suscriptor y la firma digital del emisor del certificado. La cadena de certificados validará de forma completa que el certificado es válido y llevará a la firma de una autoridad raíz.

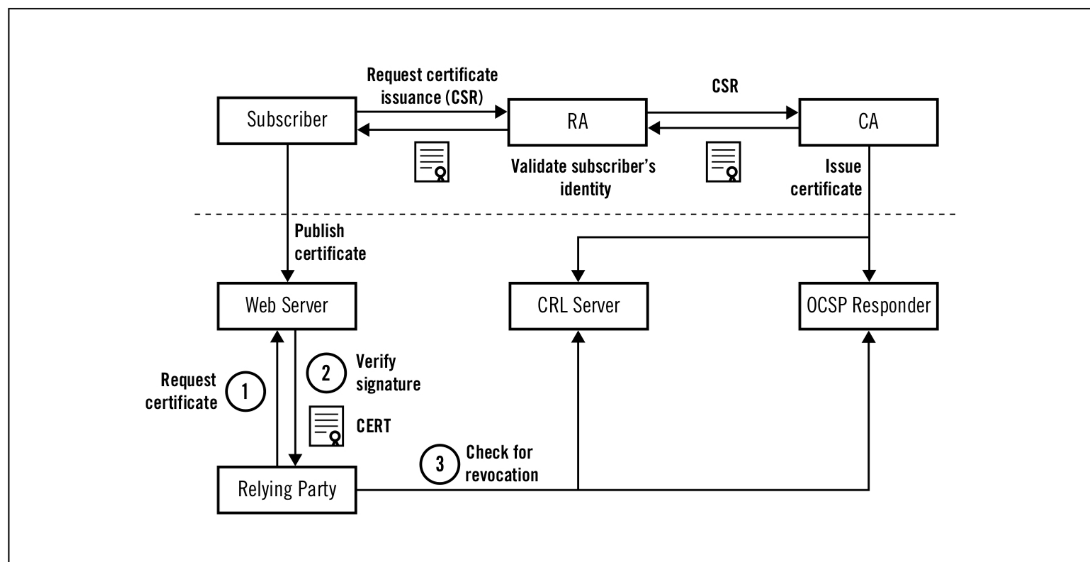


Figura 3.2: Ciclo de vida de PKI [2]

3.1.3. Limitaciones del protocolo

Debido a la capa que cubre SSL/TLS en la capa OSI, este tiene varias limitaciones de diseño [2] como pueden ser:

- El cifrado protege los contenidos de una conexión TCP (o UDP con DTLS), pero los metadatos de TCP y todas las capas inferiores viajan en texto plano. Así, un atacante pasivo puede terminar las IPs origen y destino así

como otra información adicional de capa 3 y 4. Este tipo de limitación no es de SSL/TLS sino una limitación inherente al modelo de red y la capa que ocupa.

- Incluso en la capa TLS mucha de la información viaja en texto plano y puede ser conocida. El primer *handshake* nunca viaja cifrado, permitiendo a un observador pasivo aprender las capacidades anunciadas por el cliente, examinar la información de SNI (determinando así el *virtual host* al que intenta conectarse), examinar el certificado del *host*, etc.

- Después de comenzar a cifrar la comunicación, alguna información del protocolo viaja en claro: un observador puede ver el subprotocolo y la longitud del mensaje. Dependiendo del protocolo, la longitud puede revelar información útil de la comunicación subyacente. Por ejemplo, se han realizado muchos estudios que han tratado de averiguar qué recursos estaban siendo accedidos mediante HTTP basándose en los tamaños de petición y respuesta. Sin ocultar la longitud no es posible utilizar de manera segura compresión antes del cifrado.

3.2. SSLv3

SSLv3 como se comentó en la Sección 2.1.2, fue la última versión desarrollada por Netscape antes de que el IETF se hiciera cargo del estándar, y la que más se utilizó en la época. En ella todavía se fundamentan algunos ataques a día de hoy debido a que muchas implementaciones todavía ofrecen soporte por retrocompatibilidad.

3.2.1. Características de SSLv3

3.2.1.1. Mensajes en SSLv3

Una de las principales características de SSLv3 son los mensajes que se intercambian cliente y servidor. Estos sirven para indicar el tipo de información que contiene el registro de SSL. A continuación, se citan los mensajes que existen en esta versión y su funcionalidad [3]:

- ***Alert***: informa a la otra parte de una posible brecha de seguridad o fallo en la comunicación.

- ***ApplicationData***: información que se intercambia entre las dos partes, que es cifrada, autenticada y verificada por SSL.

- ***Certificate***: mensaje que contiene el certificado con la clave pública del emisor. Este mensaje contiene la cadena de certificados, comenzando por el certificado del sistema local y acabando con el certificado de la autoridad certificadora raíz. Si el cliente no posee un certificado responde con un mensaje de alerta *NoCertificateAlert*, el servidor puede entonces continuar sin autenticar al cliente o cerrar la conexión. Además, SSl solo utiliza el certificado del cliente para asegurar la firma digital y en ningún caso para cifrar, con lo cual no existe necesidad de separar la autenticación y cifrado en el lago del cliente.

- ***CertificateRequest***: petición por parte del servidor para que el cliente envíe el certificado con su clave pública. Como la autenticación del cliente es opcional, el servidor no está obligado a autenticarlo y si esto pasa, el servidor enviará este mensaje como parte de la negociación inicial.

- ***CertificateVerify***: mensaje que envía el cliente para verificar que está en posesión de la clave privada que corresponde al certificado enviado al servidor si ha sido necesaria la autenticación del cliente. El servidor valida un *hash* calculado por el cliente después de recibir el mensaje *ClientKeyExchange*.
- ***ChangeCipherSpec***: mensaje que indica que se puede comenzar a utilizar los servicios de seguridad (como el cifrado) acordados entre ambas partes. Este mensaje lo envía el cliente una vez enviado el *ClientKeyExchange* para que el servidor se mueva de estado, conteste con el mismo mensaje al cliente y ambos comiencen a utilizar los parámetros acordados.
- ***ClientHello***: mensaje enviado por el cliente indicando los servicios de seguridad que soporta y desea utilizar. Este mensaje está compuesto de los siguientes atributos:
  - ***Version***: identifica la versión más alta de SSL soportada por el cliente, en SSLv3 el valor es 3.0. El servidor puede asumir que el cliente soporta las versiones anteriores a la citada y puede tratar de utilizar, por ejemplo, la versión 2.0, en este caso es el cliente el que debe decidir si utilizar una versión más baja o abandonar la negociación.
  - ***RandomNumber***: es un número aleatorio de 32 bytes utilizado como semilla para los cálculos criptográficos. Los primeros cuatro bytes pueden contener información de tiempo y fecha para evitar que el cliente nunca utilice el mismo valor dos veces, los 28 bytes siguientes deben ser un número aleatorio generado correctamente que no pueda ser predecible.
  - ***SessionID***: identifica la sesión SSL específica.
  - ***CipherSuites***: lista de los parámetros criptográficos que el cliente soporta. Esta lista consistirá en los *cipher suites* soportados por el cliente, aunque la decisión de cuál utilizar pertenecerá al servidor, siempre dentro de los indicados por el cliente.
  - ***CompressionMethods***: identifica los métodos de compresión que soporta el cliente. Los métodos de compresión son una parte importante

de SSL ya que el cifrado tiene consecuencias en los métodos de compresión. El cifrado cambia las propiedades de la información y debería hacer imposible la compresión, por esta razón, la compresión debería preceder al cifrado. En SSLv3 este campo no se utiliza porque no se definieron métodos de compresión, fue creado para futuras versiones del protocolo.

- ***ClientKeyExchange***: mensaje enviado por el cliente que contiene las claves criptográficas para la comunicación. El cliente envía este mensaje cuando el servidor ha finalizado la parte inicial de la negociación. El mensaje contiene la información para el cifrado simétrico, que el cliente cifra utilizando la clave pública que el servidor le ha enviado. Es importante que este mensaje viaje cifrado puesto que contiene la clave de sesión y solo el servidor puede descifrarlo utilizando su clave pública.

- ***Finished***: mensaje que indica que la etapa de negociación ha finalizado y que la comunicación es segura. Este mensaje es enviado por ambas partes una vez han finalizado con el *ChangeCipherSpec*. Además, contiene un *hash* que solo ambas partes pueden conocer y que debe coincidir.

- ***HelloRequest***: petición enviada por el servidor para que el cliente comience o reinicie el proceso de negociación de SSL.

- ***ServerHello***: mensaje enviado por el servidor indicando los servicios de seguridad que se van a utilizar en la comunicación. Este mensaje está compuesto de los siguientes atributos:

  - ***Version***: a diferencia de la versión enviada en el mensaje *ClientHello*, este determina la versión de SSL que la comunicación va a usar. El servidor puede escoger una versión anterior a la que el cliente indica, nunca superior.

  - ***RandomNumber***: este parámetro es el mismo que en el mensaje *ClientHello* pero generado por el servidor.

  - ***SessionID***: identifica la sesión SSL específica para reutilizar una existente. El servidor puede escoger si crear una sesión para que el cliente

trate de reutilizarla más adelante, o no soportarlo y no enviar este ID de sesión.

- ☐ ***CipherSuites***: escoge el *cipher suite* que se va a utilizar en la comunicación, siendo uno de los que el cliente ha incluído en el *ClientHello*.
- ☐ ***CompressionMethods***: tiene las mismas características que en el *ClientHello*.

- ***ServerHelloDone***: mensaje enviado por el servidor que indica que ha completado todas las peticiones del cliente para establecer la comunicación segura. El mensaje en sí no contiene información, pero es importante para que el cliente cambie de fase en el establecimiento de la conexión segura.

- ***ServerKeyExchange***: mensaje enviado por el servidor que contiene las claves criptográficas para la comunicación. Este mensaje complementa el campo *CipherSuite* del *ServerHello*. Este mensaje contiene la clave pública del servidor. La información específica depende del algoritmo de clave pública utilizado. Este mensaje es enviado sin cifrado, por lo cual solo la información pública puede ser enviada en este tipo de mensajes. El cliente utilizará posteriormente la clave pública del servidor para cifrar la clave de sesión (suponiendo RSA).

3.2.1.2. *Handshake* en SSLv3

Una vez explicados los mensajes que utiliza SSL se va a revisar el establecimiento de una sesión SSL al completo.

SSL utiliza una capa de abstracción para la construcción de los mensajes y su encapsulamiento posterior en segmentos TCP. Esta capa es conocida como *Record Layer* y provee de un formato común para los mensajes que ambas partes se intercambian. Esta capa consiste en 5 bytes que preceden a los mensajes del protocolo y la comprobación de integridad (si se está utilizando) al final del mensaje. Es además, responsable del cifrado si el servicio está activo. La Figura 3.3 muestra la estructura del formato de la *Record Layer* [3].

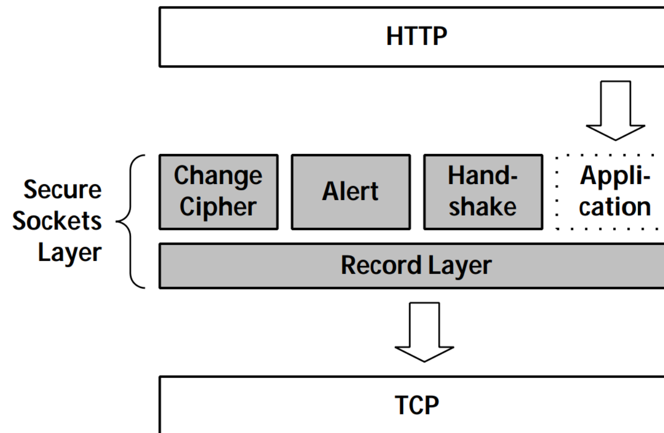


Figura 3.3: Formato de la *Record Layer*

SSL no existe como un protocolo aislado, si no que depende de un protocolo de capa de transporte que sea confiable, debe garantizar la retransmisión correcta de los mensajes sin errores. SSL utiliza TCP para esta labor.

En la Figura 3.4 se pueden apreciar los distintos segmentos que se utilizarían para el establecimiento de la sesión SSL típica. A estos mensajes le precede el *Three-way Handshake* clásico de TCP para el establecimiento de la conexión de capa de transporte.

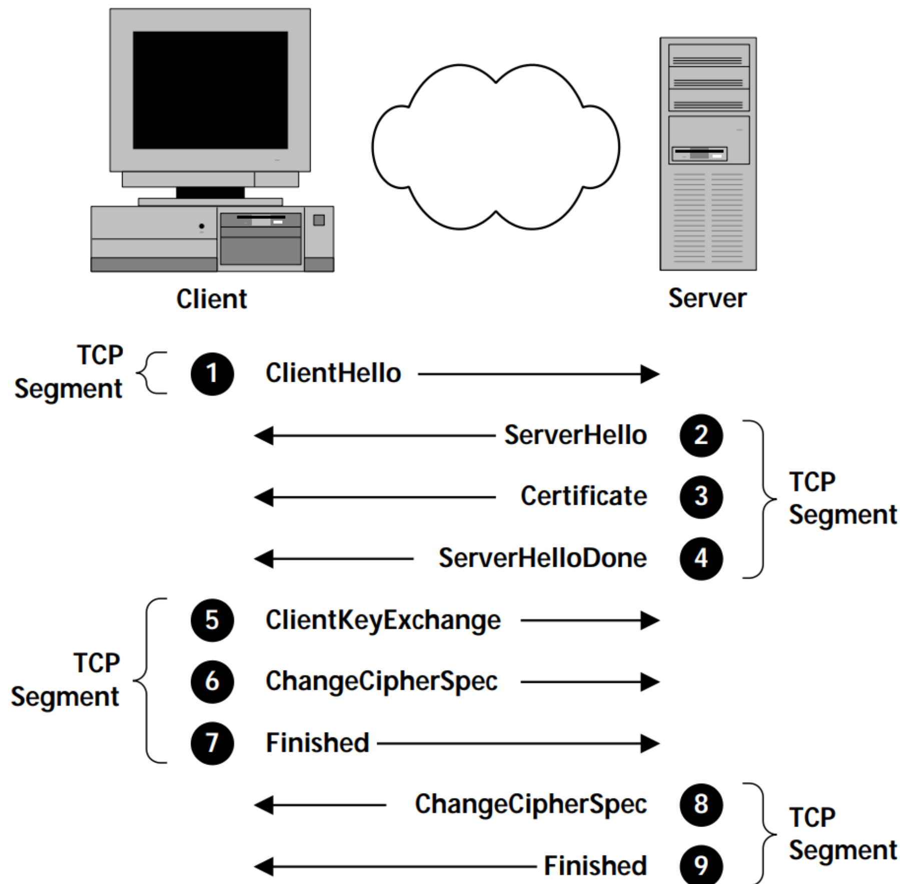


Figura 3.4: *Handshake* de SSL utilizando TCP

Los pasos de establecimiento de la sesión SSL serían los siguientes:

1. Cliente y servidor establecen una sesión TCP.

2. El cliente envía un mensaje ***ClientHello*** indicando la versión máxima de SSL que soporta, el número aleatorio utilizado de semilla, el *SessionID* vacío (puesto que no existía una sesión anterior), los *cipher suites* que soporta y sin métodos de compresión (SSLv3 no los soportaba).

3. El servidor responde con un mensaje ***ServerHello*** indicando la versión de SSL que ha seleccionado para la sesión, su número aleatorio de semilla, el ID de sesión (si decide crearlo), el *cipher suite* que ha seleccionado y sin métodos de compresión.

4. El servidor envía con un mensaje ***Certificate*** su certificado para que el cliente pueda comprobar su autenticidad y si es quién dice ser.

5. El servidor envía un mensaje ***ServerHelloDone*** para indicar que ha finalizado su mensaje de inicio.

6. El cliente envía un mensaje ***ClientKeyExchange*** que contiene la clave compartida de sesión generada por el mismo a partir del algoritmo de cifrado simétrico que el servidor ha escogido e indicado en el *cipher suite* selecionado. Esta clave es cifrada con la clave pública del servidor para que solo este pueda saber su valor.

7. El cliente envía un mensaje ***ChangeCipherSpec*** para que el servidor sepa que debe cambiar su estado y comenzar a utilizar el cifrado simétrico y la integridad seleccionada.

8. El cliente envía un mensaje ***Finished*** que indica que ha finalizado la negociación y la comunicación para él ya es segura.

9. El servidor envía un mensaje ***ChangeCipherSpec*** para que el cliente sepa que el servidor cambia de estado para utilizar los métodos de cifrado seleccionados.

10. Por último, el servidor envía un mensaje ***Finished*** que indica que ha finalizado la negociación y la comunicación para él ya es segura.

Hay que matizar que en este *Handshake* se han obviado los mensajes referentes a la autenticación del cliente, en navegación web no suele ser el caso y el servidor no suele autenticar al cliente. Si fuese necesario se añadirían los pasos que se pueden observar en la Figura 3.5, y cuyos mensajes han sido explicados en la Sección 3.2.1.1.

A modo de resumen de lo que sucedería: el servidor solicitaría el certificado, el cliente lo enviaría antes del *ClientKeyExchange*, enviaría el mensaje *CertificateVerify* para demostrar que está en posesión de la clave privada y posteriormente el servidor realizaría las comprobaciones antes de finalizar el establecimiento de sesión [3].

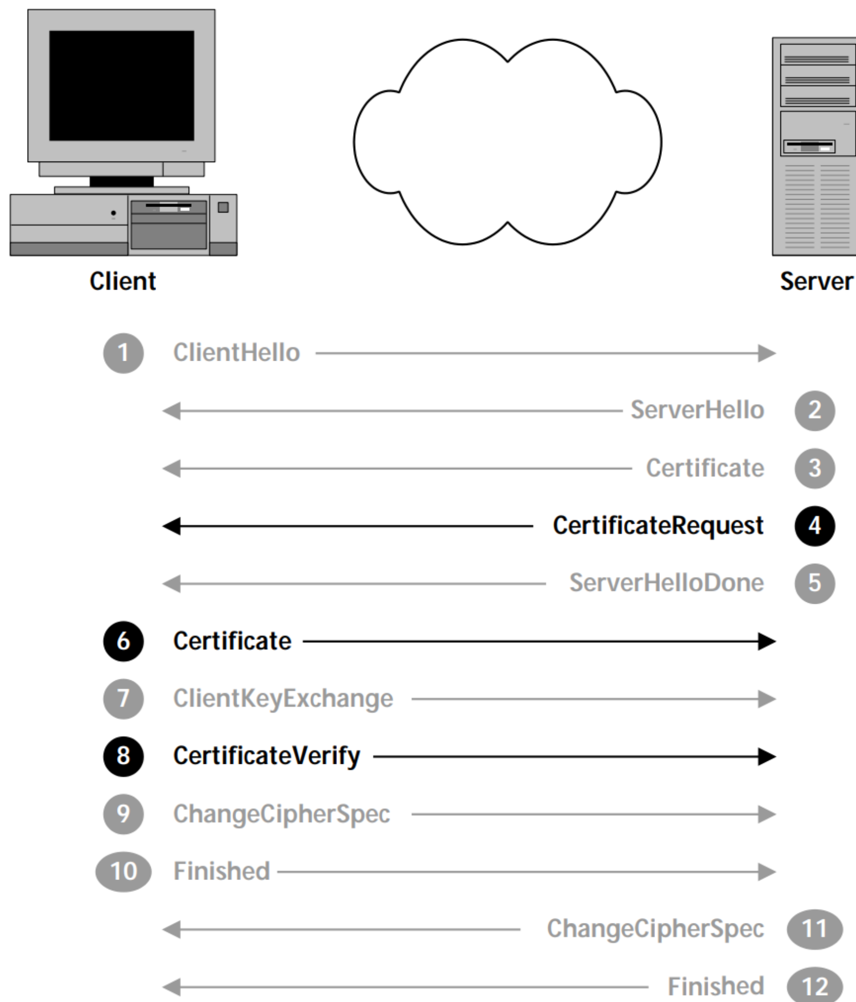


Figura 3.5: *Handshake* SSL con autenticación del cliente [3]

3.2.2. Fallos de seguridad de SSLv2 y SSLv3

Como resultado del éxito de SSLv3, muchos investigadores se fijaron en este protocolo desde su surgimiento.

En 1996 con la publicación del navegador de Netscape que soportaba esta versión, David Wagner y Ian Goldberg demostraron que el método utilizado para la generación pseudoaleatoria de la semilla para la generación del *premaster secret* era criptográficamente débil puesto que podía ser predicho. El problema radicaba

en que la semilla era derivada de unos valores determinísticos: el id del proceso, el id del proceso padre y el tiempo actual; estos valores no proveen suficiente entropía como para ser considerados seguros. Este no es un problema del protocolo, sino que, como en muchas ocasiones, es un problema de la implementación por parte de Netscape. Este problema fue resuelto rápidamente por parte de Netscape, pero es un ejemplo de que en muchas ocasiones las implementaciones de los protocolos introducen problemas que el estándar no abarca.

En este mismo año, Wagner y Bruce Schneier fueron los primeros en realizar un análisis de seguridad del protocolo SSL para la versión 2.0 y 3.0. El resultado fue la demostración de la posibilidad de realizar ataques contra la versión 2.0. Un ejemplo es que la autenticación MAC no protegía el tamaño de relleno (*padding length*) y esto podía provocar ataques contra la integridad de los datos protegidos. Además, ante la falta de autenticación e integridad en algunos *cipher suites* incluídos, un mensaje *ClientHello* podía ser explotado en un ataque de *rollback*, en el cual el atacante modifica este mensaje para incluír únicamente *cipher suites* débiles. En este ataque, como el servidor debe escoger únicamente uno de los *cipher suites* enviados por el cliente, no tiene más opción que utilizar un conjunto de cifrado débil. Debido a múltiples de estos fallos de seguridad, el IETF publicó en 2011 un RFC prohibiendo el uso de SSLv2.0 completamente. Muchos de estos fallos fueron corregidos en SSLv3.0 pero se debe prohibir el uso de SSLv2 para que no se puedan realizar ataques de bajada de versión en las versiones siguientes. En su estudio Wagner y Schneier consideraron en la época que el protocolo SSLv3 tenía suficiente seguridad [17].

En 1998 Daniel Bleichenbacher encontró fallo en la forma en que el *premaster secret* es rellenado antes de que RSA lo cifre en un mensaje *ClientKeyExchange*. Según la especificación de SSLv3.0 este relleno y cifrado debe realizarse mediante PKCS#1 versión 1.5. Bleichenbacher encontró una manera de crear un ataque contra el relleno PKCS#1 y la codificación del *premaster secret*. Si el atacante es capaz de descifrar este secreto, entonces puede averiguar todo el material de clave derivado y romper la sesión SSL. El resultado del ataque de Bleichenbacher fue el primer ejemplo de lo que ahora se llama *padding oracle attack*, un ataque que utiliza la validación del relleno de un mensaje para conseguir descifrar el texto

cifrado. El problema del ataque de Bleichenbacher es que el *oracle attack* revela únicamente un bit de información por petición. Esto significa que se necesita una gran cantidad de peticiones de *oracle* (sobre un millón), y también es conocido como el ataque de un millón de mensajes (*million message attack*). Este ataque puede ser prevenido, el servidor no debe filtrar información sobre si el *padding* es o no correcto. La manera más simple de realizar esto es que al obtener un *padding* incorrecto el servidor no devuelva ningún error, servidor y cliente acabarán con claves distintas y la sesión no será establecida pero no se filtrará el error. Después volverán a comenzar la negociación y establecerán la conexión cuando no existan errores [18].

Más adelante, en 2001 James Manger encontró otro ataque contra las implementaciones de PKCS#1 versión 2.0, con lo que surgió en 2003 la versión 2.1 de PKCS#1.

3.3. TLS

TLS es el nombre que el IETF dio al estándar una vez se hicieron cargo del mismo, como ya se comentó en la Sección 2.1.2 este nombre marca el fin de la propiedad sobre el protocolo de Netscape. Esta versión del estándar está ampliamente basado en SSLv3.0 sin tener apenas modificaciones. Es el incremento de versión que menos cambios ha sufrido. Como se ha comentado esto fue un simple cambio de nombre, ya que el protocolo se mantuvo el mismo y hubo que mantener la interoperabilidad en varios aspectos. Un ejemplo de esto es que la versión de protocolo que introduce es 3.1 para incrementar la versión 3.0 que introducía SSLv3.

3.3.1. Características de TLSv1.0

Al ser TLSv1.0 una pequeña mejora de SSLv3, se van a comentar las mejoras y diferencias introducidas, sin tratar de volver a explicar todo lo ya comentado en la Sección 3.2.1 que continúa aplicando a esta versión del estándar.

- **Versión**: para mantener la interoperabilidad, TLSv1.0 introduce en los mensajes la versión de protocolo 3.1, para diferenciarse de las versiones menores anteriores.

- **Mensajes de alerta**: TLS amplía la cantidad de mensajes de alertas de seguridad que cliente y servidor pueden intercambiar, a modo de resumen, SSLv3 definía 12 tipos de mensajes y TLSv1.0 amplía estos mensajes hasta 23 tipos.

- **Autenticación de mensajes**: TLS mejora este apartado, puesto que la autenticación de mensajes en SSL era un método *ad hoc*. TLS introduce el uso del estándar *hmac* (*Hashed Message Authentication Code*). La especificación *hmac* incluye una descripción precisa y rigurosa del proceso de autenticación, así como código de ejemplo. La especificación no indica qué algoritmo en particular de *hash* se debe utilizar, se puede utilizar cualquier algoritmo competente (md5 o sha en aquel momento). Los datos que protege TLS mediante *hmac* son:

  - *Sequence number*

- ☐ *TLS protocol message type*
- ☐ *TLS version (3.1)*
- ☐ *Message length*
- ☐ *Message contents*

En la Figura 3.6 se puede observar el funcionamiento de *hmac* para realizar la autenticación de mensajes.

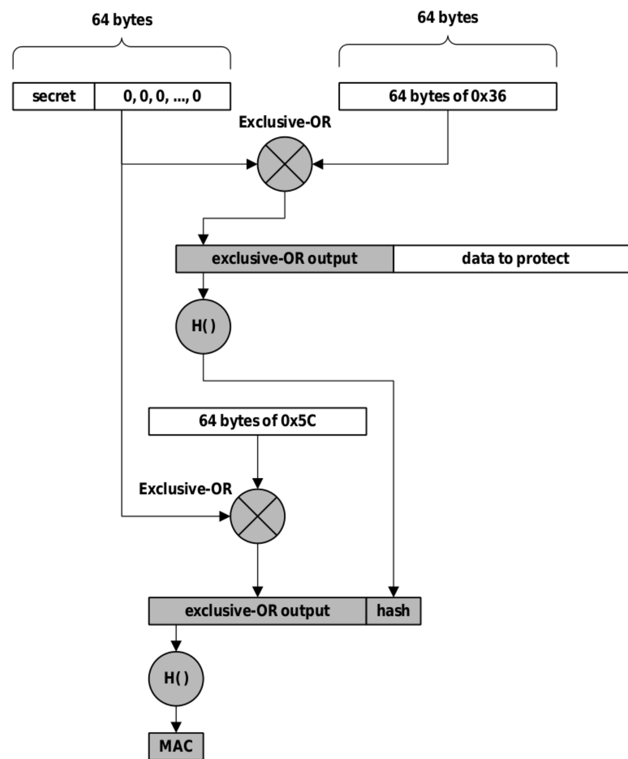


Figura 3.6: Funcionamiento de HMAC [3]

- **Material de generación de claves**: TLS define un procedimiento para usar *hmac* como creador de salida pseudoaleatoria. Este proceso toma un valor secreto y una semilla, y los mezcla durante varias iteraciones para generar una salida aleatoria. Cualquier algoritmo de *hashing* puede ser utilizado para este procedimiento. Además, se define un método adicional, una función pseudoaleatoria o PRF (*PseudoRandom Function*) que mezcla la aplicación del proceso pseudoaleatorio con dos algoritmos distintos (md5

y sha) y los combina, para que en caso de que uno de los algoritmos sea descubierto inseguro, el proceso siga teniendo una cierta seguridad gracias al uso del otro. La función divide el secreto en dos partes iguales, genera la salida de los algoritmos de *hash* para cada parte, la primera con md5 y la segunda con sha combinadas con una etiqueta y una semilla, para finalizar sobre el resultado de ambas realiza un XOR y se obtiene el resultado final. En la Figura 3.7 se pueden apreciar los pasos anteriormente descritos para PRF.

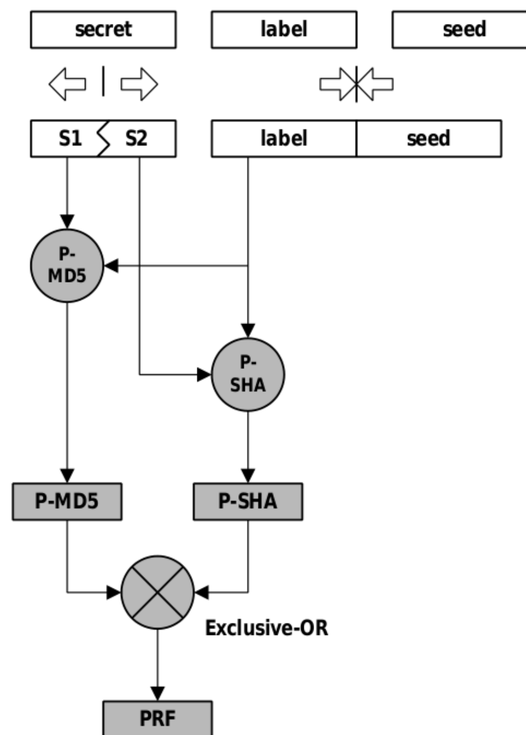


Figura 3.7: Función *PseudoRandom Function* utilizada en TLSv1.0 [3]

Para la generación del material de claves TLS utiliza el mismo método que SSL. Cada sistema empieza con la *premaster key* y después se crea el *master secret*. A continuación, genera el material de las claves a través de el *master secret* utilizando PRF. La entrada de PRF es el *master secret* (como secreto), la cadena en ASCII *'key expansion'* (como etiqueta) y la concatenación del valor aleatorio generado por el servidor y el cliente (como

semilla). Además, el *master secret* de 48 bytes también es calculado con PRF, en este caso las entradas son el *premaster secret* (como secreto), la cadena ASCII *'master secret'* (como etiqueta) y la concatenación de los valores aleatorios creados por el cliente y servidor. En la siguiente Figura 3.8 se puede apreciar el proceso descrito anteriormente para la generación del material de claves y el *master secret*.

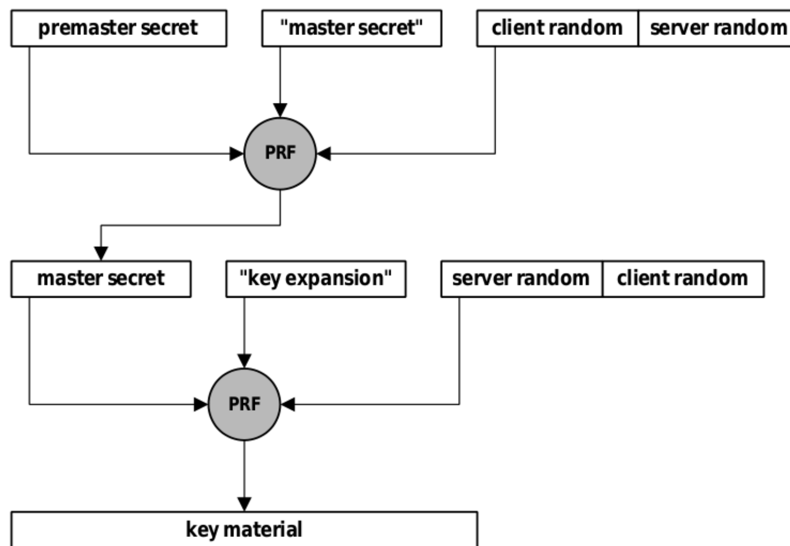


Figura 3.8: Uso de PRF para crear el *master secret* y el material de claves [3]

- *Finished*: TLS simplifica los contenidos de este mensaje, cuyo único contenido es la aplicación de PRF con los siguientes parámetros: el secreto *master secret*, las etiquetas *'client finished'* (para cliente) o *'server finished'* (para servidor), y por último la concatenación de los *hashes* md5 y sha de todos los mensajes de *handshake*.

3.3.2. Características de TLSv1.1

La versión 1.1 de TLS fue publicada oficialmente en 2006, siete años después de que TLSv1.0 viese la luz. Siguiendo la notación de versión de SSL, dentro del propio *handshake* podría ser visto como la versión 3.2.

Las principales diferencias respecto a TLSv1.0 son las siguientes:

- El cifrado CBC (*Cipher Block Chaining*) ahora utiliza un vector de inicialización que es incluido en cada registro TLS. Esto trata de resolver la debilidad de poder predecir el vector de inicialización y que fue explotado por el ataque BEAST.
- Las implementaciones del protocolo ahora deben utilizar la alerta bad_record_mac en respuesta a problemas de relleno como defensa contra los ataques de *padding*. La alerta decryption_failed queda obsoleta.
- Esta versión incluye las extensiones TLS (RFC 3546) [19].
- TLSv1.1 también introduce una nueva manera de extender el protocolo sin necesidad de pasar por revisiones de RFC, la IANA (*Internet Assigned Numbers Authority*) decidió añadir la flexibilidad de modificar registros para parámetros de TLS, como los tipos de certificados, *cipher suites*, tipos de mensajes de alerta, *handshake*, etc. Si cualquiera de estos parámetros necesita ser modificado o añadido, no es necesaria la publicación de un nuevo RFC, estos parámetros serán modificados en los respectivos registros.

3.3.2.1. Extensiones

Las extensiones TLS tienen una gran importancia al proporcionar el mecanismo de añadir funcionalidades al protocolo sin necesidad de modificar el mismo. Estas extensiones son muy importantes a día de hoy puesto que son ampliamente utilizadas y su funcionalidad se ha convertido en algo totalmente necesario en algunos protocolos de capa de aplicación como puede ser HTTP.

Las extensiones se introducen en forma de bloque al final de los mensajes *ClienHello* y *ServerHello* añadidas una detrás de otra. Cada extensión comienza con campo de 2 bytes que indica el tipo de extensión y a continuación los datos de

la misma. El formato de la extensión y su funcionalidad depende la especificación de las mismas. A continuación se citarán las extensiones más importantes y utilizadas [2]. La lista completa de las extensiones TLS más importantes pueden ser encontradas en la página web de la IANA que mantiene un registro oficial [20].

- ***Application Layer Protocol Negotiation***: es una extensión que posibilita la negociación de diferentes protocolos de capa de aplicación sobre una conexión TLS. Un servidor web podría ofrecer con TLS el protocolo HTTP/1.1 por defecto, y permitir la negociación de otros como SPDY o HTTP/2. Un cliente que soporta la extensión ALPN usa la extensión *application_layer_protocol_negotiation* en el *ClientHello* para listar los protocolos de capa de aplicación que soporta e informar así al servidor. Si el servidor lo soporta elegirá el protocolo que desee y en caso de no soportarlo simplemente ignoraría esta extensión.

  Esta extensión provee la misma funcionalidad que NPN *Next Protocol Negotiation* diseñada por Google en el momento de la creación de SPDY para indicar el protocolo que transportaba TLS. Este no fue adoptado porque ocultaba bajo el cifrado los datos de protocolo que se estaba utilizando y esto podría interferir en los intermediarios por donde pasa el tráfico, como pueden ser *firewalls o proxies.*

- ***Elliptic Curve Capabilities***: el RFC 4492 especifica dos extensiones que son utilizadas para comunicar las capacidades de utilizar algoritmos de curva elíptica por parte del cliente. El nombre de la extensión en el *ClientHello* es *elliptic_curves* y contiene una lista de los algoritmos que el cliente soporta. Otros RFCs han ampliado la cantidad de algoritmos de curva elíptica y es de esperar que conforme vayan apareciendo más se integren en esta extensión para indicar el soporte.

  Una segunda extensión es *ec_point_formats* que se utiliza para negociar el punto de compresión óptimo de la curva elíptica. En teoría esta extensión podría ayudar a ahorrar ancho de banda consumido pero generalmente no se utiliza.

- ***Heartbeat***: es una extensión creada para añadir soporte a las funcionalidades de *keep-alive*, que de manera general trata de comprobar si el otro

interlocutor sigue disponible en una comunicación. Además, añade soporte al descubrimiento de la unidad máxima de transmisión en el canal (PMTUD *Path MTU Discovery*) para TLS. Realmente TLS es utilizado en la mayor parte de las comunicaciones sobre TCP, el cual ya tiene funcionalidades de *keep-alive*, pero *Heartbeat* está destinado a DTLS que utiliza UDP como capa de transporte.

Se ha estudiado esta extensión y realmente no sería necesaria, puesto que se podrían utilizar registros TLS con tamaño cero, que son explícitamente permitidos por el protocolo y podrían ser utilizados como método de *keep-alive*. En la práctica, los intentos de mitigar BEAST han demostrado que multitud de aplicaciones no toleran registros sin ningún dato. Esta técnica tampoco podría ayudar con el PMTUD [2].

La extensión se incluye en los paquetes *hello* de cliente y servidor para anunciar su soporte durante la negociación.

Heartbeat está implementado como un subprotocolo de TLS, esto significa que los mensajes de *heartbeat* pueden intercalarse con los datos de la aplicación o cualquier mensaje del protocolo. De acuerdo al RFC, los mensajes de *heartbeat* son permitidos únicamente cuando el *handshake* ha finalizado, sin embargo en la práctica librerías como OpenSSL lo permiten desde que las extensiones han sido intercambiadas. Librerías como OpenSSL y GnuTLS implementan esta extensión y la habilitan por defecto. Esta extensión pasó desapercibida hasta que en abril de 2014 se descubrió un fallo muy grave en la implementación de OpenSSL que permitía la extracción de datos sensibles de la memoria del servidor. Este fallo ha sido uno de los peores en la historia de TLS por lo fácil de su explotación, se revisará en la Sección 3.3.4.4.

- ***Secure Renegotiation***: esta extensión (*renegotiation_info*) fue introducida para mejorar la identificación de que la renegociación de la sesión TLS estaba siendo ejecutada por ambas partes que negociaron el *handshake* anterior. Durante la primera negociación del *handshake* esta extensión es utilizada por ambas partes para indicar que soportan la renegociación, simplemente introducen *renegotiation_info* sin datos. En SSL3, el cual no soporta extensiones, los clientes pueden utilizar el *cipher suite* especial

TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0xff) para obtener el mismo resultado.

En los *handshakes* siguientes, la extensión es utilizada para enviar datos que prueban la existencia de los *handshakes* anteriores. El cliente envía el valor del *verify_data* del mensaje *Finished* anterior y el servidor envía este mensaje del cliente junto con su propio campo *verify_data* que solo podrían ser conocidos por él, al viajar por la red cifrados.

- ***Server Name Indication***: esta extensión provee un mecanismo para que el cliente especifique el nombre del servidor al que se desea conectar. Esto puede ser importante en ámbitos como la navegación web, cuando un cliente trata de conectarse a un *virtual host* de un servidor y este debe discernir entre los dominios que tiene alojados y servir el certificado y posteriormente la web adecuada. Sin este mecanismo únicamente se podría servir un certificado por dirección IP. Debido a que SNI se añadió a TLS en 2006, hay *software* antiguo que no lo soporta.

- ***Session Tickets***: con los *tickets* de sesión se introduce un mecanismo de recuperación de una sesión existente sin necesidad de almacenamiento en el lado del servidor. La idea detrás de esta extensión es crear un mecanismo con el cual el servidor guarda la información de la sesión, la cifra y la envía al cliente en forma de *ticket*. En conexiones futuras, el cliente enviará este *ticket* al servidor, este comprobará la integridad, descifrará el contenido y utilizara la información del *ticket* para resumir la sesión sin necesidad de una nueva (y costosa) negociación de todos los materiales de clave. Además, esta aproximación ayuda a que los *clusters* de servidores web no tengan que sincronizar las sesiones entre nodos y trasladen esta responsabilidad al cliente.

  Los clientes indican si soportan este mecanismo enviando la extensión *session_ticket* vacía en el *ClienHello*, un servidor que soporte esta extensión introducirá el mismo campo vacío en el *ServerHello*. El servidor después del mensaje *Finished* del cliente, lo verifica y envía un mensaje de *NewSessionTicket*. En una sesión posterior, si el cliente desea reutilizar el *ticket* simplemente debe introducir la información en esta extensión en el siguiente *ClientHello* y el servidor tratará de resumir la sesión si es posible mediante

un *handshake* abreviado.

Cuando un servidor decide utilizar *session tickets* para el reestablecimiento de sesiones, envía el campo *session ID* vacío en el *ServerHello*. De esta manera la sesión no tiene un identificador único. Sin embargo, la especificación de la extensión permite al cliente seleccionar y enviar un *session ID* en el *ClientHello* en *handshakes* subsecuentes que utilice el *ticket*. Un servidor que acepta el *ticket* debe además responder con el mismo *session ID* [21].

Hay que matizar que los *session tickets* rompen el modelo de seguridad de TLS, ya que exponen el estado de una sesión y envían los datos de clave cifrados por la red, que pueden ser reutilizados en un futuro cuando se descubra alguna vulnerabilidad o simplemente exista suficiente capacidad computacional para romper el algoritmo simétrico con el cual ha sido cifrado. Con el *ticket* y el tráfico capturado se podría recuperar toda la información de esa conexión pasada. Esto rompe la filosofía de *perfect forward secrecy*. Por esta razón, si se utilizan los *tickets* de sesión, deben ser rotados con frecuencia.

- ***Signature Algorithms***: esta extensión definida realmente en TLSv1.2, posibilita a los clientes la comunicación de varios algoritmos de firma e integridad. La especificación TLS lista RSA, DSA y ECDSA como algoritmos de firma y MD5, SHA1, SHA224, SHA256, SHA384 y SHA512 como funciones de *hash*. Utilizando la extensión *signature_algorithm* los clientes pueden enviar los pares de firma-integridad que soportan. Si esta extensión no está presente el servidor inferirá los algoritmos soportados de los *cipher suites* que el cliente dice soportar.

3.3.3. Características de TLSv1.2

La versión 1.2 de TLS fue publicada oficialmente en 2008, dos años después de TLSv1.1. Siguiendo la notación de versión de SSL, dentro del propio *handshake* podría ser visto como la versión 3.3.

Las principales diferencias respecto a TLSv1.1 son las siguientes:

- Se añade soporte para *cipher suites* que utilicen HMAC-SHA256.
- Los *cipher suites* que utilizan IDEA y DES son eliminados.

- Las extensiones TLS son incorporadas a la especificación principal del protocolo, aunque muchas siguen documentadas en otros sitios.
- Se añade la extensión *signature_algorithms* para que los clientes comuniquen los algoritmos de firma y *hash* que aceptan.
- La combinación de MD5 y SHA1 utilizada en PRF (explicado en la Sección 3.3.1) se reemplaza por SHA256 para TLSv1.2.
- En esta versión, los *cipher suites* pueden especificar sus propios PRFs.
- La combinación de MD5 y SHA1 utilizada para las firmas digitales es reemplazada por un único *hash*. Por defecto, SHA256 es el algoritmo utilizado pero se puede escoger otro. Anteriormente el algoritmo de resumen para la firma era escogido por el protocolo.
- La longitud del elemento *verify_data* en el mensaje *Finished* puede ser especificado explícitamente por el *cipher suite*.

3.3.4. Fallos de seguridad de TLSv1.0-1.2

En esta sección se van a agrupar las vulnerabilidades que han afectado a alguna de las versiones de TLS hasta la actualidad. Existen multitud de vulnerabilidades del protocolo o fallos en las distintas implementaciones, pero a continuación se detallarán las que más impacto han tenido por su gravedad y su difusión.

3.3.4.1. BEAST (Browser Exploit Against SSL/TLS)

Esta vulnerabilidad fue descubierta en verano de 2011 por Duong y Rizzo. Consiste en un ataque que puede ser utilizado contra TLSv1.0 y anteriores para extraer datos cifrados de manera lenta. Este fallo ataca la debilidad de que el vector de inicialización (IV) puede ser predecible en TLSv1.0. El fallo que no parecía explotable en la práctica fue corregido en TLSv1.1, pero en el momento del descubrimiento ningún *software* daba soporte esta nueva versión.

Este ataque es un *exploit* cuyo objetivo es el cifrado CBC (*CIpher Block Chaining*) utilizado en TLSv1.0 y anteriores versiones del protocolo. El problema con el vector de inicialización es que es predecible y esto reduce el modo CBC a ECB

(*Electronic Code Book*) que es inseguro. ECB es el modo más simple de operación, se dividen los datos de entrada en bloques y se cifra cada bloque individualmente. El principal problema de este modo es que no oculta la naturaleza determinística de los cifrados de bloque, cada vez que se cifra la misma entrada, la salida vuelve a ser la misma, lo cual es una ventaja clara para el atacante. La principal diferencia entre ECB y CBC es que este último utiliza un vector de inicialización para enmascarar cada mensaje antes de cifrarlo, el objetivo de este método es ocultar patrones ya que la salida siempre será distinta incluso con la misma entrada. Para que el vector de inicialización sea efectivo, este debe ser impredecible en cada mensaje. La manera de alcanzar este objetivo podría ser generar un bloque de datos aleatorios para cada bloque que se desee cifrar, pero esto en la práctica lastraría el rendimiento porque doblaría la salida de datos. En la práctica, CBC en SSLv3 y TLSv1.0 utiliza solo un bloque de datos aleatorios al principio. A partir de ese momento la versión cifrada del bloque actual es utilizada como vector de inicialización del siguiente, de ahí el nombre *chaining* (encadenamiento). Este enfoque es seguro siempre y cuando un atacante no pueda influenciar que va a ser cifrado justo después de un determinado bloque, ya que si no, simplemente observando los datos de un bloque sería capaz de saber que vector de inicialización se utilizaría en el siguiente.

Desafortunadamente para TLSv1.0, este trata toda la conexión como un único mensaje y utiliza un vector de inicialización aleatorio solo para el primer registro TLS. Todos los subsiguientes utilizan el último bloque como vector de inicialización y esto lleva al atacante a poder conocer todos los vectores de inicialización a partir del segundo bloque.

El la Figura 3.9 se puede apreciar el modo CBC con un vector de inicialización predecible, se envían dos bloques por parte de la víctima y el tercero inyectado por el atacante. Después de ver los dos primeros bloques, el atacante va realizando sus pruebas y observando la versión cifrada de la respuesta, como conoce todos los vectores de inicialización, puede realizar pruebas de tal manera que los efectos de los vectores de inicialización sean eliminados. Cuando, por ejemplo, una prueba (C3) sea exitosa creará el mismo bloque que la víctima (C2) [2].

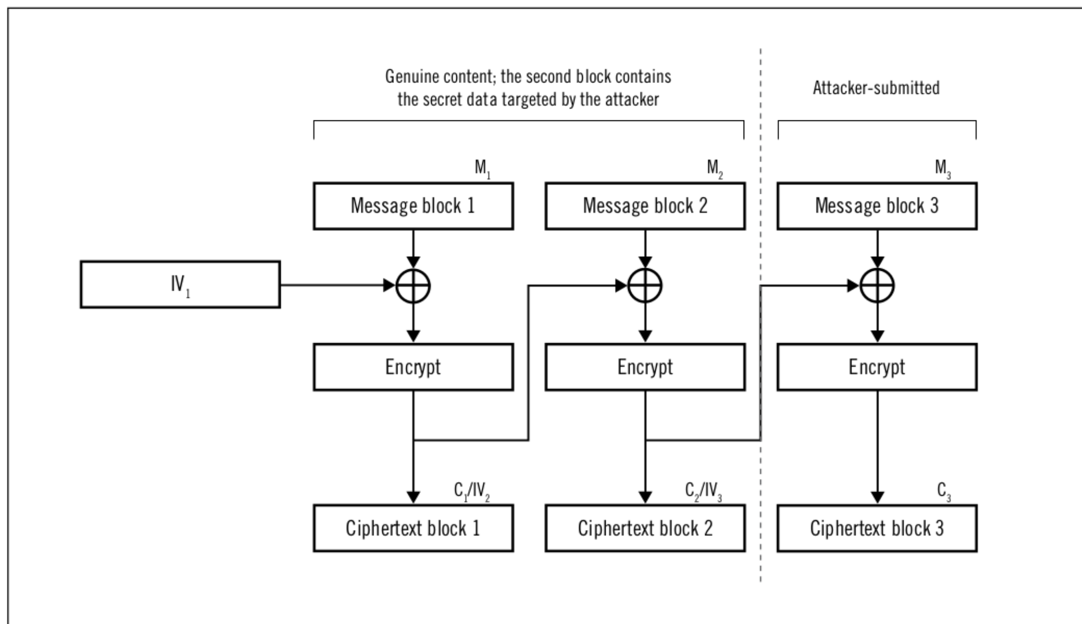


Figura 3.9: Ataque BEAST contra CBC con un IV predecible [2]

Este tipo de ataque parece difícil de trasladar a un escenario práctico pero HTTP otorga al atacante varias optimizaciones:

- Los mensajes HTTP contienen pequeños fragmentos de datos sensibles, como pueden ser contraseñas o *tickets* de sesión. A veces averiguando solo 16 bytes (un bloque) es suficiente para un ataque exitoso.

- Los datos sensibles típicamente utilizan conjuntos de datos restringidos, por ejemplo, datos codificados en hexadecimal que solo pueden tomar 16 valores.

- La estructura de una petición HTTP es muy predecible, los datos sensibles estarán mezclados con otros datos conocidos. Por ejemplo, la cadena *Cookie:* siempre será posicionado antes del nombre de la primera *cookie*.

Aún así, para llevar a cabo este ataque, es necesario el control del navegador de la víctima que está siendo cifrado y enviado, y además influenciar la posición del secreto en la petición. Los investigadores utilizaron *exploits* en *Applets* Java para llevar a cabo este ataque aunque esto último no es necesario. En la práctica

BEAST es un ataque de red activo y que necesita de ingeniería social para enviar a la víctima a sitios que contengan código malicioso (p.ej. Javascript) para llevar a cabo el ataque.

BEAST es una vulnerabilidad del lado del cliente que requiere contramedidas en esta parte. En 2004 cuando surgió el problema, OpenSSL intentó corregir el problema introduciendo un registro TSL sin datos antes de uno real, a pesar de que en aquel momento no había forma práctica de explotar la vulnerabilidad. Esta solución no funcionó como se esperaba puesto que múltiples clientes(Internet Explorer sobre todo) reaccionaban mal a registros TLS con tamaño cero. En 2011, los navegadores mitigaron BEAST utilizando una variante de este método de registros vacíos. Esta técnica utiliza dos registros en vez de uno pero el primero llevará un byte de datos y el segundo todo lo demás. Un byte sigue siendo expuesto al vector de inicialización inicial, pero como el bloque de cifrado utiliza como mínimo 7 bytes a mayores el resto van a ser aleatorios y diferentes en cada registro, con lo que el atacante no va a poder averiguarlos.

3.3.4.2. CRIME (Compression Ratio Info-leak Made Easy)

Esta vulnerabilidad descubierta por Duong y Rizzo (al igual que BEAST) es un ataque a la compresión en TLS que permitiría extraer las *cookies* de un cliente con un ataque MITM (*Man in the middle*) utilizando JavaScript. Fue expuesta en septiembre de 2012. El mecanismo de CRIME es el mismo que el de BEAST, se instrumentaliza el navegador de la víctima para emitir peticiones al servidor objetivo, mientras se observan los datos enviados y recibidos. Lo que hace este ataque es realizar pruebas con la compresión de los datos con varias peticiones e ir averiguando si las modificaciones que se introducen producen la salida adecuada, reduciendo o aumentando el tamaño de los datos cifrados para saber si el contenido introducido es el correcto.

La mayor parte de internet utiliza DEFLATE (RFC 1951) para realizar la compresión, este es una combinación del algoritmo LZ77 y la codificación Huffman:

- **LZ77** evalúa la entrada, busca por cadenas que se repiten y las reemplaza por índices hacia la última aparición, distancia y longitud.

- **Huffman** reemplaza bytes comunes por códigos más pequeños. Crea una tabla de símbolos en la cual guarda cada byte como un código único.
- Un tercer elemento de la compresión es el **tamaño de ventana**, por el coste computacional se define un tamaño de ventana en el cual cuando se comprime se buscará una ocurrencia dentro de ese espacio únicamente. Esta ejerce de límite de búsqueda.

En CRIME para cada búsqueda se envían dos peticiones, la primera con una prueba dentro de la ventana de la *cookie* y la segunda como permutación de la primera con una prueba fuera del límite. Si la prueba es incorrecta se avanza la ventana y si es correcta se ha encontrado el dato buscado dentro de esa ventana.

La mitigación para este ataque fue simple, los clientes, como por ejemplo los navegadores, deshabilitaron la compresión TLS y ya no la anuncian en sus *ClientHello*.

3.3.4.3. BREACH (Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext)

Es un ataque a la compresión HTTP parecido a CRIME surgido en agosto de 2013. Los autores se centraron en demostrar que CRIME trabaja igualmente bien en respuestas HTTP que utilizan compresión. BREACH es conceptualmente igual a CRIME, requiriendo acceso al tráfico de red de la víctima y necesitando la habilidad de que ejecute un código Javascript. La compresión HTTP solo se aplica a los cuerpos en las respuestas, lo cual significa que no se pueden extraer secretos de estas. Sin embargo, los cuerpos de las respuestas si contienen datos sensibles. Los autores se centraron en extraer *tokens* mediante CSRF (*Cross-site request forgery*), que les permitirían impersonar a la víctima para atacar a la aplicación web.

Para establecer un punto de partida, un atacante introduce un valor *canary=*[1] en la primera petición. Debido a la duplicación que existirá en la respuesta, el cuerpo de la misma será más pequeño y podrá ser detectado. Desde este punto el ataque continúa como en CRIME.

[1]Nombre típicamente utilizado en el ámbito de la seguridad para definir un elemento del que se desea conocer la presencia o valor antes y después de un ataque

3.3.4.4. *Heartbeat*

Heartbleed es el nombre de una de las peores vulnerabilidades que han afectado a OpenSSL descubierta en abril de 2014. El ataque explota un fallo de implementación de la extensión *Heartbeat* poco utilizada de la librería OpenSSL. Se considera el peor fallo de seguridad que ha afectado a TLS, aunque no es un fallo del protocolo ni criptográfico, sino de implementación de la librería, la cual era conocida por su poca calidad durante mucho tiempo. Este fallo produjo la publicación de un *roadmap* de revisión de OpenSSL para identificar y corregir los problemas que se pudiesen encontrar y también un *fork* por parte de OpenBSD llamado LibreSSL [2].

La vulnerabilidad se produce por una incorrecta validación de la longitud de entrada en el código. La explotación otorga la capacidad al atacante de recuperar hasta 64 KB de datos de la memoria principal del servidor por cada petición de *heartbeat*. El atacante puede enviar múltiples peticiones con las cuales puede recuperar un número ilimitado de información del servidor. Una de las consecuencias de esta vulnerabilidad y por la propia naturaleza de OpenSSL lo más probable es que el atacante desee y consiga la clave privada del servidor, además de *session tickets*, contraseñas y demás información sensible.

Heartbleed afecta a OpenSSL desde la versión 1.0.1 hasta la 1.0.1f. Las versiones anteriores no son vulnerables. El impacto fue muy grande puesto que se calculaba que el 17 % de los servidores utilizaban esta librería [2].

La mitigación para esta vulnerabilidad es parchear la librería con una versión posterior a 1.0.1g. Además se podría compilar la librería con el *flag OPENSSL_NO_HEARTBEATS* para deshabilitar la extensión. El mayor peligro de estas vulnerabilidades de implementación son los productos que tienen OpenSSL embebido y que son difíciles de actualizar o no tienen soporte del fabricante.

En su momento se recomendó crear nuevas claves por si en algún momento la vulnerabilidad hubiese permitido a algún atacante robar las claves privadas, así como avisar a los usuarios de las plataformas para modificar su contraseña por si en algún momento pudiese ser extraída de la memoria del servidor.

Aquellos que utilizaban *forward secrecy* son los que quedaron en la mejor situación, puesto que el robo de la clave privada no habilita al atacante a descifrar todas las comunicaciones pasadas.

3.3.4.5. FREAK (Factoring RSA Export Keys)

FREAK es el nombre de una vulnerabilidad publicada en el proyecto OpenSSL que fue descubierta en enero de 2015 (CVE-2015-0204). Este fallo tiene una peculiar historia detrás. Antes del despliegue del comercio electrónico y la concienciación de que debía existir seguridad en la web, Estados Unidos consideraba la tecnología de cifrado como restringida para su exportación a otros países por si estos algoritmos de cifrado pudiesen caer en manos "enemigas". A finales de los años 90 del siglo pasado las versiones internacionales del *software* creado en Estados Unidos utilizaban longitudes de clave menores que aquellos productos que se ponían en su mercado, limitando la longitud de los cifrados a 40 bits y el intercambio de claves a 512 bits. A esos conjuntos de cifrado se los marcó con la etiqueta EXPORT. Aunque esta situación absurda fue cambiando y los cifrados deliberadamente débiles fueron desapareciendo, estos *cipher suites* no terminaron por desaparecer del todo. A pesar de esto, definir *cipher suites* intencionadamente débiles no era suficiente puesto que para mejorar el rendimiento, los conjuntos con RSA combinaban autenticación e intercambio de claves. Aunque la autenticación fuerte era generalmente permitida, con los conjuntos RSA no se podía separar del intercambio de claves, por lo cual, la solución fue extender el protocolo de producir claves RSA débiles para utilizar con conjuntos de cifrado de EXPORT. Así, un servidor con una clave fuerte RSA puede seguir utilizándola para la autenticación pero, para el intercambio de claves se debería generar una clave RSA débil de 512 bits de máximo para la negociación con el conjunto de EXPORT. Estas *cipher suites* fueron declaradas obsoletas en el año 2000, pero el código desarrollado en las librerías SSL/TLS permaneció [2].

Este fallo permite a un atacante, en situación de interceptar la comunicación entre el cliente y el servidor, renegociar la conexión y tratar de que ambos utilicen un *cipher suite* de la categoría EXPORT, que al ser más débiles podrían ser descifrados sin la necesidad de una gran cantidad de *hardware*. A continuación,

se detalla más en profundidad el ataque.

En un intercambio normal con RSA, el cliente genera un secreto (*premaster secret*) y lo envía al servidor cifrado con la clave pública RSA de este último. Si la clave es fuerte, el intercambio también lo es. Sin embargo, cuando se negocia una *suite* EXPORT, el servidor genera una clave RSA débil, la firma con su clave fuerte y la envía al cliente en el mensaje *ServerKeyExchange*. El cliente utiliza la clave débil para cifrar el *premaster secret* para cumplir con las regulaciones de EXPORT. Aunque la clave sea débil, un atacante activo no podría descifrar en vivo la comunicación puesto que la firma continúa siendo fuerte (suponiendo una clave RSA original de un buen tamaño).

Los clientes modernos no soportan este tipo de conjuntos de cifrado pero FREAK no lo necesita. Para explotar FREAK, el atacante debe de alguna manera forzar el mensaje *ServerKeyExchange* hacia la víctima con lo que podría realizar un *downgrade* de la complejidad de la clave a 512 bits. Pero existen dos obstáculos: la firma inyectada debe ser firmada con una clave RSA fuerte utilizada por el servidor y habiendo interferido el *handshake* TLS, el atacante debe buscar una manera de generar un mensaje *Finished* legítimo con los cambios, cosa que solo debería poder realizar un servidor legítimo.

Debido al primer obstáculo, el ataque solo funcionará contra servidores que soporten *suites* EXPORT. El atacante debe conectarse directamente al servidor ofreciendo únicamente *suites* EXPORT. Esto genera una negociación de un conjunto EXPORT, durante el cual el servidor envía un mensaje *ServerKeyExchange*. El truco aquí es reutilizar este mensaje contra la víctima. Aunque TLS se defiende contra ataques de reenvío de paquetes firmados, la defensa se basa en los números aleatorios generados por el cliente y el servidor enviados en el *ClientHello* y *ServerHello* respectivamente. Pero un atacante puede esperar a que el cliente se conecte primero, copiar los números aleatorios originales en una conexión separada hacia el servidor objetivo. El resultado es que el mensaje *ServerKeyExchange* pasa la validación del cliente.

El mayor problema es reproducir un mensaje correcto *Finished*. Este mensaje

es cifrado y contiene el *hash* de todos los mensajes de *handshake* y ambos interlocutores validan el contenido de este mensaje de acuerdo a sus propios cálculos. Debido al cifrado y a la validación de integridad el atacante no podría modificar los contenidos de este mensaje. Sin embargo, en ese momento, el atacante ha reducido el tamaño de la clave a 512 bits y con suficiente potencia computacional puede mediante fuerza bruta descifrar el *premaster secret* enviado por el cliente, ganando así completo control sobre la conexión.

Se debe reconocer que aunque las claves de 512 bits son débiles, estos ataques de fuerza bruta llevan un tiempo y pocas organizaciones serán capaces de romperlas en tiempo real. El problema está en que, por razones de rendimiento, los servidores web no generan nuevas claves débiles por sesión, sino que las reutilizan durante un período de tiempo a veces demasiado amplio. Esto ha llevado a investigadores a romper estas claves en 7 horas con 100$ en máquinas de AWS modestas, con lo que podrían efectuar el ataque si la clave siguiese siendo la misma por parte del servidor [2].

Esta vulnerabilidad afectó a OpenSSL, Secure Transport (libería de Apple) y Schannel (librería de Microsoft). Además, fue un ejemplo de que las funcionalidades obsoletas deben ser eliminadas del *software* y dejar de ser soportadas por los servidores. Aquellos servidores dejarán de dar soporte a *suites* EXPORT no quedaron expuestos a este ataque.

3.3.4.6. POODLE (Padding Oracle On Downgraded Legacy Encryption)

Esta vulnerabilidad fue descubierta en octubre de 2014 por el equipo de seguridad de Google, afecta a SSLv3 y permite a un atacante recuperar pequeñas porciones de datos cifrados.

El principal problema de esta vulnerabilidad es el diseño de los bloques CBC, que autentica el texto plano pero deja el *padding* desprotegido. Esto habilita al atacante a poder realizar cambios en el relleno de los bloques y transmitir y explotar los *padding oracles* para revelar los contenidos cifrados. Lo que hace posible este ataque es la construcción y reglas de comprobación que existe en

SSLv3 y que fueron corregidas en TLSv1.0 y posteriores. En la Figura 3.10 se pueden apreciar las diferentes formas de construcción del relleno entre SSLv3 y TLSv1.0.

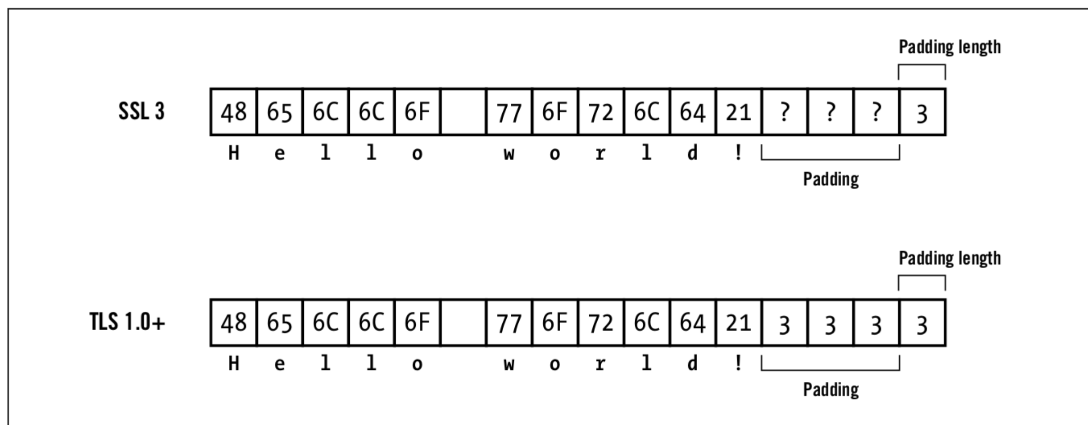


Figura 3.10: Relleno de bloques en SSLv3 vs TLSv1.0 [2]

Como se puede observar en la imagen, SSLv3 reserva el último byte en el bloque cifrado para guardar el tamaño del relleno, pero no tiene ninguna regla para la creación de los bytes de relleno. Esto produce que no exista ninguna comprobación para asegurarse de que el relleno no ha sido modificado. En TLSv1.0 y posteriores esto ha sido corregido y los bytes de relleno tienen el mismo valor que el último byte de relleno, el contador. En estas versiones más modernas el receptor comprueba los valores del relleno justo después del descifrado, si el tamaño del relleno y todos los bytes no son iguales, todo el bloque es rechazado al ser inválido.

Supongamos que el atacante realiza un pequeño cambio. Como esto va a producir multitud de cambios después del descifrado, el contenido que producirá será aleatorio. Pero, gracias al fallo de SSLv3 los cambios en los bytes de relleno no serán detectados, el único byte que se comprobará, y es necesario que sea correcto, es el último que transmite la longitud de relleno. En la práctica esto significa que una vez de cada 256 cambios el bloque será aceptado sin detección. Aunque el atacante no puede obtener el texto plano directamente, sabe que cuando su modificación es aceptada, el último byte después del descifrado tendrá el valor correcto de longitud de relleno. Así, obtiene un “oráculo”.

El ataque a nivel práctico es complicado. El atacante debe tener el control del navegador de la víctima para enviar peticiones arbitrarias al servidor objetivo y además debe ser capaz de ver el tráfico de red para coordinar su actividad con el propio navegador. Este es el mismo caso que BEAST y otros ataques ya citados. Cada intento consiste en que el navegador inicia una petición y el atacante reemplaza el último bloque cifrado con el que desea probar. El proceso se repite hasta que una prueba es correcta y un byte de texto plano es recuperado. El atacante entonces se mueve al siguiente byte. Con cada petición, el atacante debe influenciar la posición de los bytes del secreto que desea averiguar. Esto significa ser capaz de inyectar algo antes y después del secreto.

El impacto de este fallo de SSLv3 debería ser mínimo, puesto que todos los servidores a día de hoy (y también cuando POODLE fue descubierto) soportan TLSv1.0 y posteriores. El problema es que existen multitud de servidores que todavía soportan esta versión y antes de realizar este ataque se realiza un *downgrade* de versión. Desafortunadamente la mayoría de navegadores fueron diseñados para realizar *downgrade* cuando se producen fallos de *handshake* en TLS. Un ataque relativamente fácil de realizar y que tuvo como resultado la posibilidad de realizar el ataque POODLE incluso con versiones modernas de navegadores.

La mitigación por parte de los vendedores de navegadores web fue fácil, acelerar la muerte de SSLv3, primero deshabilitando el retroceso a SSLv3 o deshabilitando CBC (como fue el caso de Safari). Posteriormente se deshabilitó SSLv3 por defecto. Además, para protegerse de los ataques de *downgrade* se creó un nuevo *cipher suite*, llamado TLS_FALLBACK_SCSV, que el navegador envía al servidor para señalizar que se está realizando el *downgrade*. Con lo que el servidor puede decidir rechazar la conexión para prevenir el ataque.

3.4. TLSv1.3

Esta versión fue publicada oficialmente en agosto de 2018, a través del RFC 8446 [9]. Esta versión se publica después de 10 años con TLSv1.2 vigente, llega para actualizar múltiples parámetros de seguridad pero también para mejorar el rendimiento de las comunicaciones cifradas que llevaban muchos años sin ninguna mejora y unos tiempos de establecimiento de sesión demasiado costosos debido a los mensajes necesarios para completar cada *handshake*.

3.4.1. Características de TLSv1.3 y diferencias con TLSv1.2

Las principales novedades de TLSv1.3 son las siguientes:

- Se han eliminado de la lista de algoritmos de cifrado simétricos aquellos algoritmos obsoletos. Los únicos algoritmos simétricos que se utilizarán en esta nueva versión serían del tipo AEAD (*Authenticated Encryption with Associated Data*).
- Existe un nuevo modo llamado zero-RTT (0-RTT) que ha sido añadido para mejorar los tiempos de conexión con el coste de algunas capacidades de seguridad.
- Los *cipher suites* estáticos de RSA y Diffie-Hellman han sido eliminados, toda la criptografía basada en claves públicas debe proveer *forward secrecy.*
- Todos los mensajes del *handshake* tras el *ServerHello* viajan ahora cifrados.
- Las funciones de derivación de claves han sido rediseñadas, basadas en HMAC utilizando *Extract-and-Expand Key Derivation Function (HKDF).*
- La máquina de estados del *handshake* ha sido reestructurada par ser más consistente y eliminar mensajes y estados superfluos.
- ECC es ahora la especificación básica e incluye nuevos algoritmos de firma.
- La compresión, los grupos personalizados DHE y DSA han sido eliminados.

- El resumen de sesión en la parte del servidor o del cliente y los *cipher suites* de anteriores versiones de TLS han sido reemplazadas por un único intercambio PSK.

A continuación se detallarán estas mejoras y características de TLSv1.3

3.4.1.1. *Handshake* en TLSv1.3

El *handshake* en esta nueva versión ha sufrido cambios para mejorar el rendimiento del establecimiento de sesión, que llevaba años sin actualizarse con demasiados pasos que aumentaban el tiempo de establecimiento. En TLSv1.2 el establecimiento de sesión consume en total 3 RTT (*Round Trip Time*), definido como el tiempo de ida y vuelta entre el cliente y el servidor. En la Figura 3.11 se pueden apreciar los pasos del *handshake* en TLSv1.2 en comparación con TLSv1.3.

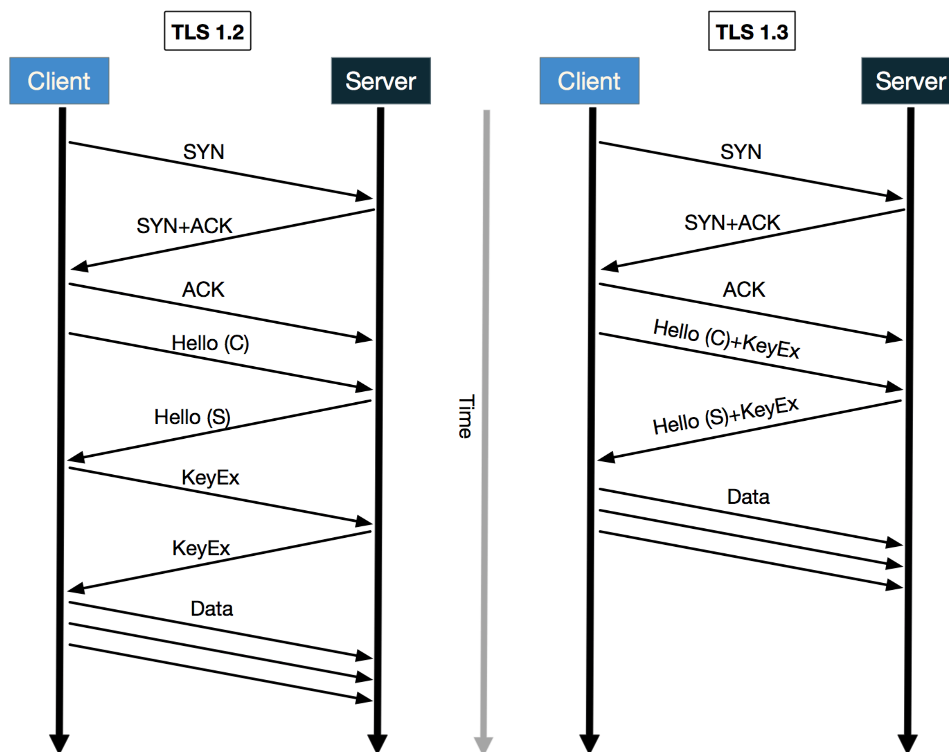


Figura 3.11: Handshake en TLSv1.2 y 1.3 [4]

Como se puede apreciar en la Figura anterior, el *handshake* se ha acortado gracias a unir las fases de *Hello* y el intercambio de claves, es un cambio simple pero efectivo. A continuación se define el proceso [9]:

- En TLSv1.3 el cliente empieza el establecimiento de la sesión enviando un ***ClientHello*** con la siguiente información, que es aplicable al *ServerHello* con la misma información.

    1. ***Client Version***: por interoperabilidad, la versión de los mensajes de *Hello*, anuncia TLSv1.2 con el valor 0x0303. Este valor ya no es utilizado en la negociación, se codifica de manera fija para que la mayoría de software de nodos intermedios no rompa. Para la negociación real, se introduce la extensión *Supported Versions*.
    2. ***Client/Server Random***: datos generados aleatoriamente para ser utilizados a lo largo del *handshake* (32 bytes).
    3. ***Session ID***: este campo ya no es necesario en TLSv1.3, puesto que el resumen de sesión no se realiza con un ID de sesión. En TLSv1.3 el resumen de sesión se realiza mediante un mecanismo de claves precompartidas. Por compatibilidad y para no romper *software* antiguo se introduce un valor aleatorio. El servidor simplemente responderá con el mismo valor.
    4. ***Cipher Suites***: lista ordenada de los conjuntos de cifrado compatibles.
    5. ***Copmression Methods***: TLSv1.3 no permite compresión, que es vulnerable a ataques como CRIME (Sección 3.3.4.2) y este campo llevará siempre un valor nulo (0x00) indicando su no uso.
    6. ***Extensions Length***: el cliente provee una lista opcional de extensiones que el servidor puede utilizar para alcanzar una determinada funcionalidad. Este campo indica el tamaño en bytes del conjunto de extensiones utilizados.
    7. ***Extension - Supported Groups***: indica los tipos de curva elíptica que el cliente soporta para el intercambio de claves. Anteriormente era conocido como *Elliptic Curves* [9].

8. ***Extension - Signature Algorithms***: indica los algoritmos de firma que el cliente soporta. La lista está ordenada por orden de preferencia.
9. ***Extension - Key Share***: el cliente envía una o varias claves públicas utilizando un algoritmo que cree que el servidor va a soportar. Esto permite que el resto de mensajes del *handshake* en TLSv1.3 vayan cifrados, a diferencia de los demás.
10. ***Extension - PSK Key Exchange Modes***: el cliente indica los modos disponibles para establecer claves precompartidas (PSKs).
11. ***Extension - PSK***: el cliente incluye el *ticket* con esta extensión. Explicado con mayor detalle en la Sección 3.4.1.2.
12. ***Extension - Early Data***: en el modo 0-RTT el cliente indica con esta extensión que enviará datos junto al primer mensaje.
13. ***Extension - Supported Versions***: el cliente indica soporte para TLSv1.3.

- Por compatibilidad con el *software* antiguo, el resto del *handshake* en esta nueva versión será disfrazado de un resumen de sesión TLSv1.2 que concluye exitosamente.

- A partir del mensaje de *ClientHello*, cliente y servidor comienzan los cálculos criptográficos para alcanzar, con el intercambio de claves y los algoritmos de curva elíptica, la clave simétrica que utilizarán durante el resto del *handshake*. De hecho, como el servidor ya tiene la clave pública del cliente y la información criptográfica, puede enviar su certificado, extensiones y el mensaje de que ha finalizado ya cifrados.

- En el mensaje del servidor ***Change Cipher Spec*** que viaja al mismo tiempo que el *ServerHello* (ahorrando un RTT) el servidor inserta la siguiente información cifrada.

1. ***Server Encrypted Extensions***: cualquier extensión que el servidor desee enviar al cliente que no sea necesaria para la negociación de las claves viajará cifrada como parte de este mensaje.
2. ***Server Certificate***: el servidor envía uno o más certificados con la información del host.

3. ***Server Certificate Verify***: el servidor provee la información para confirmar que está en posesión de la clave privada del certificado enviado. Puesto que la generación de claves en TLSv1.3 es siempre efímera, el servidor prueba estar en posesión de la clave privada firmando el *hash* de uno de los mensajes del *handshake*.
4. ***Server Handshake Finished***: para verificar que el *handshake* se completó correctamente, el servidor calcula un hash con todos los mensajes del *handshake* y los envía al cliente para demostrar que el establecimiento fue correcto.

- Para finalizar, el cliente envía un mensaje ***Client Handshake Finished*** para que el servidor verifique que este tiene los datos de toda la negociación igual que con el mensaje *Server Handshake Finished*.

Este mensaje puede viajar en el mismo segmento de una petición de capa 7, con lo cual, contando con el establecimiento de la sesión TCP, el *handshake* consumirá 2 RTT antes de poder enviar datos de capa de aplicación.

3.4.1.2. Resumen de sesión

Las versiones anteriores a TLSv1.3, como se comentó en las secciones previas, tenían dos métodos de resumir una sesión preexistente. Se podía reutilizar el *Session ID*, que utilizaba la caché de sesiones del servidor para volver a establecer los parámetros de la sesión; o utilizar un *ticket* que enviaba el servidor al cliente, transfiriéndole así la capacidad y responsabilidad del restablecimiento de sesión (y liberando recursos del servidor). La reutilización continua de un ID o *ticket* de sesión tiene un coste de seguridad, puesto que el *master secret* es reutilizado y si es comprometido se podrán descifrar las sesiones resumidas.

En TLSv1.3 el *Session ID* y *tickets* de sesión desaparecen, pero se mantiene la capacidad de que el cliente pueda resumir una sesión anterior mediante claves precompartidas. Una vez se ha completado el *handshake*, el servidor puede enviar al cliente una clave precompartida o PSK (*Pre Shared Key*) que es una clave única derivada del *handshake* inicial. El cliente puede utilizar esa PSK en futuros *handshakes* para negociar el resumen de sesión. Si el servidor acepta esta PSK,

el contexto de seguridad de la nueva conexión está criptográficamente atado a la conexión original, y la clave derivada del *handshake* original es utilizada para desencadenar un nuevo estado criptográfico en vez de una nueva negociación completa. Si el servidor no acepta la clave precompartida, se iniciaría un nuevo *handshake* completo con el cliente [9].

Las claves PSK pueden ser utilizadas con intercambio de claves ECDHE para proveer *forward secrecy* en combinación con las claves precompartidas, o pueden ser utilizadas por si mismas perdiendo esta capacidad. Puesto que el resumen de la sesión con la clave precompartida crea otra negociación con Diffie-Hellman para el *forward secrecy* y la creación de una clave derivada, el resumen es más lento que en versiones anteriores de TLS, con el uso de curvas elípticas que son más seguras y computacionalmente eficientes se reduce esta diferencia, aunque continúe existiendo. No cabe duda, que con ECDH se obtiene la mejor relación entre rendimiento y seguridad para este caso.

3.4.1.3. Modos 1-RTT y Zero-RTT

Relacionado con el resumen de sesión está la forma en la cual ésta se lleva a cabo. Se puede decir que TLSv1.2 y anteriores realizan el resumen de sesión utilizando 1-RTT.

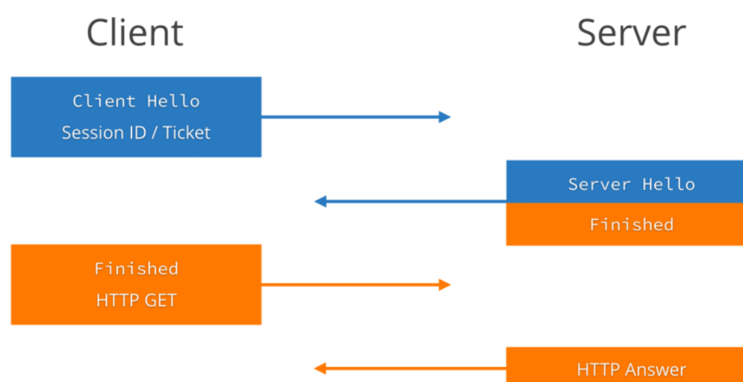


Figura 3.12: 1-RTT en TLSv1.2 y 1.3 [5]

Esto es así porque el cliente en el mensaje *ClientHello* introduce la información de sesión (ID o *ticket*) y salta directamente al estado finalizado cuando el servidor

contesta afirmativamente, pudiendo así enviar datos de aplicación. En TLSv1.3 este mecanismo puede ser utilizado con 1-RTT de una manera equivalente con las claves precompartidas, pero se introduce un nuevo modo llamado 0-RTT. En la Figura 3.12 se puede apreciar un resumen de sesión con 1-RTT de latencia.

El modo 0-RTT trata de resumir una sesión con el mínimo impacto de latencia posible permitiendo enviar la información de sesión al mismo tiempo que se envían los datos de aplicación. El servidor entonces comprobaría que la información de la clave precompartida es válida y en un mismo mensaje introduciría la finalización del resumen de sesión junto con los datos de repuesta de la capa de aplicación. En la Figura 3.13 se puede apreciar un resumen de sesión con 0-RTT de latencia.

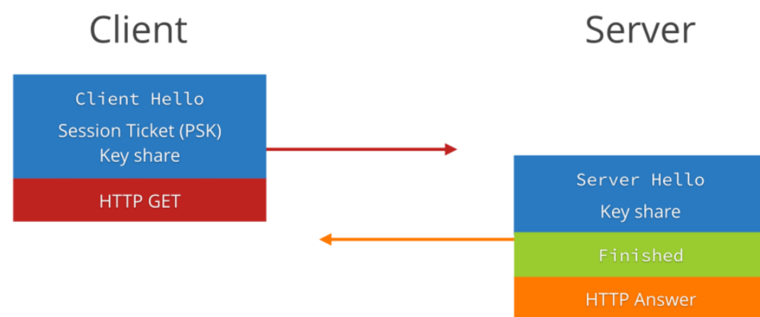


Figura 3.13: 0-RTT en TLSv1.3 [5]

Este nuevo modo mejora la latencia de la comunicación a costa de reducir la seguridad, a continuación algún ejemplo [5]:

- Puesto que la PSK no se negocia de nuevo, no provee *forward secrecy* ante un *ticket* de sesión comprometido. Si un atacante en un futuro descifra el *ticket* de sesión con un envío con 0-RTT, esos datos estarían comprometidos (pero no el resto de la comunicación). Esto muestra la importancia de rotar los *tickets* de sesión entre cliente y servidor. TLSv1.2 nunca ha otorgado *forward secrecy* contra un *ticket* de sesión comprometido, con lo cual incluso existiendo este problema, 0-RTT en TLSv1.3 es una gran mejora.
- Existe un gran problema con ataques de *replay*, puesto que los *tickets* de sesión por su propia naturaleza no mantienen control de estado, no hay

manera de saber si un paquete con 0-RTT se ha enviado anteriormente. Esto es un problema con operaciones que producen cambios de estado en los servidores, por lo cual, la solución es que los servidores no deben ejecutar operaciones no idempotentes con reestablecimientos 0-RTT. En estos casos, debe forzar al cliente a una negociación 1-RTT que protege de la replicación, puesto que contienen datos aleatorios y números de secuencia que no es posible replicar. El caso típico, es que un reestablecimiento con 0-RTT, por ejemplo, en HTTP sea una solicitud GET de un recurso que no provoca cambio de estado alguno.

3.4.1.4. Protección *Anti-downgrade*

Puesto que una nueva versión del protocolo no puede modificar como funcionan las versiones antiguas, TLSv1.3 no podrá mejorar la seguridad de las versiones anteriores. Con el deseo de asegurar que si un día TLSv1.2 es comprometido, cliente y servidor que soporten ambas versiones no serán engañados para realizar la vuelta atrás hacia esta versión.

TLSv1.3 tiene una solución ingeniosa para este caso, si un servidor tiene que utilizar TLSv1.2 porque parece que el cliente no soporta la nueva versión, ocultará un mensaje en el valor *Server Random* que es firmado con certificado en 1.2 e imposible de falsificar *a priori*. Un cliente real que utilice la versión 1.2 ignorará este mensaje oculto, pero un cliente soportando 1.3 sabrá al momento que alguien está tratando de realizar un *downgrade* de la conexión. El mensaje que se introducirá serán los bytes: 44 4F 57 4E 47 52 44 01 (D O W N G R D 01) cuando se trata de negociar TLSv1.2 y 44 4F 57 4E 47 52 44 00 cuando se trata de negociar TLSv1.1 o menor [9].

3.4.1.5. *Cipher suites* soportadas por TLSv1.3

En esta nueva versión se ha realizado una revisión en profundidad de todos aquellos elementos criptográficos de los cuales existe duda por su seguridad y se han eliminado. Este escrutinio ha llevado a que esta versión sea una de las que menos conjuntos de cifrado soporta, una ventaja de implementación (ya que no existen cientos de posibilidades en la negociación) y de seguridad, puesto que solo se ha aprobado los algoritmos que no tienen ninguna duda. Han sido eliminados:

- Algoritmos de intercambio de clave que no ofrecen *forward secrecy* como RSA o DH estático.
- Algoritmos de cifrado simétrico que utilizan modos de bloque CBC, responsables de múltiples fallos de seguridad y reemplazados todos ellos por AEAD.
- Algoritmos de *hash* antiguos, como SHA1 y MD5.
- RSA PKCS#v1.5
- Grupos Diffie-Hellman propios.

A continuación, en la Tabla 3.1 se puede apreciar la corta lista de conjuntos de cifrado citados en el RFC 8446 [9].

***cipher suites* en TLSv1.3**	
Valor	***Cipher suite***
0x01	TLS_AES_128_GCM_SHA256
0x02	TLS_AES_256_GCM_SHA384
0x03	TLS_CHACHA20_POLY1305_SHA256
0x04	TLS_AES_128_CCM_SHA256
0x05	TLS_AES_128_CCM_8_SHA256

Cuadro 3.1: *Cipher suites* en TLSv1.3

Capítulo 4

Soporte y uso de TLSv1.3

Índice general

4.1. Soporte *software* para TLSv1.3 . . . . **63**

4.2. Grado de adopción de TLSv1.3 . . . . **64**

4.3. Mejoras de rendimiento de TLSv1.3 . . . . **68**

En este capítulo se va a analizar el grado de adopción de esta nueva versión, tanto en el *software* que tiene que darle soporte, como su propio uso y adopción por parte de las entidades y usuarios que lo utilizan.

4.1. Soporte *software* para TLSv1.3

En esta sección se analiza el soporte *software* que existe para esta nueva versión y desde qué versión y momento se ha ofrecido este soporte. Algunas versiones *software* ofrecen soporte desde antes de la publicación del RFC 8446, porque han participado en los distintos *drafts* o porque llevaban tiempo desarrollando el producto basándose en los mismos. A continuación, se muestra el listado de *software* más utilizado en la industria y es una colección que puede quedarse corta dado el uso a día de hoy que se hace de TLS.:

- Nginx: este servidor web ofrece soporte desde la versión 1.13 (en fase de pruebas en un primer momento), publicada en abril de 2017; y la versión 1.17 de manera estable publicada en julio de 2019.

- Apache: otro servidor web que incluye soporte a partir de la versión 2.4.38, publicada en febrero de 2019.
- F5 BIG-IP: este balanceador, waf, firewall y demás, integró en su versión 14.X (publicada en enero de 2019) soporte para este nuevo estándar, pero algunas funcionalidades como el modo 0-RTT no han sido integradas hasta la versión 15.X.
- GnuTLS: esta librería ofrece soporte desde la versión 3.6.4, publicada en septiembre de 2018.
- OpenSSL: esta utilidad ofrece soporte desde la versión 1.1.1, publicada en septiembre de 2018.
- Google Chrome: este navegador ofrece soporte desde la versión 63 (versión *draft*), publicada en diciembre de 2017; y la versión 70 (estable) publicada en octubre de 2018.
- Mozilla Firefox: otro navegador que ofrece soporte desde la versión 61, publicada en junio de 2018.
- Microsoft Windows: el sistema operativo ha integrado de manera oficial esta versión en su build 20170 en agosto de 2020, que habilita TLSv1.3 en IIS (su servidor web) y que permite habilitar esta versión en los navegadores antiguos.
- Apple iOS/macos: iOS desde su versión 12.3.1 y macos desde la 10.14.4 (safari 12) integran soporte para el nuevo estándar, publicadas a principios de 2019.

4.2. Grado de adopción de TLSv1.3

En esta sección se analizará la adopción de TLSv1.3 con algunos estudios que han sido publicados, siendo la mayor fuente de información el estudio del SANS Institute[1] (organización ampliamente respetada que aporta información y estudios de ciberseguridad). El estudio del SANS es el más reciente que se puede

[1]SANS Institute: `https://www.sans.org/`

encontrar, publicado en julio de 2020.

A continuación, se revisan los datos que aporta la plataforma Insights[2] de la Internet Society[3] (organización sin ánimo de lucro dedicada exclusivamente al desarrollo mundial de Internet). Esta organización publica información mes a mes de la adopción de ciertas tecnologías como: HTTPS, TLSv1.3, HTTP/3, DNSSEC, IPv6, etc. Los datos de adopción de TLSv1.3 son de la telemetría de Mozilla Firefox y muestran que ya se utiliza (según sus datos) en el 56.9 % de las plataformas web, en donde diciembre de 2020 muestra una gran subida de soporte a esta nueva versión de TLS. En la Figura 4.1 se puede observar un gráfico que muestra los datos de soporte durante los últimos dos años.

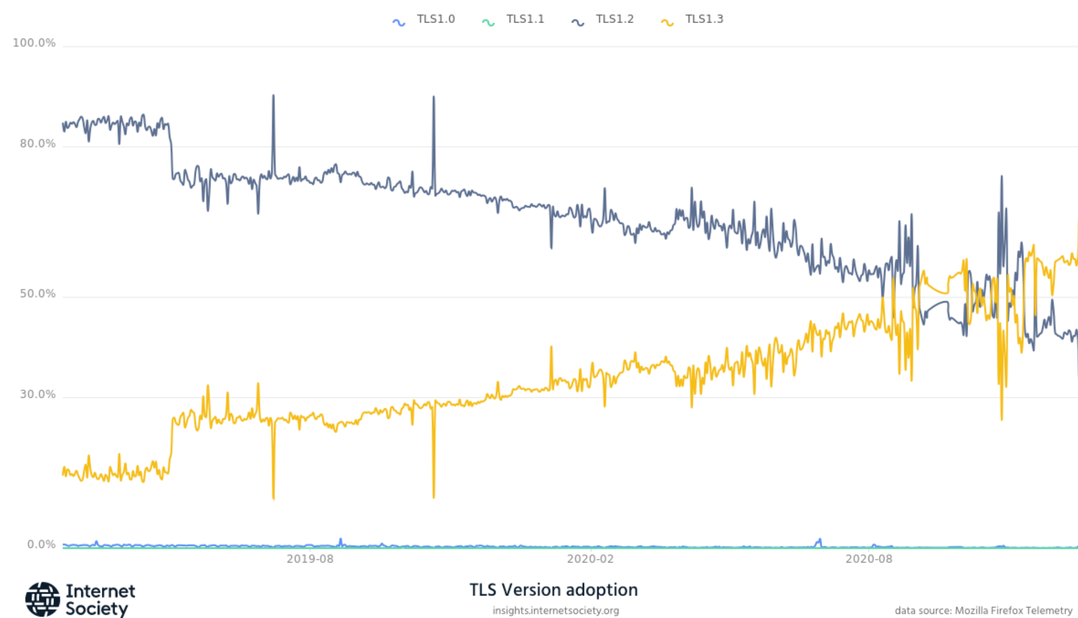


Figura 4.1: Adopción de TLSv1.3 según datos de Mozilla

De este gráfico se pueden extraer dos conclusiones, TLSv1.0 y TLSv1.1 son versiones residuales sin apenas uso, lo cual es una buena noticia. Además, se aprecia que TLSv1.3 va ganando soporte al mismo tiempo que TLSv1.2 lo va perdiendo en favor del primero.

[2]Insights: `https://insights.internetsociety.org/technologies`
[3]Internet Society `https://www.internetsociety.org/`

Cada vez más *software* está utilizando TLSv1.3 por defecto, y es de esperar que el soporte crezca de manera rápida. Además, la mayoría de plataformas web de cara al cliente utiliza una red de distribución de contenidos (CDN) como Akamai o Cloudflare y estas llevan dando soporte para esta versión desde hace bastante tiempo.

Este estudio ofrece información de adopción de las 500 webs más populares por categoría en julio de 2020, mostrando la versión más alta que soportan los servidores web. Comenzando con la categoría Ciencia, esta tiene una adopción de 28.4 % como se puede ver en la Figura 4.2.

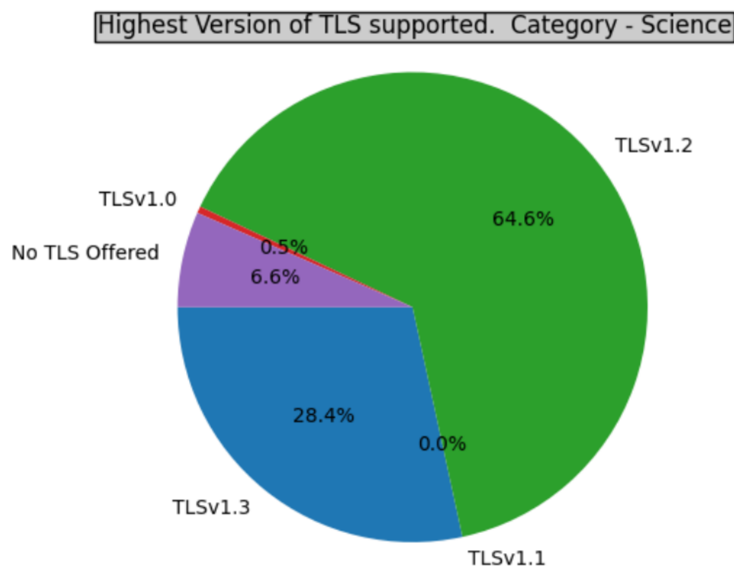


Figura 4.2: Adopción en la categoría Ciencia según SANS [6]

En la categoría Compras, casi un 29 % de las webs ofrecía soporte para TLSv1.3 como se observa en la Figura 4.3.

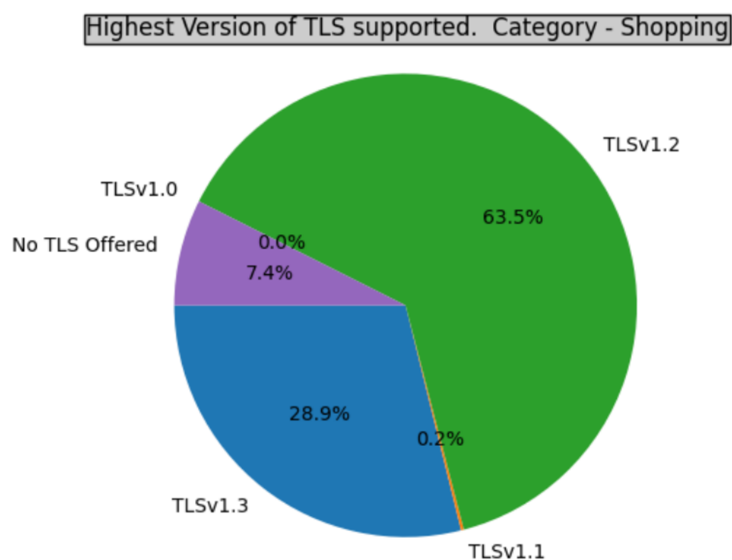


Figura 4.3: Adopción en la categoría Compras según SANS [6]

En la categoría Negocios, este porcentaje de soporte se reduce a un 15.5 % como se aprecia en la Figura 4.4. Esto muestra qué este tipo de webs tienen un despliegue más lento de TLSv1.3.

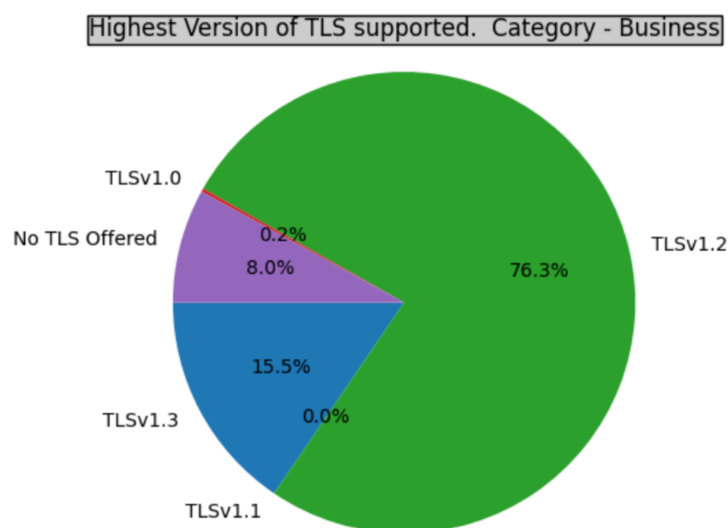


Figura 4.4: Adopción en la categoría Negocios según SANS [6]

El resultado de estos estudios muestra que la adopción de TLSv1.3 está siendo más lenta de lo que debería, pero día a día está mejorando. El problema es que la mayoría de webs soportan versiones de TLS antiguas por compatibilidad, aunque, como se ha visto, TLSv1.3 tiene mecanismos de defensa *anti-downgrade*. La utilización de TLSv1.3 mejorará cuando todo el *software* moderno lo utilice por defecto, cada vez más lo hace, y el *software* antiguo vaya siendo reemplazado.

4.3. Mejoras de rendimiento de TLSv1.3

Examinando el estudio de adopción del SANS [6] se demuestra como TLSv1.3 es entre un 5 %-10 % más rápido que TLSv1.2 y tiene entre 10 %-35 % más de estabilidad en las conexiones por segundo, cuando estas crecen dependiendo de la librería y herramienta que se utilice.

La mejora de rendimiento de TLSv1.3 con sus versiones anteriores en el establecimiento del *handshake* debería ahorrar 1-RTT al reducir los pasos del mismo. Además, los cálculos del intercambio de claves con curva elíptica puede ahorrar tiempo frente a cálculos con claves RSA muy grandes, aunque esto comparado con la latencia en la red es completamente despreciable.

Unas pruebas realmente interesantes son, por ejemplo, las realizadas en el blog `nooshu.github.io`[4]. En este, se realizan una serie de peticiones con *curl*, utilizando una de las capacidades de este programa que, con una plantilla, mide los tiempos del programa internamente diferenciando los tiempos de conexión, transferencia, total, etc. Realizando pruebas contra una web pública del gobierno de UK alojada en la CDN Fastly (el punto de presencia está muy cerca del cliente), demuestra que hay un incremento del rendimiento de entre un 20 %-50 % utilizando TLSv1.3. La reducción del tiempo de respuesta, así como cuanto tiempo se dedica a cada paso de la conexión como se puede observar en la Figura 4.5.

Además, en los casos que el modo 0-RTT de TLSv1.3 se pueda utilizar, la reducción de latencia deja la comunicación en 1-RTT, para esperar la respuesta de capa de aplicación.

[4]`https://nooshu.github.io/blog/2020/07/30/measuring-tls-13-ipv4-ipv6-performance/`

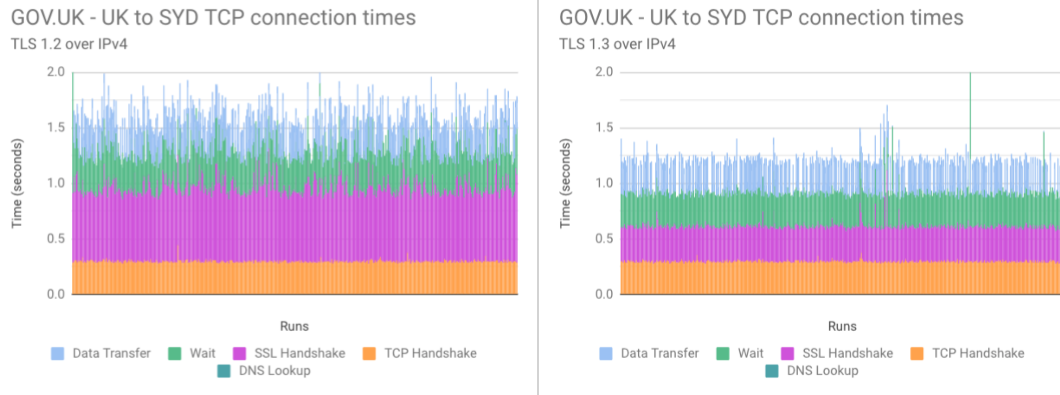


Figura 4.5: *Benchmark* TLSv1.2 vs TLSv1.3

Capítulo 5

Pruebas en laboratorio

Índice general

5.1. Topología y *software* utilizado **71**

5.2. Instalación y montaje **73**

5.2.1. Configuración de F5 BIG-IP 73

5.2.2. Configuración de los servidores web 79

5.3. Pruebas realizadas **82**

5.3.1. Diferencias de *Handshake* entre TLSv1.2 y 1.3 82

5.3.2. Resumen de sesión en TLSv1.2 vs TLSv1.3 89

5.3.3. Resumen de sesión 0-RTT con TLSv1.3 92

5.3.4. *Downgrade* en TLSv1.3 94

En este capítulo se realizan una serie de pruebas para demostrar de manera empírica el funcionamiento de TLSv1.3 y sus diferencias con sus antecesores. Se crea un laboratorio virtualizado con *software* ampliamente utilizado en la industria.

5.1. Topología y *software* utilizado

La topología será simple, se supone un entorno en el cual existen varios servidores web que soportan versiones antiguas de TLS y un balanceador que sirve a los clientes que soporta TLSv1.3. De esta manera, se podrán comparar las dos

versiones capturando el tráfico en el propio balanceador y experimentar con las distintas opciones de TLS de una manera sencilla. En la Figura 5.1 se puede apreciar la topología del laboratorio.

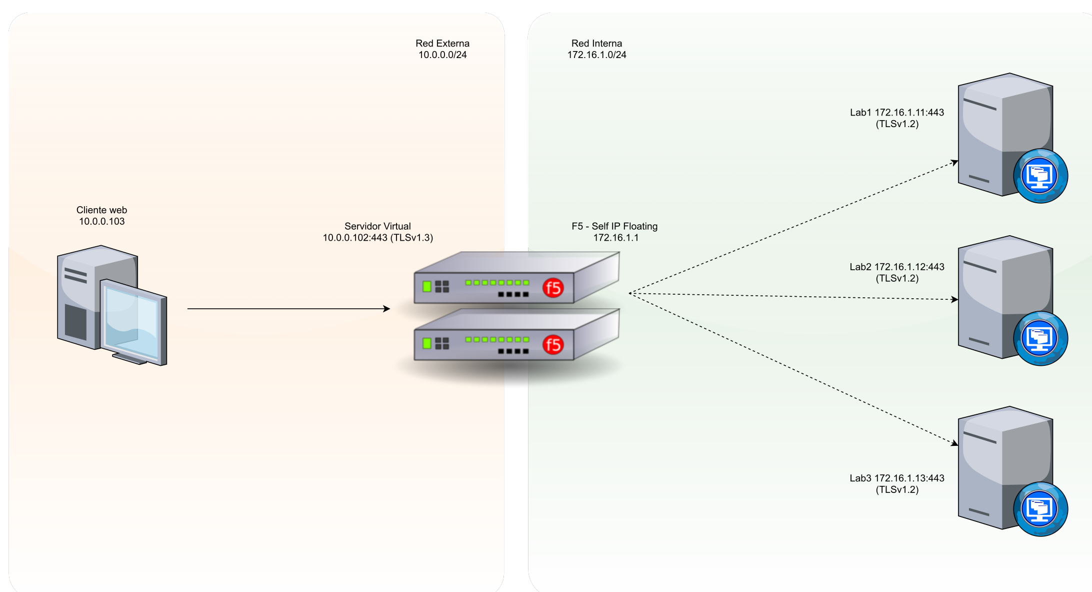


Figura 5.1: Topología del laboratorio.

Para montar este laboratorio se han levantado varias máquinas virtuales en un servidor físico con VMware ESXi como hipervisor. Los componentes son los siguientes:

- Un cliente web, que es un Fedora 32 de escritorio. Este simplemente utilizará los navegadores web para acceder al balanceo que presenta el balanceador. Tiene instalado Google Chrome 87 y OpenSSL 1.1.1i.
- Un balanceador, que es un BIG-IP de F5. Únicamente se utilizará el módulo de LTM[1] (*Local Traffic Manager*), para exponer el servidor virtual que servirá la web. El F5 está en una versión 15.1.2 con soporte para TLSv1.3.
- Tres servidores web, que son Fedora 32 *Server*. Estos linux tienen alojada una web simple, que indica en qué servidor ha caído la petición y los datos del cliente que se ha conectado. Como servidor web utilizan Nginx 1.18.0[2],

[1] F5 Local Traffic Manager: `https://www.f5.com/es_es/products/big-ip-services/local-traffic-manager`

[2] Nginx: `https://www.nginx.com/`

y además utilizarán PHP 7.4.13[3] para recuperar y mostrar los datos del cliente. Nginx está configurado para utilizar TLSv1.2.

5.2. Instalación y montaje

Sin entrar en profundidad, se explica la instalación y montaje de este laboratorio. Se obvia en esta explicación la parte del cliente, puesto que cualquier *software* que soporte un navegador web moderno puede ser utilizado para estas pruebas.

5.2.1. Configuración de F5 BIG-IP

Para la instalación del F5 se ha utilizado una única instancia, pero en un entorno real donde sea necesaria la alta disponibilidad, se desplegarán un mínimo de dos máquinas y se crearía un cluster entre las mismas. En este caso, con la única instancia que se ha desplegado, se ha configurado la red de gestión y los parámetros de instalación básicos y después se ha comenzado con la configuración específica que sería la siguiente:

- Creación de los **nodos**: en el balanceador hay que crear los nodos a los cuales se va a balancear. En este laboratorio se utilizarán tres servidores web, de los cuales se indica la IP y que serán monitorizados a nivel general mediante un *ping* desde el F5. Un ejemplo de configuración de los mismos es la Figura 5.2.

[3]PHP: `https://www.php.net/`

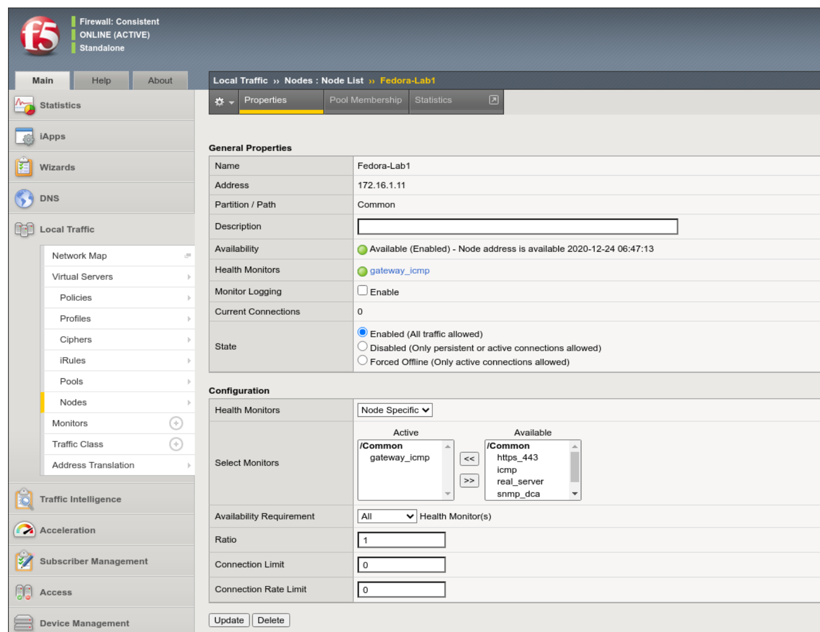


Figura 5.2: Definición de un nodo en F5.

Con la definición de los tres y su respuesta a los paquetes icmp, el F5 marcará los tres como levantados. Si no responden a icmp por cualquier razón se quitarán del balanceo y no se les enviarán nuevas conexiones. En la siguiente imagen se pueden ver los tres marcados como *OK*.

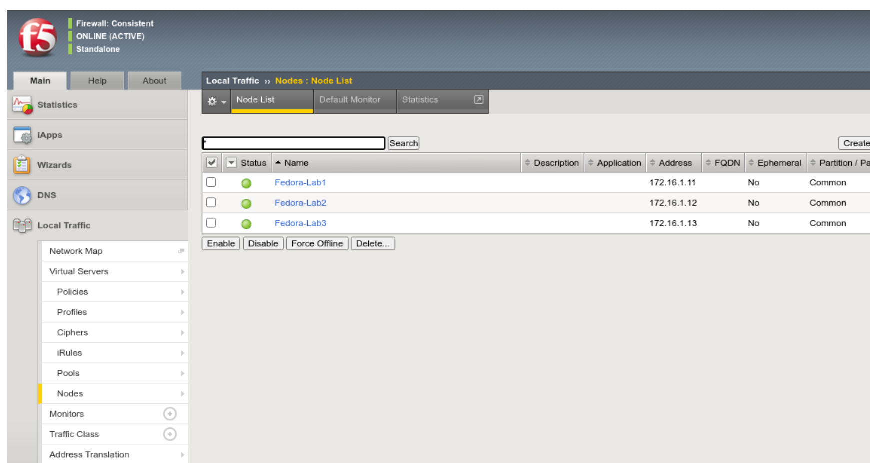


Figura 5.3: Nodos levantados en F5.

- Creación de los ***pools***: una vez están los nodos creados, el siguiente paso es crear la agrupación de los mismos, que además indica el servicio al cual

se está balanceando. La definición de un *pool* contiene la información de que nodos lo componen, el puerto (servicio) al cual se está balanceando y como se monitoriza este. En este laboratorio, aunque existen más *pools* de prueba, el más importante es el que balancea contra el puerto 443/tcp de los servidores web y que tiene un monitor que realiza un GET a la raíz mediante HTTP/1.1 y lo marca como vivo si contesta con un 200 OK. Esto obviamente, requiere un establecimiento de conexión TLS y una respuesta HTTP correcta. En la Figura 5.4 se puede apreciar la definición del *pool* de balanceo de HTTPS.

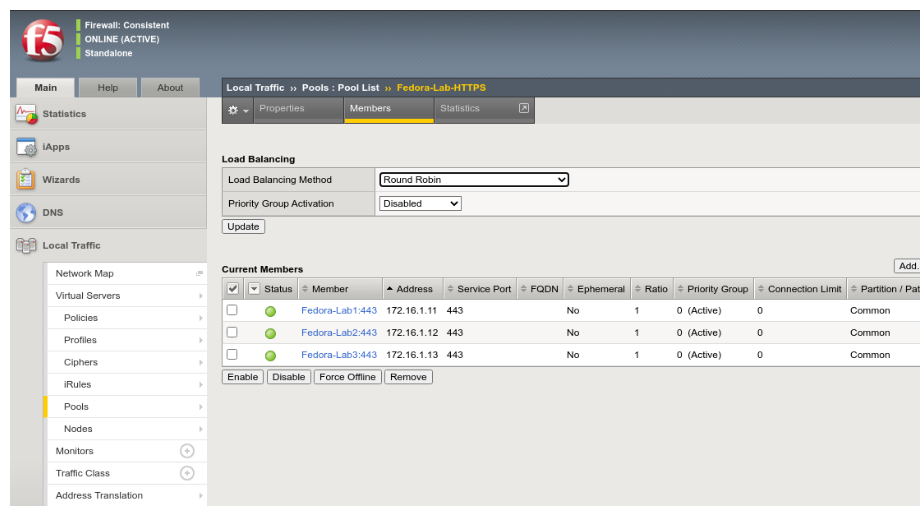


Figura 5.4: Definición de un *pool* en F5.

Además de este *pool*, e igual que pasará con los *virtual servers*, para este laboratorio se ha creado un *pool* a mayores para realizar el balanceo hacia el puerto 80/tcp del servidor que no utiliza TLS, se ha utilizado para pruebas y además muestra un caso de uso muy común. En muchos casos en entornos reales, se realiza el llamado *SSL Offloading* en el cual por varias razones, como pueden ser sobrecarga de los nodos o latencia por el uso continuado de TLS en varias capas de una aplicación web, el balanceador utiliza TLS pero los servidores web dentro del *data center* y en redes internas aisladas, sirven el contenido mediante HTTP sin ningún tipo de cifrado.

- Creación de los ***virtual servers***: el servidor virtual es el elemento al cual se conecta el cliente y sirve realmente el contenido. En el caso de este

laboratorio el componente importante es aquel que sirve la web por HTTPS. Este *virtual server* utiliza un perfil TLS hacia el cliente que únicamente soporta TLSv1.3 y hacia el servidor que utilizará cualquier versión que este utilice. A mayores, como se ha comentado, existen dos balanceos más para el caso del *SSL Offloading* y el caso de no utilizar cifrado alguno. En estos balanceadores se puede realizar cualquier cambio imaginable para cualquier caso de uso, pero en este caso la creación ha sido muy sencilla, se escucha en la IP y puerto indicados, se establecen los perfiles de SSL/TLS y se indica el *pool* destino hacia el cual balancear las peticiones. En la Figura 5.5 se puede observar la definición de un servidor virtual.

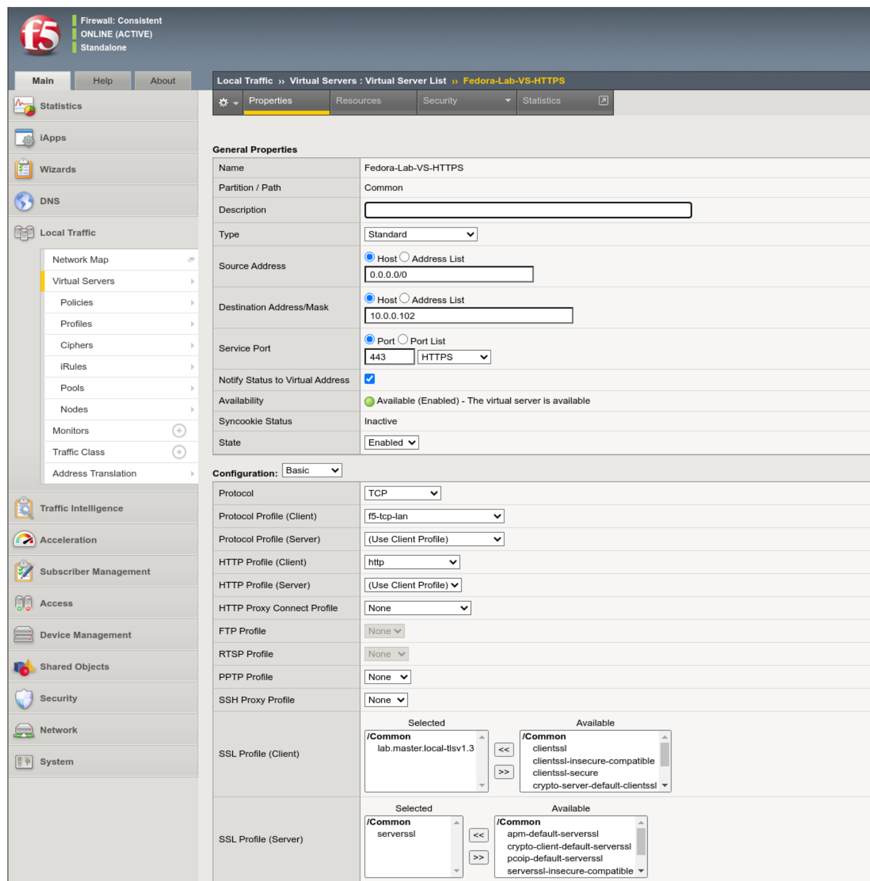


Figura 5.5: Definición de un *virtual server* en F5.

Como se ha comentado, existen otros servidores virtuales creados en este laboratorio que se muestran a continuación. Todos comparten la misma IP,

pero escuchan en puertos distintos.

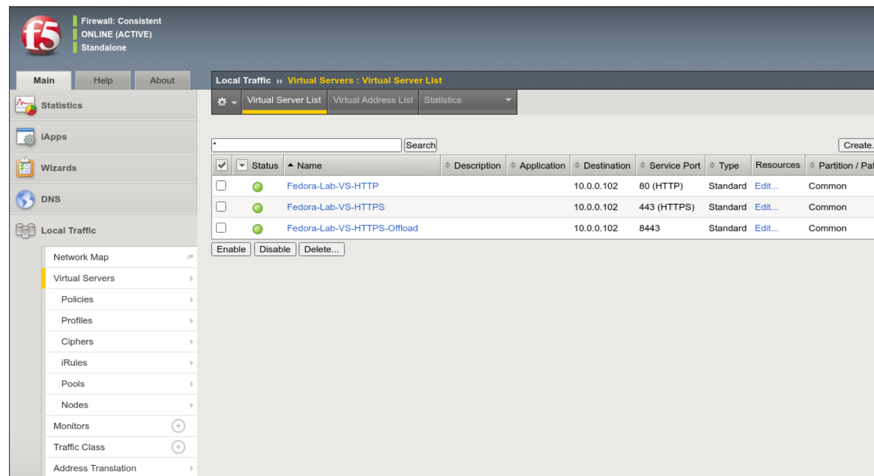


Figura 5.6: *Virtual servers* en F5.

- Subida de los **certificados**: puesto que el F5 va a presentar el certificado hacia el cliente, se deben subir los certificados creados al mismo, para posteriormente crear los distintos perfiles TLS que se vayan a utilizar y que se aplicarán en los *virtual servers*. La subida de los mismos es una simple importación y se muestran en la siguiente sección dentro de la interfaz web, en este caso lab.master.local es el certificado creado para este laboratorio.

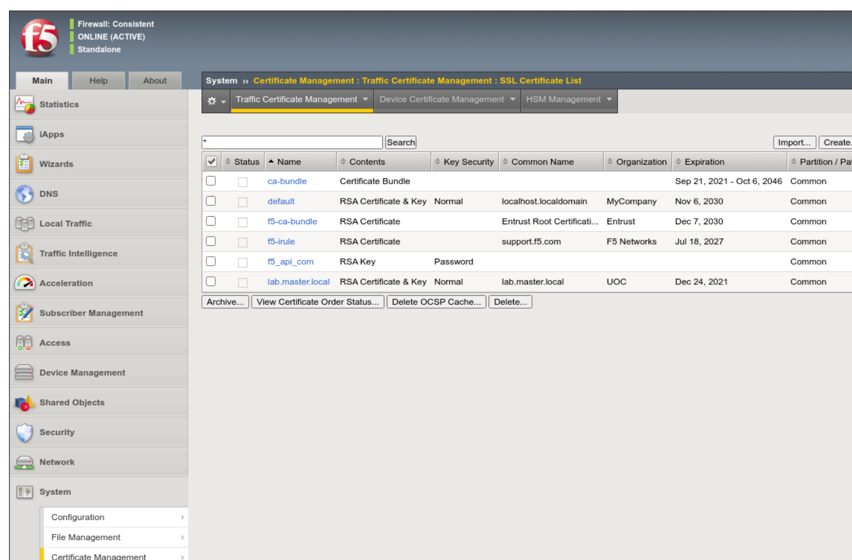


Figura 5.7: Certificados en F5.

- **Perfiles** de TLS: una vez los certificados están en el balanceador, el siguiente paso es crear los perfiles que se utilizarán hacia cliente y servidor. En los perfiles podemos escoger entre todas las opciones que nos ofrece F5 en lo referente a TLS. Lo más importante en el caso de este laboratorio es negar cualquier versión de TLS que no sea 1.3 para el perfil de cliente, y se aceptará cualquier protocolo que utilice el servidor. En la Figura 5.8 se puede apreciar la creación del perfil con el certificado y clave subidos y las opciones seleccionadas.

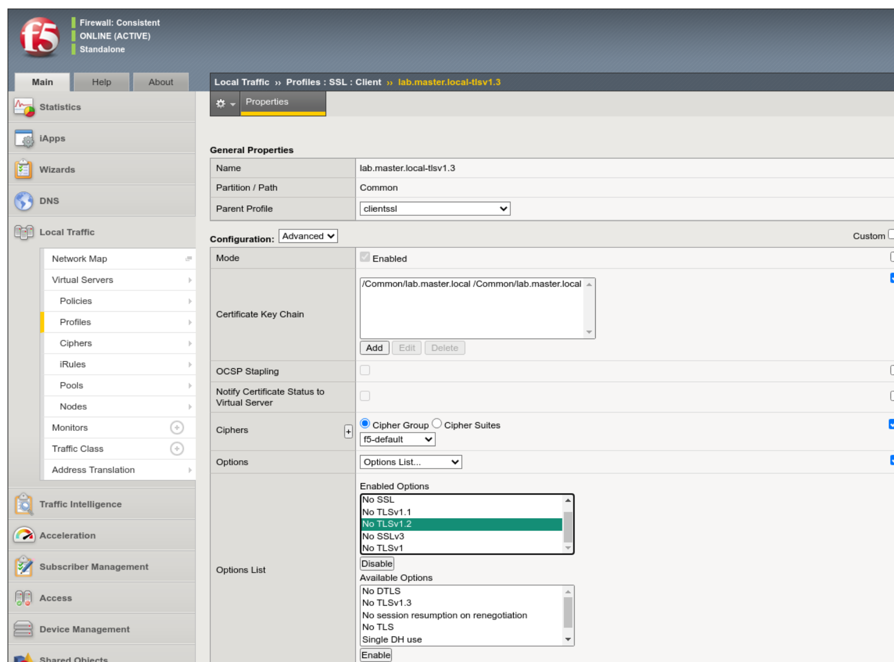


Figura 5.8: Perfil TLS en F5.

Este perfil se puede ver aplicado en el *virtual server* que se mostró en la Figura 5.5 en la sección *SSL Profile (Client)*. Hacia el servidor, el perfil *serverssl* que existe por defecto es más que suficiente y aceptará cualquier protocolo TLS (no así SSL).

Además de las versiones de SSL/TLS permitidas en las opciones, estos perfiles nos permite escoger que clase de *cipher suites* aceptaremos, como se verá más adelante, para este caso el *cipher group* f5-default ofrece ECDH TLS_AES_128_GCM_SHA256 y TLS_AES_256_GCM_SHA384 que son considerados seguros y cualquier navegador moderno soporta.

5.2.2. Configuración de los servidores web

Estos servidores tienen una instalación de Fedora 32 *Server* básica al cual se le han configurado las interfaces de gestión y servicio y se le han instalado y configurado los siguientes paquetes.

Como servidor web se ha seleccionado ***Nginx***, por su facilidad de configuración y al igual que con F5, por propio conocimiento del producto. Este servidor web es ampliamente utilizado (Cloudflare lo integra en toda su CDN por ejemplo) y es un *software* con un gran rendimiento. Además, se utiliza ***PHP*** como lenguaje de servidor para obtener y mostrar los datos del cliente que se conecta de una manera sencilla. A continuación, se muestran los primeros pasos de instalación.

```
# Instalación de los paquetes de nginx y php
sudo yum install nginx php-fpm php-cli
# Arranque de los servicios
sudo systemctl start nginx
sudo systemctl start php-fpm
# Arranque de los servicios en el encendido del servidor
sudo systemctl enable nginx
sudo systemctl enable php-fpm
```

Para la configuración básica de Nginx se ha optado por la configuración clásica con dos carpetas de *sites-enabled* y *sites-available*. En esta última se introducen los ficheros de configuración de todos los servicios web a los que el servidor podrá dar servicio. En *sites-enabled* se creará un enlace simbólico a los ficheros de *sites-available* para habilitar este servicio y que Nginx lo ejecute.

```
# Creación de las carpetas necesarias
mkdir /etc/nginx/sites-enabled
mkdir /etc/nginx/sites-available
# Eliminación de conf.d que no se va a utilizar
rmdir /etc/nginx/conf.d
# Se edita nginx.conf y se refleja la nueva configuración
vim /etc/nginx/nginx.conf
# Se incluye sites-enabled y se comenta conf.d

```

```
    # include /etc/nginx/conf.d/*.conf;
    ##
    # Virtual Host Configs
    ##
    include /etc/nginx/sites-enabled/*;
```

A partir de este momento, se continúa con la creación del certificado RSA que se utilizará para cifrar con TLS la comuniación. Para evitar complicaciones y puesto que no es el objetivo de este laboratorio, se ha obviado la creación de una autoridad certificadora y se ha optado por crear un certificado autofirmado directamente sobre el servidor con la utilidad ***openssl***. Además, para que la negociación de los grupos Diffie-Hellman sea segura, se genera una clave de 4096 bits para utilizar dentro de Nginx.

```
# Creación del certificado RSA con 365 días de caducidad
sudo openssl req -x509 -nodes -days 365 -newkey rsa:4096 -keyout /
    etc/ssl/private/lab.master.local.key -out /etc/ssl/certs/lab.
    master.local.crt
# Creación de una clave DH de 4096 bits
sudo openssl dhparam -out /etc/nginx/dhparam.pem 4096
```

Después de tener lo básico configurado, se realiza la configuración de Nginx del *site* de este laboratorio al que se ha llamado lab.master.local.

```
server {
  # El servidor web escucha en el puerto 80 sin SSL/TLS
  listen 80 default_server;
  # En este directorio se encuentran los ficheros del sitio web
  root /usr/share/nginx/lab/server;
  # Fichero que el servidor web busca por defecto cuando no se
    indica uno
  index index.html index.php;
  # Nombre para identificar el servidor virtual que hay que servir
  server_name lab.master.local fedora-lab1.master.local;

  # El servidor web escucha en el puerto 443 con TLS
```

```
    listen 443 ssl default_server;
13
    # Estos parámetros indican donde se encuentran el cert y clave
      privada
15  ssl_certificate /etc/ssl/certs/lab.master.local.crt;
    ssl_certificate_key /etc/ssl/private/lab.master.local.key;
17  # Para la negociación de Diffie-Hellman se indica el PEM generado
    ssl_dhparam /etc/nginx/dhparam-nginx.pem;
19  # Se indica la versión de TLS que se quiere utilizar
    ssl_protocols TLSv1.2;
21  ssl_prefer_server_ciphers on;
    # Se indican las cipher suites que se desean soportar en el
      servidor
23  ssl_ciphers "EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH";
    # Se habilita la caché de sesiones en el servidor y los session
      tickets para las pruebas
25  ssl_session_cache shared:SSL:10m;
    ssl_session_tickets on;
27
    # Para los ficheros PHP se indica donde encontrar el socket unix
      del mismo
29  location ~ \.php$ {
      try_files $uri =404;
31    fastcgi_pass  unix:/run/php-fpm/www.sock;
      fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name
     ;
33    fastcgi_index index.php;
      include  fastcgi_params;
35  }
}
```

Con esta configuración, reiniciando el servicio de Nginx ya se tendría configurada la web y se podría dar servicio. Esta configuración la tienen los tres servidores web, pero el certificado obviamente ha sido generado en uno y copiado a los demás y al propio F5, puesto que con certificados distintos no se podrían utilizar los *tickets* de sesión.

5.3. Pruebas realizadas

5.3.1. Diferencias de *Handshake* entre TLSv1.2 y 1.3

En esta prueba se ha realizado una petición HTTPS hacia el *virtual server* del balanceador y se ha guardado un pcap para su análisis. El F5 con un balanceo de tipo estándar utiliza una arquitectura *Full-Proxy*, esto significa que el cliente establecerá una conexión TCP contra el F5, y este último a su vez con los servidores, siendo dos conexiones completamente independientes. En la Figura 5.9 se puede apreciar esta arquitectura y el orden en la cual se realizan las conexiones.

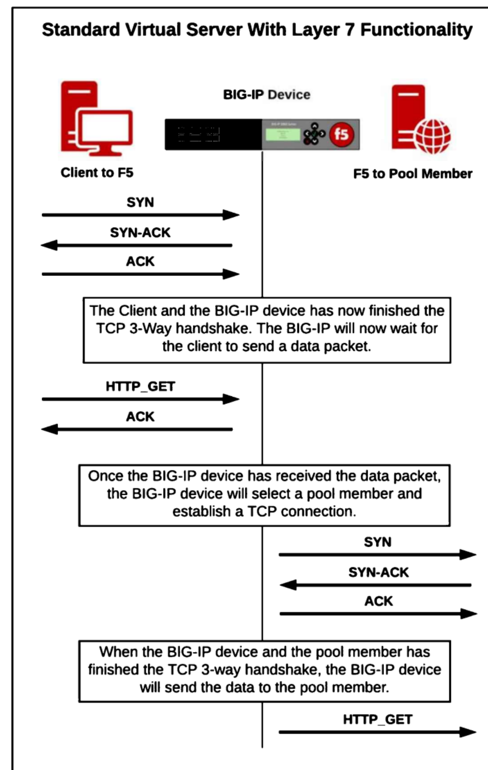


Figura 5.9: Arquitectura *Full-Proxy* en F5 [7].

5.3.1.1. Comunicación entre cliente y balanceador

En la comunicación entre el cliente web y el balanceador la captura muestra el establecimiento TCP y TLS. En la Figura 5.10 se puede apreciar la captura del establecimiento de sesión, empezando por el *three-way handshake* TCP, pasando

por los *Client y Server Hello* con sus ACKs TCP, moviendo los datos de aplicación y finalizando con el FIN, ACK y ACK.

No.	Time	Source	Destination	Protocol	Length	Info
238	18.863211546	10.0.0.103	10.0.0.102	TCP	74	27822 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=4068583444 TSecr=0 WS=2048
239	18.865087668	10.0.0.102	10.0.0.103	TCP	70	443 → 27822 [SYN, ACK] Seq=0 Ack=1 Win=23360 Len=0 MSS=1460 SACK_PERM=1 TSval=2540679130 TSecr=4068583444
240	18.865100278	10.0.0.103	10.0.0.102	TCP	66	27822 → 443 [ACK] Seq=1 Ack=1 Win=65535 Len=0 TSval=4068583446 TSecr=2540679130
241	18.865172958	10.0.0.103	10.0.0.102	TLSv1.3	583	Client Hello
242	18.874278452	10.0.0.102	10.0.0.103	TCP	66	443 → 27822 [ACK] Seq=1 Ack=518 Win=23877 Len=0 TSval=2540679133 TSecr=4068583446
243	18.886041605	10.0.0.102	10.0.0.103	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data
244	18.886049535	10.0.0.103	10.0.0.102	TCP	66	27822 → 443 [ACK] Seq=518 Ack=1449 Win=65535 Len=0 TSval=4068583467 TSecr=2540679151
245	18.886064015	10.0.0.102	10.0.0.103	TLSv1.3	967	Application Data, Application Data, Application Data
246	18.886066895	10.0.0.103	10.0.0.102	TCP	66	27822 → 443 [ACK] Seq=518 Ack=2350 Win=65535 Len=0 TSval=4068583467 TSecr=2540679151
247	18.886646802	10.0.0.103	10.0.0.102	TLSv1.3	96	Change Cipher Spec, Application Data
248	18.886697692	10.0.0.103	10.0.0.102	TCP	66	27822 → 443 [FIN, ACK] Seq=548 Ack=2350 Win=65535 Len=0 TSval=4068583468 TSecr=2540679151
250	18.888414665	10.0.0.102	10.0.0.103	TCP	66	443 → 27822 [ACK] Seq=2350 Ack=548 Win=23907 Len=0 TSval=2540679154 TSecr=4068583468
251	18.888414705	10.0.0.102	10.0.0.103	TCP	66	443 → 27822 [ACK] Seq=2350 Ack=549 Win=23907 Len=0 TSval=2540679154 TSecr=4068583468

Figura 5.10: Captura del *handshake* entre cliente y balanceador.

Si se observa la información que contiene este *ClientHello* se puede observar que grado de soporte ofrece el cliente. Como punto de partida la Figura 5.11 aporta información de la versión que anuncia el cliente (TLSv1.2, 0x0303) que es utilizada, como se ha comentado, tanto para negociar TLSv1.2 como 1.3. Lo siguiente que se puede apreciar, es la lista de *Cipher Suites* que el cliente puede utilizar y, por último, que no soporta métodos de compresión (TLSv1.3 no los utiliza por seguridad).

```
▸ Transmission Control Protocol, Src Port: 27822, Dst Port: 443, Seq: 1, Ack: 1, Len: 517
▾ Transport Layer Security
  ▾ TLSv1.3 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 512
    ▾ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 508
        Version: TLS 1.2 (0x0303)
        Random: 9ddde74d0fa758a0ee671293674eb7f871e332705bdfb6f8c9852929196afa07
        Session ID Length: 32
        Session ID: 52584d9f3f7ede6bf2da571b65132fcb7316fbc3eaf3b2c7b7cbd6b3c220259f
        Cipher Suites Length: 32
      ▾ Cipher Suites (16 suites)
          Cipher Suite: Reserved (GREASE) (0x8a8a)
          Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
          Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
          Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
          Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
          Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
          Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
          Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
          Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
        Compression Methods Length: 1
      ▾ Compression Methods (1 method)
          Compression Method: null (0)
```

Figura 5.11: Primera captura del *ClientHello* entre cliente y balanceador.

En las Figuras 5.12 y 5.13 se pueden apreciar las extensiones que anuncia el cliente. Entre las cuales encontramos:

- El ***server_name*** (SNI o *Server Name Indication*) indica a qué web está tratando de acceder el cliente, en este caso **`lab.master.local`**.
- Los ***supported_groups*** son los algoritmos de intercambio de clave que soporta el cliente ordenados por preferencia. Este parámetro ya indica soporte para TLSv1.3 (aunque hoy en día, también se ha incluido en implementaciones de versiones anteriores [22]), puesto que anteriormente esta extensión era enviada con el nombre ***elliptic_curves*** [9]. **x25519** es el algoritmo de curva elíptico preferido por el cliente.
- El ***application_layer_protocol_negotiation*** indica el protocolo de capa de aplicación del que se desea la comunicación con el servidor. En este caso HTTP/1.1 y HTTP/2.
- El ***signature_algorithms*** indica los algoritmos de firma soportados por el cliente.
- El ***key_share***, en TLSv1.3 el cliente comienza el paso *Client Key Exchange* dentro del *ClientHello* y comienza la negociación de claves suponiendo que el servidor pueda soportar el método seleccionado por el cliente, en el mejor de los casos lo hará, en el peor, el servidor reintentará el acuerdo con un mensaje *Retry Hello Request*. En este caso el cliente selecciona **x25519** como algoritmo preferido de intercambio de claves y envía al servidor su secreto para el comienzo de los cálculos.
- El ***supported_versions*** indica las versiones de TLS que soporta el cliente. En este caso Chrome anuncia soporte para cualquier versión de TLS.
- El ***session_ticket*** indica si existe *ticket* de sesión para resumir una conexión existente, en este caso no existe sesión anterior y además, en TLSv1.3 esta extensión se incluye por compatibilidad pero no sería utilizada, ya que fue sustituida por ***pre_shared_key*** [9].

```
▾ Extension: server_name (len=21)
    Type: server_name (0)
    Length: 21
  ▾ Server Name Indication extension
      Server Name list length: 19
      Server Name Type: host_name (0)
      Server Name length: 16
      Server Name: lab.master.local
▸ Extension: extended_master_secret (len=0)
▸ Extension: renegotiation_info (len=1)
▾ Extension: supported_groups (len=10)
    Type: supported_groups (10)
    Length: 10
    Supported Groups List Length: 8
  ▾ Supported Groups (4 groups)
      Supported Group: Reserved (GREASE) (0x4a4a)
      Supported Group: x25519 (0x001d)
      Supported Group: secp256r1 (0x0017)
      Supported Group: secp384r1 (0x0018)
▸ Extension: ec_point_formats (len=2)
▸ Extension: session_ticket (len=0)
▾ Extension: application_layer_protocol_negotiation (len=14)
    Type: application_layer_protocol_negotiation (16)
    Length: 14
    ALPN Extension Length: 12
  ▾ ALPN Protocol
      ALPN string length: 2
      ALPN Next Protocol: h2
      ALPN string length: 8
      ALPN Next Protocol: http/1.1
▸ Extension: status_request (len=5)
▸ Extension: signature_algorithms (len=18)
▸ Extension: signed_certificate_timestamp (len=0)
▸ Extension: key_share (len=43)
▸ Extension: psk_key_exchange_modes (len=2)
▸ Extension: supported_versions (len=11)
▸ Extension: compress_certificate (len=3)
▸ Extension: Reserved (GREASE) (len=1)
```

Figura 5.12: Segunda captura del *ClientHello* entre cliente y balanceador.

```
▸ Extension: session_ticket (len=0)
▸ Extension: application_layer_protocol_negotiation (len=14)
▸ Extension: status_request (len=5)
▾ Extension: signature_algorithms (len=18)
    Type: signature_algorithms (13)
    Length: 18
    Signature Hash Algorithms Length: 16
  ▾ Signature Hash Algorithms (8 algorithms)
    ▸ Signature Algorithm: ecdsa_secp256r1_sha256 (0x0403)
    ▸ Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
    ▸ Signature Algorithm: rsa_pkcs1_sha256 (0x0401)
    ▸ Signature Algorithm: ecdsa_secp384r1_sha384 (0x0503)
    ▸ Signature Algorithm: rsa_pss_rsae_sha384 (0x0805)
    ▸ Signature Algorithm: rsa_pkcs1_sha384 (0x0501)
    ▸ Signature Algorithm: rsa_pss_rsae_sha512 (0x0806)
    ▸ Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
▸ Extension: signed_certificate_timestamp (len=0)
▾ Extension: key_share (len=43)
    Type: key_share (51)
    Length: 43
  ▾ Key Share extension
      Client Key Share Length: 41
    ▾ Key Share Entry: Group: Reserved (GREASE), Key Exchange length: 1
        Group: Reserved (GREASE) (19018)
        Key Exchange Length: 1
        Key Exchange: 00
    ▾ Key Share Entry: Group: x25519, Key Exchange length: 32
        Group: x25519 (29)
        Key Exchange Length: 32
        Key Exchange: 0c4b23504cd02fde11747d12848e46b2ed24bd1256b37bc36c28457fe907dd67
▸ Extension: psk_key_exchange_modes (len=2)
▾ Extension: supported_versions (len=11)
    Type: supported_versions (43)
    Length: 11
    Supported Versions length: 10
    Supported Version: Unknown (0x8a8a)
    Supported Version: TLS 1.3 (0x0304)
    Supported Version: TLS 1.2 (0x0303)
    Supported Version: TLS 1.1 (0x0302)
    Supported Version: TLS 1.0 (0x0301)
▸ Extension: compress_certificate (len=3)
```

Figura 5.13: Tercera captura del *ClientHello* entre cliente y balanceador.

Después de recibir el *ClientHello*, el balanceador contesta con un *ServerHello* con la información que se puede apreciar en la Figura 5.14.

```
Transmission Control Protocol, Src Port: 443, Dst Port: 27822, Seq: 1, Ack: 518, Len: 1448
Transport Layer Security
  TLSv1.3 Record Layer: Handshake Protocol: Server Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 122
    Handshake Protocol: Server Hello
      Handshake Type: Server Hello (2)
      Length: 118
      Version: TLS 1.2 (0x0303)
      Random: 57788b61f92e5887247f14b1e5619239fc7adf0d0a1350dd1e4782ef0b10d112
      Session ID Length: 32
      Session ID: 52584d9f3f7ede6bf2da571b65132fcb7316fbc3eaf3b2c7b7cbd6b3c220259f
      Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
      Compression Method: null (0)
      Extensions Length: 46
      Extension: key_share (len=36)
        Type: key_share (51)
        Length: 36
        Key Share extension
          Key Share Entry: Group: x25519, Key Exchange length: 32
            Group: x25519 (29)
            Key Exchange Length: 32
            Key Exchange: 644b0ea1974ee612d3676a0da52f50716521e5d0967a54d97252a90977a85d32
      Extension: supported_versions (len=2)
        Type: supported_versions (43)
        Length: 2
        Supported Version: TLS 1.3 (0x0304)
  TLSv1.3 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
    Content Type: Change Cipher Spec (20)
    Version: TLS 1.2 (0x0303)
    Length: 1
    Change Cipher Spec Message
  TLSv1.3 Record Layer: Application Data Protocol: http-over-tls
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 23
    Encrypted Application Data: 77f861ce4fd03a12e861193d01367bf6e748712be9d472
    [Application Data Protocol: http-over-tls]
```

Figura 5.14: Captura del *ServerHello* entre el balanceador y el cliente.

En esta captura se puede observar que el servidor selecciona el conjunto TLS_AES_128_GCM_SHA256 y que el intercambio de claves se lleva a cabo con **x25519** (el cliente ha acertado). En *supported_versions* anuncia soporte únicamente para TLSv1.3. En el campo *Session ID* se puede observar como el servidor contesta con el mismo valor que envió el cliente, recordar que este parámetro en TLSv1.3 ya no se utiliza, y se mantiene simplemente por compatibilidad.

A partir del *ServerHello* se puede apreciar que el siguiente paquete ya contiene datos de aplicación, y aunque no se pueden ver los datos cifrados, también tienen que existir los registros que llevan los mensajes *Finished* por cada parte. Este es el *handshake* reducido de TLSv1.3 que se observa en la Figura 3.11.

5.3.1.2. Comunicación entre balanceador y servidores

En este apartado se detalla la comunicación entre el balanceador y los servidores web que utilizan TLSv1.2. En la Figura 5.15 se puede apreciar la captura del establecimiento de sesión entre el balanceador y el servidor fedora-lab1.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.1.1	172.16.1.11	TCP	70	56673 → 443 [SYN] Seq=0 Win=23360 Len=0 MSS=1460 SACK_PERM=1 TSval=2549300249 TSecr=0
2	0.000055	172.16.1.11	172.16.1.1	TCP	70	443 → 56673 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 SACK_PERM=1 TSval=1419520489 TSecr=2549300249
3	0.000275	172.16.1.1	172.16.1.11	TCP	66	56673 → 443 [ACK] Seq=1 Ack=1 Win=23360 Len=0 TSval=2549300249 TSecr=1419520489
4	0.000320	172.16.1.1	172.16.1.11	TLSv1.2	253	Client Hello
5	0.000335	172.16.1.11	172.16.1.1	TCP	66	443 → 56673 [ACK] Seq=1 Ack=188 Win=65535 Len=0 TSval=1419520489 TSecr=2549300249
6	0.020968	172.16.1.11	172.16.1.1	TLSv1.2	2371	Server Hello, Certificate, Server Key Exchange, Server Hello Done
8	0.032375	172.16.1.1	172.16.1.11	TCP	66	56673 → 443 [ACK] Seq=188 Ack=2306 Win=25665 Len=0 TSval=2549300281 TSecr=1419520509
9	0.035868	172.16.1.1	172.16.1.11	TLSv1.2	224	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
10	0.035881	172.16.1.11	172.16.1.1	TCP	66	443 → 56673 [ACK] Seq=2306 Ack=346 Win=65535 Len=0 TSval=1419520524 TSecr=2549300285
11	0.038708	172.16.1.11	172.16.1.1	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message
12	0.041540	172.16.1.1	172.16.1.11	TCP	66	56673 → 443 [ACK] Seq=346 Ack=2357 Win=25716 Len=0 TSval=2549300290 TSecr=1419520527
14	0.041731	172.16.1.1	172.16.1.11	TLSv1.2	597	Application Data

Figura 5.15: Captura del *handshake* entre el balanceador y un servidor.

En la Figura 5.16 se puede observar la información del *ClientHello* que el balanceador envía a los servidores.

```
Transport Layer Security
  TLSv1.2 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 182
    Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 178
      Version: TLS 1.2 (0x0303)
      Random: 665af0a10d8cbfed3e24877a2835a910f03badcd34a2e1a4b41bdbf72e9fc1e5
      Session ID Length: 0
      Cipher Suites Length: 58
      Cipher Suites (29 suites)
      Compression Methods Length: 1
      Compression Methods (1 method)
      Extensions Length: 79
      Extension: server_name (len=21)
        Type: server_name (0)
        Length: 21
        Server Name Indication extension
      Extension: supported_groups (len=8)
        Type: supported_groups (10)
        Length: 8
        Supported Groups List Length: 6
        Supported Groups (3 groups)
          Supported Group: secp256r1 (0x0017)
          Supported Group: x25519 (0x001d)
          Supported Group: secp384r1 (0x0018)
      Extension: ec_point_formats (len=2)
      Extension: signature_algorithms (len=28)
      Extension: extended_master_secret (len=0)
```

Figura 5.16: Captura del *ClientHello* entre balanceador y servidor.

La versión anunciada es TLSv1.2 (igual que en el caso anterior). Una diferencia sustancial es que el *Session ID* va vacío, puesto que no es un restablecimiento de conexión. El SNI viaja de la misma manera que TLSv1.3, y se puede observar que esta implementación de TLSv1.2 utiliza la extensión *supported_groups* (o que así es marcada por Wireshark). En lo referente a los *cipher suites* vemos que aparecen 29 posibilidades, esto es así porque el F5 utiliza un perfil de compatibilidad que acepta multitud de conjuntos (esto podría cambiarse de una manera muy rápida en el perfil de SSL/TLS hacia el servidor).

En el caso del *ServerHello* la información encontrada en la Figura 5.17 indica soporte para TLSv1.2. Ha aceptado el conjunto TLS_ECDHE_RSA_WITH_AES _256_GCM_SHA384 (0xc030) y ha generado un *Session ID* para el posible restablecimiento de la sesión TLS. En el mismo segmento TCP se han enviado varios registros TLS. En este se han enviado los mensajes con el certificado del servidor, el intercambio de claves y la marca de final del *ServerHello*.

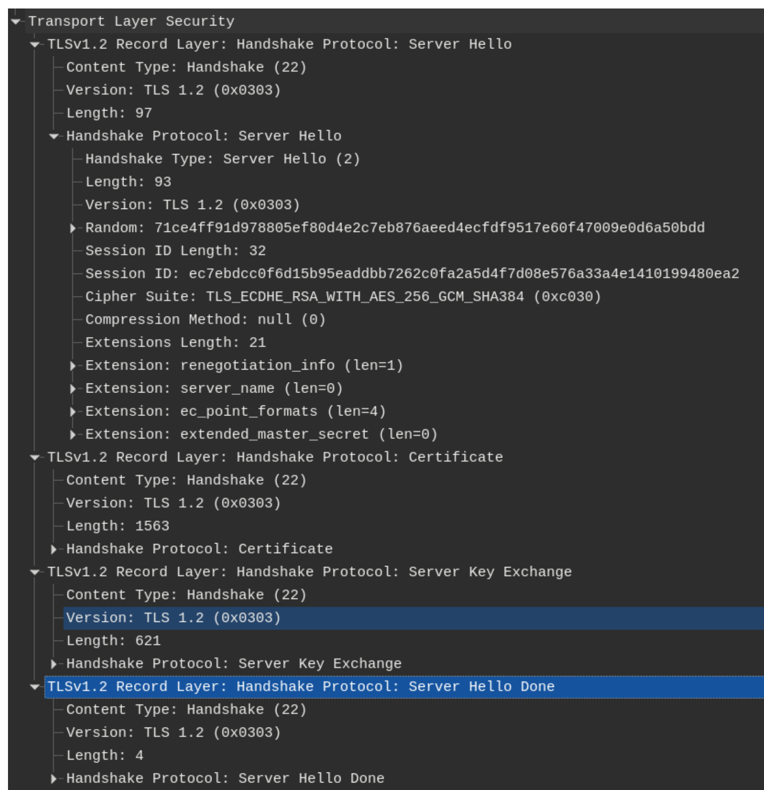


Figura 5.17: Captura del *ServerHello* entre servidor y balanceador.

Además, el balanceador envía otro segmento con los mensajes: *Client Key Exchange*, *Change Cipher Spec* y *Finished*. A continuación el servidor envía su último segmento TCP de *handshake* con los mensajes *Change Cipher Spec* y *Finished*, con esto el establecimiento de la sesión TLS habría acabado. Como se ha podido apreciar, este establecimiento de *handshake* consume 3-RTT entre TCP y TLS a diferencia del ahorro acaecido con TLSv1.3.

5.3.2. Resumen de sesión en TLSv1.2 vs TLSv1.3

Comenzando con TLSv1.2, y como ya se comentó en el Capítulo 3, existen dos formas de resumir la sesión. La primera es la utilización de los *Session ID*. El cliente en la primera comunicación envía este valor vacío al no existir una sesión para resumir; el servidor envía un identificador para esta sesión y almacena en su caché el material criptográfico de la sesión durante un período de tiempo (o tamaño de sesiones) configurable.

```
tls.handshake.session_id == 98:39:fb:06:63:cd:41:b7:1b:5a:85:bd:64:36:14:b3:94:c3:0a:d6:3f:88:a1:5c:ac:93:89:6c:33:87:6a:50
No.  Time       Source       Destination  Protocol  Length Info
  6  0.029112   172.16.1.11  172.16.1.1   TLSv1.2   2371 Server Hello, Certificate, Server Key Exchange, Server Hello Done
329  88.455857  172.16.1.1   172.16.1.11  TLSv1.2    289 Client Hello
341  88.478159  172.16.1.11  172.16.1.1   TLSv1.2    207 Server Hello, Change Cipher Spec, Encrypted Handshake Message

Frame 341: 207 bytes on wire (1656 bits), 207 bytes captured (1656 bits)
Ethernet II, Src: VMware_0a:2e:50 (00:0c:29:0a:2e:50), Dst: VMware_09:f3:ed (00:0c:29:09:f3:ed)
Internet Protocol Version 4, Src: 172.16.1.11, Dst: 172.16.1.1
Transmission Control Protocol, Src Port: 443, Dst Port: 37971, Seq: 1, Ack: 224, Len: 141
Transport Layer Security
  TLSv1.2 Record Layer: Handshake Protocol: Server Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 85
    Handshake Protocol: Server Hello
      Handshake Type: Server Hello (2)
      Length: 81
      Version: TLS 1.2 (0x0303)
      Random: ed0b32eb77cabaf273d9503606678263c22145aad87854df81704f65fd1744e1
      Session ID Length: 32
      Session ID: 9839fb0663cd41b71b5a85bd643614b394c30ad63f88a15cac93896c33876a50
      Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
      Compression Method: null (0)
      Extensions Length: 9
      Extension: renegotiation_info (len=1)
        Type: renegotiation_info (65281)
        Length: 1
        Renegotiation Info extension
      Extension: extended_master_secret (len=0)
        Type: extended_master_secret (23)
        Length: 0
  TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
    Content Type: Change Cipher Spec (20)
    Version: TLS 1.2 (0x0303)
    Length: 1
    Change Cipher Spec Message
      [Expert Info (Note/Sequence): This session reuses previously negotiated keys (Session resumption)]
        [This session reuses previously negotiated keys (Session resumption)]
        [Severity level: Note]
        [Group: Sequence]
  TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 40
    Handshake Protocol: Encrypted Handshake Message
```

Figura 5.18: Resumen de sesión mediante el *Session ID*.

En el caso de este laboratorio, los servidores web (Nginx) tienen configurada una caché SSL con la directiva ssl_session_cache. Se ha capturado el tráfico entre el balanceador y un servidor. Después de varias peticiones desde el cliente (hacia el balanceador) se aprecia que se empiezan a reutilizar las sesiones por parte del balanceador, y el servidor tiene en caché esas sesiones, las cuales resume. En la Figura 5.18 se puede apreciar la reutilización de una de las sesiones.

La segunda forma de resumir una sesión en TLSv1.2 es la utilización de *Session tickets*, donde es el cliente el que guarda el material criptográfico de la sesión y se lo envía al servidor en un futuro *ClientHello* para resumir la sesión (en caso de ser posible). En este laboratorio, para forzar su uso, se ha deshabilitado la caché de sesiones y se han habilitado los *tickets* de sesión en Nginx con la directiva ssl_session_tickets. En la Figura 5.19 se puede apreciar un establecimiento de sesión entre el balanceador y uno de los servidores web. En esta sesión TLS recibe un segmento con el *ticket* de sesión para ser almacenado y usado a futuro.

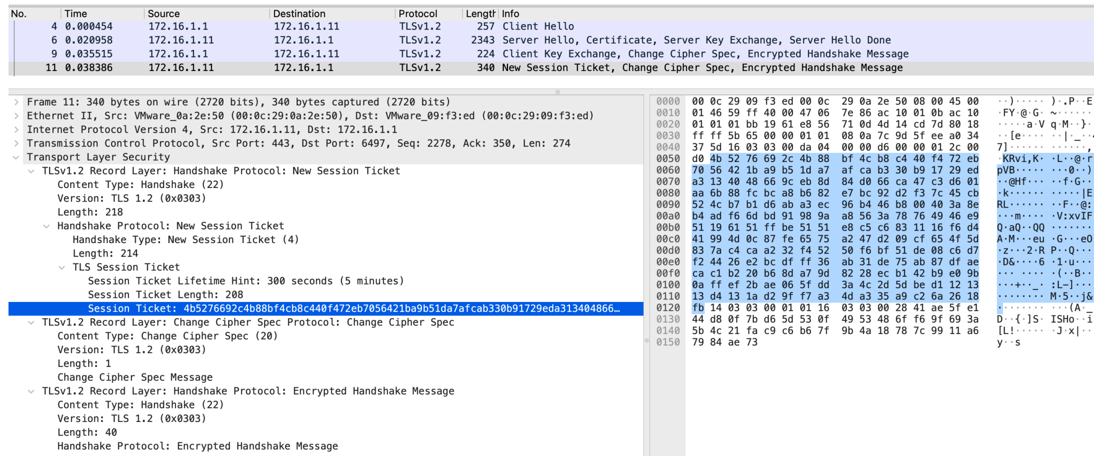


Figura 5.19: Envío de un *Session ticket* al balanceador.

Cuando el cliente quiere resumir la sesión envía este *ticket* en la extensión, y resume la sesión si esta no ha expirado para el servidor. En la Figura 5.20 se puede observar el resumen de la sesión con el *ticket* que el cliente tenía guardado de la sesión anterior. El servidor al recibir el *ticket* válido, resume la sesión con el cliente (en este caso el balanceador). En este caso, a diferencia de la caché de sesiones que se almacena en cada servidor, el *ticket* podría usarse contra cualquier

servidor web, ya que todos tienen el mismo certificado.

No.	Time	Source	Destination	Protocol	Length	Info
139	0.276730	172.16.1.1	172.16.1.11	TLSv1.2	583	Client Hello
141	0.277019	172.16.1.11	172.16.1.1	TLSv1.2	175	Server Hello, Change Cipher Spec, Encrypted Handshake Message
143	0.279343	172.16.1.1	172.16.1.11	TLSv1.2	117	Change Cipher Spec, Encrypted Handshake Message

```
> Frame 139: 583 bytes on wire (4664 bits), 583 bytes captured (4664 bits)
> Ethernet II, Src: VMware_09:f3:ed (00:0c:29:09:f3:ed), Dst: VMware_0a:2e:50 (00:0c:29:0a:2e:50)
> Internet Protocol Version 4, Src: 172.16.1.1, Dst: 172.16.1.11
> Transmission Control Protocol, Src Port: 6502, Dst Port: 443, Seq: 1, Ack: 1, Len: 517
v Transport Layer Security
  v TLSv1.2 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 512
    v Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 508
        Version: TLS 1.2 (0x0303)
      > Random: 2d1ea4b5dd0c3ce568d4d59834aea0238a37d552cec7992d2070ad5d5e45dd28
        Session ID Length: 0
        Cipher Suites Length: 58
      > Cipher Suites (29 suites)
        Compression Methods Length: 1
      > Compression Methods (1 method)
        Extensions Length: 409
      > Extension: server_name (len=21)
      v Extension: session_ticket (len=208)
          Type: session_ticket (35)
          Length: 208
          Data (208 bytes)
      > Extension: supported_groups (len=8)
      > Extension: ec_point_formats (len=2)
      > Extension: signature_algorithms (len=28)
      > Extension: extended_master_secret (len=0)
      > Extension: padding (len=114)
```

Figura 5.20: Resumen de sesión utilizando un *ticket*.

En TLSv1.3, como se ha comentado anteriormente, no existe la reutilización de sesión por ID, y los *tickets* han sido sustituidos por las claves precompartidas. En esta prueba, se va a realizar el resumen de sesión con una PSK en el modo normal (1-RTT), en la siguiente sección se explicará el nuevo modo 0-RTT.

Esta prueba se ha realizado utilizando el comando openssl de la siguiente manera:

```
# Creación de la petición
echo -e "HEAD / HTTP/1.1\r\nHost: lab.master.local\r\nConnection:
    close\r\n\r\n" > request.txt
# Primera petición guardando el ticket de sesión
openssl s_client -connect lab.master.local:443 -tls1_3 -sess_out
    ticket -ign_eof < request.txt
# Segunda petición utilizando el ticket de sesión guardado
openssl s_client -connect lab.master.local:443 -tls1_3 -sess_in
    ticket -ign_eof < request.txt
```

Se omite el primer *ClientHello* que no contiene información importante. La información sería la misma a la mostrada en el *handshake* de la Sección 5.3.1.1.

No.	Time	Source	Destination	Protocol	Length	Info
62	12.006488	10.0.0.103	10.0.0.102	TLSv1.3	306	Client Hello
64	12.037023	10.0.0.102	10.0.0.103	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data
65	12.037036	10.0.0.102	10.0.0.103	TLSv1.3	967	Application Data, Application Data, Application Data
67	12.039609	10.0.0.103	10.0.0.102	TLSv1.3	130	Change Cipher Spec, Application Data
69	12.044579	10.0.0.103	10.0.0.102	TLSv1.3	151	Application Data
70	12.045310	10.0.0.102	10.0.0.103	TLSv1.3	386	Application Data
82	12.101531	10.0.0.102	10.0.0.103	TLSv1.3	387	Application Data
86	12.102073	10.0.0.103	10.0.0.102	TLSv1.3	90	Application Data
129	20.104529	10.0.0.103	10.0.0.102	TLSv1.3	625	Client Hello
131	20.109002	10.0.0.102	10.0.0.103	TLSv1.3	291	Server Hello, Change Cipher Spec, Application Data, Application Data
133	20.109687	10.0.0.103	10.0.0.102	TLSv1.3	130	Change Cipher Spec, Application Data
135	20.120278	10.0.0.102	10.0.0.103	TLSv1.3	386	Application Data

```
> Frame 129: 625 bytes on wire (5000 bits), 625 bytes captured (5000 bits) on interface en0, id 0
> Ethernet II, Src: Apple_bb:ef:0d (c4:b3:01:bb:ef:0d), Dst: VMware_09:f3:e3 (00:0c:29:09:f3:e3)
> Internet Protocol Version 4, Src: 10.0.0.103, Dst: 10.0.0.102
> Transmission Control Protocol, Src Port: 51229, Dst Port: 443, Seq: 1, Ack: 1, Len: 559
v Transport Layer Security
  v TLSv1.3 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 554
    v Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 550
        Version: TLS 1.2 (0x0303)
        Random: e011bfee4ee2d5f0ff5dda00dda68fa4558f84757407fd2d2e2c20fcbbab0bbf
        Session ID Length: 32
        Session ID: e8362e2a7d5d7679af77b6389b35b25d8e7794555a3c0cfebc95c5ead490fc9e
        Cipher Suites Length: 8
      > Cipher Suites (4 suites)
        Compression Methods Length: 1
      > Compression Methods (1 method)
        Extensions Length: 469
      > Extension: server_name (len=21)
      > Extension: ec_point_formats (len=4)
      > Extension: supported_groups (len=12)
      > Extension: session_ticket (len=0)
      > Extension: encrypt_then_mac (len=0)
      > Extension: extended_master_secret (len=0)
      > Extension: signature_algorithms (len=30)
      > Extension: supported_versions (len=3)
      > Extension: psk_key_exchange_modes (len=2)
      > Extension: key_share (len=38)
      v Extension: pre_shared_key (len=315)
          Type: pre_shared_key (41)
          Length: 315
        > Pre-Shared Key extension
```

Figura 5.21: Resumen de sesión utilizando una PSK en TLSv1.3.

En la Figura 5.21 podemos apreciar el resumen de sesión en el segundo *Client-Hello* introduciendo la extensión pre_shared_key con la información de la clave precompartida, que es aceptada por el servidor para el resumen de la sesión.

Este resumen de sesión con 1-RTT es el método más seguro de restablecimiento, puesto que provee de protección anti *replay* y envía los datos una vez se ha vuelto a establecer la conexión con el servidor.

5.3.3. Resumen de sesión 0-RTT con TLSv1.3

TLSv1.3 introduce el nuevo modo 0-RTT explicado en la sección 3.4.1.3. La petición generada es la misma que en el caso anterior, pero además, se han introducido los datos a enviar para este modo.

```
1 # Segunda petición utilizando el ticket guardado con early_data
openssl s_client -connect lab.master.local:443 -tls1_3 -sess_in
    ticket -early_data request.txt
```

En la Figura 5.22 se puede observar un *ClientHello* realizando un resumen de sesión. Además, como se ha seleccionado el modo 0-RTT, al introducir datos para enviar con el resumen se mete la extensión early_data y se envían los datos en el primer segmento junto con el mensaje de *Hello*.

No.	Time	Source	Destination	Protocol	Length	Info
246…	566.180880	10.0.0.103	10.0.0.102	TLSv1.3	306	Client Hello
246…	566.210757	10.0.0.102	10.0.0.103	TLSv1.3	1514	Server Hello, Change Cipher Spec, Application Data
246…	566.210764	10.0.0.102	10.0.0.103	TLSv1.3	967	Application Data, Application Data, Application Data
246…	566.213173	10.0.0.103	10.0.0.102	TLSv1.3	130	Change Cipher Spec, Application Data
246…	566.220418	10.0.0.103	10.0.0.102	TLSv1.3	151	Application Data
246…	566.220546	10.0.0.102	10.0.0.103	TLSv1.3	386	Application Data
246…	566.270494	10.0.0.102	10.0.0.103	TLSv1.3	387	Application Data
246…	566.271325	10.0.0.103	10.0.0.102	TLSv1.3	90	Application Data
369…	618.639423	10.0.0.103	10.0.0.102	TLSv1.3	720	Client Hello, Change Cipher Spec, Application Data
369…	618.653863	10.0.0.102	10.0.0.103	TLSv1.3	295	Server Hello, Change Cipher Spec, Application Data,
369…	618.654540	10.0.0.103	10.0.0.102	TLSv1.3	150	Application Data, Application Data

```
Transport Layer Security
  TLSv1.3 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 558
    Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 554
      Version: TLS 1.2 (0x0303)
      Random: f1e53ff090c94ea6ae2fede214d75aea4213a1c79eb239313b17191701983f8d
      Session ID Length: 32
      Session ID: 23154448d3ab84465beb421f3c171311fc1e1350d134aa5db53f28bbaf1cd00e
      Cipher Suites Length: 8
      Cipher Suites (4 suites)
      Compression Methods Length: 1
      Compression Methods (1 method)
      Extensions Length: 473
      Extension: server_name (len=21)
      Extension: ec_point_formats (len=4)
      Extension: supported_groups (len=12)
      Extension: session_ticket (len=0)
      Extension: encrypt_then_mac (len=0)
      Extension: extended_master_secret (len=0)
      Extension: signature_algorithms (len=30)
      Extension: supported_versions (len=3)
      Extension: psk_key_exchange_modes (len=2)
      Extension: key_share (len=38)
      Extension: early_data (len=0)
      Extension: pre_shared_key (len=315)
        Type: pre_shared_key (41)
        Length: 315
        Pre-Shared Key extension
          Identities Length: 278
          PSK Identity (length: 272)
            Identity Length: 272
            Identity: 9006907473ae2461e68aec23a32725a8eefd42a243d58d74a1e61f82c6086a8f3208d3ca…
            Obfuscated Ticket Age: 1613774617
          PSK Binders length: 33
          PSK Binders
  TLSv1.3 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
  TLSv1.3 Record Layer: Application Data Protocol: http-over-tls
```

Figura 5.22: Resumen de sesión 0-RTT con TLSv1.3.

5.3.4. *Downgrade* en TLSv1.3

Como se comentó en la Sección 3.4.1.4, TLSv1.3 introduce una protección *anti-downgrade* a modo de mensaje oculto. En el *ServerHello* cuando un servidor que soporta la nueva versión se ve obligado a utilizar TLSv1.2 porque es la versión que le llega anunciada por el cliente, introduce un mensaje en el valor *ServerRandom* cuyos bytes en ASCII serían DOWNGRD01.

Para alcanzar este objetivo se ha habilitado TLSv1.2 en el perfil SSL/TLS en el balanceador y se ha enviado una petición con openssl.

```
# Petición con TLSv1.2 hacia el virtual server del balanceador
openssl s_client -connect lab.master.local:443 -tls1_2
```

El balanceador, al tener soporte para TLSv1.3, desconoce si esta petición viene del cliente o de un tercero que trata de realizar el *downgrade* de la conexión, con lo cual, incluye el mensaje que se comentó anteriormente. En la Figura 5.23 se puede apreciar que entre los datos aleatorios se encuentra el mensaje.

Figura 5.23: *Anti-Downgrade* en TLSv1.3.

Capítulo 6
Conclusiones

Índice general

6.1. Líneas de trabajo futuro 96

El proyecto sobre el que trata la presente memoria es un estudio sobre la versión 1.3 del protocolo TLS. Esta versión era muy necesaria desde el punto de vista de la seguridad y del rendimiento.

Desde el punto de vista de la seguridad, se ha realizado una revisión de los distintos estándares desde SSLv3, primera versión comercial, hasta el análisis de TLSv1.3. Además, se han revisado las vulnerabilidades más graves que han afectado a SSL/TLS a lo largo de su historia.

Lo más importante de esta nueva versión es que los únicos *cipher suites* que soporta son seguros. TLSv1.2 y las versiones anteriores cometieron el error de que, en aras de la máxima compatibilidad, permitieron algoritmos de intercambio, simétricos y *hmac* que no eran seguros o que se sabía que podían tener problemas en un futuro muy cercano. TLSv1.3 corrigió esto permitiendo únicamente los algoritmos considerados seguros y además, eliminó cualquier intercambio que no proveyera *forward secrecy*. La protección *anti-downgrade* de la manera que ha sido implementada también aporta un punto interesante de seguridad.

Aparte de la seguridad, una deuda pendiente de un nuevo estándar de TLS era mejorar el rendimiento, puesto que representaba un punto de lentitud en las comunicaciones, sobre todo en aquellas con una alta latencia de acceso. La mejora

del *handshake* ha ahorrado como mínimo la latencia de 1-RTT, y además, el modo 0-RTT, si se implementa de manera correcta, puede aportar una velocidad de resumen de sesión increíblemente rápida.

6.1. Líneas de trabajo futuro

Como líneas de trabajo futuro desde el punto de vista de la seguridad, el modo 0-RTT plantea serias dudas respecto a las implementaciones que lo permitan y no sean idempotentes, un ataque de *replay* puede producir daños en la plataforma que da el servicio. Otro caso de análisis sería el intento de realizar *downgrade* a la conexión y ver la respuesta de cada implementación *software*.

Además, desde el punto de vista del rendimiento, se podrían realizar pruebas de rendimiento exhaustivas, las cuales no fueron planteadas en la realización de este trabajo. Estas pruebas, deberían realizarse con latencias relativamente altas, simulando a un cliente normal accediendo a través de internet a una plataforma web. Pero también, con unas latencias reducidas, para medir las mejoras que ha podido aportar TLSv1.3 en *data center*, donde en muchas ocasiones se evita el uso de TLS para reducir los tiempos de acceso.

Apéndice A

Anexo I: *Cipher suites* obligatorias (SSLv3 y TLSv1.0)

En este anexo se incluyen los conjuntos de cifrado de las primeras versiones de SSL y TLS utilizadas además de las establecidas para TLSv1.3, esto aportará una visión de qué clase de algoritmos de intercambio de clave, simétricos o *hmac* se utilizaban en cada época. A partir de TLSv1.1 los conjuntos de cifrado se ampliaron considerablemente y en TLSv1.3 se ha reducido considerablemente ante la eliminación de todos aquellos algoritmos obsoletos.

A.1. *Cipher suites* soportadas por SSLv3

En esta sección se muestran, sin entrar en profundidad, los conjuntos de cifrado soportados por esta versión y su valor hexadecimal que se enviaría realmente en los mensajes de *Hello*.

***Cipher suites* en SSLv3**	
Valor	***Cipher suite***
0x00	SSL_NULL_WITH_NULL_NULL
0x01	SSL_RSA_WITH_NULL_MD5
0x02	SSL_RSA_WITH_NULL_SHA
0x03	SSL_RSA_EXPORT_WITH_RC4_40_MD5
0x04	SSL_RSA_WITH_RC4_128_MD5

0x05	SSL_RSA_WITH_RC4_128_SHA
0x06	SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
0x07	SSL_RSA_WITH_IDEA_CBC_SHA
0x08	SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
0x09	SSL_RSA_WITH_DES_CBC_SHA
0x0A	SSL_RSA_WITH_3DES_EDE_CBC_SHA
0x0B	SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
0x0C	SSL_DH_DSS_WITH_DES_CBC_SHA
0x0D	SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA
0x0E	SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
0x0F	SSL_DH_RSA_WITH_DES_CBC_SHA
0x10	SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA
0x11	SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
0x12	SSL_DHE_DSS_WITH_DES_CBC_SHA
0x13	SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
0x14	SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
0x15	SSL_DHE_RSA_WITH_DES_CBC_SHA
0x16	SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
0x17	SSL_DHE_anon_EXPORT_WITH_RC2_CBC_40_MD5
0x18	SSL_DHE_anon_WITH_RC4_128_MD5
0x19	SSL_DHE_anon_EXPORT_WITH_DES40_CBC_SHA
0x1A	SSL_DHE_anon_WITH_DES_CBC_SHA
0x1B	SSL_DHE_anon_WITH_3DES_EDE_CBC_SHA
0x1C	SSL_FORTEZZA_DMS_WITH_NULL_SHA
0x1D	SSL_FORTEZZA_DMS_WITH_FORTEZZA_CBC_SHA
0x1E	SSL_FORTEZZA_DMS_WITH_RC4_128_SHA

Cuadro A.1: *Cipher suites* en SSLv3

A.2. *Cipher suites* obligatorias por RFC para TLSv1.0

En esta sección se muestran, sin entrar en profundidad, los conjuntos de cifrado de obligado soporte y definidos en el RFC 2246 [15] para esta versión y su valor hexadecimal que se enviaría realmente en los mensajes de *Hello*. Esta lista se ha extendido con multitud de *cipher suites* que pueden soportar distintas implementaciones cuya interoperabilidad depdende de cada una de las mismas. Una diferencia respecto a SSLv3 es que FORTEZZA ya no está soportado en TLS.

***Cipher suites* en TLSv1.0**	
Valor	***Cipher suite***
0x00	TLS_NULL_WITH_NULL_NULL
0x01	TLS_RSA_WITH_NULL_MD5
0x02	TLS_RSA_WITH_NULL_SHA
0x03	TLS_RSA_EXPORT_WITH_RC4_40_MD5
0x04	TLS_RSA_WITH_RC4_128_MD5
0x05	TLS_RSA_WITH_RC4_128_SHA
0x06	TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
0x07	TLS_RSA_WITH_IDEA_CBC_SHA
0x08	TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
0x09	TLS_RSA_WITH_DES_CBC_SHA
0x0A	TLS_RSA_WITH_3DES_EDE_CBC_SHA
0x0B	TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
0x0C	TLS_DH_DSS_WITH_DES_CBC_SHA
0x0D	TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
0x0E	TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
0x0F	TLS_DH_RSA_WITH_DES_CBC_SHA
0x10	TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA

0x11	TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
0x12	TLS_DHE_DSS_WITH_DES_CBC_SHA
0x13	TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
0x14	TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
0x15	TLS_DHE_RSA_WITH_DES_CBC_SHA
0x16	TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
0x17	TLS_DH_anon_EXPORT_WITH_RC4_40_MD5
0x18	TLS_DH_anon_WITH_RC4_128_MD5
0x19	TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA
0x1A	TLS_DH_anon_WITH_DES_CBC_SHA
0x1B	TLS_DH_anon_WITH_3DES_EDE_CBC_SHA

Cuadro A.2: *Cipher suites* en TLSv1.0

Apéndice B

Glosario de acrónimos

TLS *Transport Layer Security*

SSL *Secure Sockets Layer*

RFC *Request for Comments*

HTTP *Hypertext Transfer Protocol*

HTTPS *Secure Hypertext Transfer Protocol*

ASCII *American Standard Code for Information Interchange*

HMAC *Hashed Message Authentication Code*

PSK *Pre Shared Key*

PRF *PseudoRandom Function*

ALPN *Application Layer Protocol Negotiation*

SNI *Server Name Indication*

CBC *Cipher Block Chaining*

ECB *Electronic Code Book*

RTT *Round Trip Time*

Apéndice C

Glosario de términos

***Benchmark*:** método de medición del rendimiento o calidad por comparación con un punto de referencia. Existen una serie de *benchmarks* creados por cada sector de la industria de la informática. Uno de los más famosos son los *benchmarks* SPEC [1].

***Padding Oracle*:** clase especial de ataque que se construye hacia el receptor de la comunicación en el cual el relleno de un bloque en cifrados simétricos en modo CBC se puede modificar. Esto es posible cuando no se autentica la parte de relleno en los bloques. El atacante no puede modificar directamente el relleno, ya que está cifrado, pero puede modificar el texto cifrado arbitrariamente donde cree que el relleno puede estar situado. El "oráculo" se dice que existe cuando el atacante al modificar el *padding* cifrado consigue que un bloque sea correcto después de ser descifrado. Prueba tras prueba el atacante conseguirá descifrar el bloque completo byte a byte.

***SSL Offloading*:** técnica por la cual un dispositivo realiza la negociación SSL/TLS y envía a un tercero la información sin cifrar. Utilizado por ejemplo, entre balanceadores y servidores web. El balanceador asume la negociación SSL/TLS con el cliente y después envía la información sin cifrar hacia los servidores.

***Proxy*:** es un recurso *hardware* o *software* destinado a actuar como intermediario en una comunicación telemática.

[1] SPEC - https://www.spec.org/benchmarks.html

Bibliografía

[1] A. Sundararajan, A. Chavan, D. Saleem, and A. Sarwat, "A survey of protocol-level challenges and solutions for distributed energy resource cyber-physical security," *Energies*, vol. 11, p. 2360, 09 2018.

[2] I. Ristic, *Bulletproof SSL and TLS*.
Feisty Duck, 2014.

[3] S. A. Thomas, *SSL & TLS Essentials, Securing the Web*.
Wiley, 2000.

[4] "MPTCP and TLS 1.3," 2020.
https://www.internetsociety.org/blog/2017/06/mptcp-and-tls-1-3-big-announcements-from-apple/.

[5] "An overview of TLS 1.3 and Q&A," 2020.
https://blog.cloudflare.com/tls-1-3-overview-and-q-and-a/.

[6] "Benefits and Adoption Rate of TLSv1.3," 2020.
https://www.sans.org/reading-room/whitepapers/vpns/benefits-adoption-rate-tls-13-39715.

[7] P. Jönsson and S. Ivenson, *F5 Networks TMOS Administration Study Guide*.
F5 Books, 2018.

[8] "How to enable HTTPS/SSL encryption to secure your Facebook account," 2011.
https://nakedsecurity.sophos.com/2011/01/28/how-to-enable-httpsssl-encryption-to-secure-your-facebook-account/
Online; accedido el 24 de Septiembre de 2020.

[9] "RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3," 2018.

https://tools.ietf.org/html/rfc8446.

[10] "IETF Mission Statement," 2020.
https://tools.ietf.org/html/rfc3935.

[11] "The Transport Layer Security (TLS) Protocol Version 1.3 draft-ietf-tls-tls13-00," 2014.
https://tools.ietf.org/html/draft-ietf-tls-tls13-00.

[12] K. Hafner and M. Lyon, *Where wizards stay up late (the origins of the Internet).*
TOUCHSTONE, 1996.

[13] "ISO/IEC 7498-1:1994(en) - Information technology — Open Systems Interconnection ," 1994.
https://www.iso.org/obp/ui/#iso:std:iso-iec:7498:-1:ed-1:v2:en.

[14] "RFC 6101: The Secure Sockets Layer (SSL) Protocol Version 3.0," 1996.
https://tools.ietf.org/html/rfc6101.

[15] "RFC 2246: The Transport Layer Security (TLS) Protocol Version 1.0," 1999.
https://tools.ietf.org/html/rfc2246.

[16] "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1," 2003.
https://tools.ietf.org/html/rfc3447.

[17] D. Wagner and B. Schneier, "Analysis of the ssl 3.0 protocol," in *Proceedings of the 2nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*, WOEC'96, (USA), p. 4, USENIX Association, 1996.

[18] R. Oppliger, *SSL and TLS: Theory and Practice, Second Edition.*
Artech House, 2016.

[19] "RFC 3546: Transport Layer Security (TLS) Extensions," 2003.
https://tools.ietf.org/html/rfc3546.

[20] "Transport Layer Security (TLS) Extensions," 2020.
http://www.iana.org/assignments/tls-extensiontype-values/.

[21] "RFC 5077: Transport Layer Security (TLS) Session Resumption without Server-Side State," 2008.

https://tools.ietf.org/html/rfc5077.

[22] "Requisitos de Seguridad para TLS," 2020.
https://www.ccn.cni.es/index.php/es/docman/documentos-publicos/boletines-pytec/378-pildorapytec-nov2020-seguridad-tls/file.

[23] "RFC 4346: The Transport Layer Security (TLS) Protocol Version 1.1," 2006.
https://tools.ietf.org/html/rfc4346.

[24] "RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2," 2008.
https://tools.ietf.org/html/rfc5246.

[25] I. Grigorik, *High Performance Browser Networking*.
Shroff Publishers & Distr, 2014.

[26] "RFC 6066: Transport Layer Security (TLS) Extensions: Extension Definitions," 2011.
https://tools.ietf.org/html/rfc6066.

[27] "Introducing 0-RTT," 2020.
https://blog.cloudflare.com/introducing-0-rtt.