Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

DISEÑO E IMPLEMENTACIÓN DE UN PROXY SEGURO BASADO EN EL PROTOCOLO MULTI-CONTEXT TLS

TRABAJO FIN DE MÁSTER

Álvaro Andrés Anaya Amariles

2019

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

Máster Universitario en
Ingeniería de Redes y Servicios Telemáticos

TRABAJO FIN DE MÁSTER

DISEÑO E IMPLEMENTACIÓN DE UN PROXY SEGURO BASADO EN EL PROTOCOLO MULTI-CONTEXT TLS

Autor
Álvaro Andrés Anaya Amariles

Director
Borja Bordel Sánchez

Departamento de Ingeniería de Sistemas Telemáticos

2019

Resumen

Desde hace varios años se viene implementando seguridad en las comunicaciones para asegurarle al usuario un porcentaje alto de anonimato e integridad de los datos personales y en ocasiones sensibles o privados, las formas de aplicar seguridad e encriptación a la información se convierte en una tarea indispensable para los desarrolladores de aplicaciones que ofrecen sus productos al mercado, es por eso que la tarea de implementar encriptación de los datos empieza desde que se envían los datos por la red hasta que lleguen a su destino sin ser alterados o suplantados.
Uno de los métodos más conocidos en la actualidad es HTTPS, protocolo que maneja una capa adicional de encriptación que mantiene los datos en un alto porcentaje de fiabilidad, por debajo de HTTPS se encentran varias versiones de protocolos (más conocido TLS) que permite realizar esta labor, sin embargo al momento de optimizar los recursos de la red en los proveedores de contenidos/servicios esta tarea obstaculiza esta labor, debido a que estos proveedores no pueden identificar el contenido que se está enviado por a través de las redes, esto hace imposible proteger de amenazas a los servidores que ofrecen contenido, ya que las amenazas también pueden transportarse en contenido encriptado, si aquellas amenazas llegan a tener éxito de ataque, podrían afectar a millones de usuarios.

Es por eso que en este estudio se ha propuesto analizar y estudiar el protocolo de encriptación basado en TLS (McTLS) e implementarlo en middlebox ofreciendo la posibilidad al usuario de dictaminar si desea que sus datos puedan ser analizados para ofrecer un mejor rendimiento en la redes, es decir, un "todo o nada" en leer o escribir en la información enviada por la red para ofrecer seguridad tanto en el cliente como el servidor, esto mejoraría las comunicaciones y la seguridad en los dos extremos (cliente y servidor)

También se pudo evaluar McTLS como implementación para entornos reales, es decir, acoplar e integrar en un mismo entorno McTLS con TLS, iniciar una comunicación en McTLS y terminarla en el otro extremo (servidor) con TLS

Con esto es posible implementar muchas funcionalidades de seguridad como DPI, control parental, cachering, IDS, etc. Que ayudarán a tener una comunicación en las redes mucho más segura y fiable. Para entender este trabajo fue necesario estudiar la forma de funcionamiento de McTLS e intentar acoplarlo para que fuese capaz de comunicarse con extremos que esperaban y hablaran comunicaciones mediante TLS.

Abstract

From several years old the communications is being implanted to improve the level security in the network and protect the user in the high percentage of the anonymity and integrity of the personal data and in several moments the data can be sensitive or private, the method to apply security and encryption of the information is becomes in the priority task for the development of the applications that offer them products to the users, is for it, that the task of implement encryption of the data start when the data is sending over network until that the packets is received in the other side (server) without tampering or impersonation.
One of the method knew in the actuality is HTTPS, this protocol have an additional layer of encryption that stay the data in percentage high of reliability, by down of HTTS there are several versions of protocols (TLS) that make encryption, however to optimize resources of the network in the content/service providers this tasks make the labor more difficult to protect of thereat to the servers that offer the content to the users, because the threat can be encrypted over network where the servers can be affect in case that the threat can be execute the goal in the server and could affect to millions of users.

The proposal in this research was analyze the protocol of encryption based in TLS (McTLS) and implement this one in middlebox technology and that the user take actions about permit read or write to the service provides to enforcing the network and communications of end to end.

In this investigation was possible implement a demo in a real environment where was possible to integrate McTLS wit TLS and start a communication in McTLS and the other side can close the same communication in TLS.

With McTLS is possible implement many functionalities of security as DPI, parental filtering, cachering, IDS, etc. those functionalities will help to have a communication of network more safe and reliable. To understand this research was necessary study and analyze the operation of McTLS and try to integrate with TLS.

Índice general

Índice de figuras

Siglas

HTTPS	Hypertext Transfer Protocol Secure
HTTP	Hypertext Transfer Protocol
ITSA	Internet Traffic Statistics Archive
TLSv2/SSL	Transport Layer Security version 2/Secure Sockets Layer
TLSv1.1	Transport Layer Security version 1.1
MitM	Man in the Middle
CERT	Computer Emergency Response Team
CIA	Confidentiality, Integrity and Availability
IDS	Intrusion detection System
IPS	Intrusion Prevention System
NFV	Network Functions Virtualization
VNF	Virtual Network Function
VM	Virtual Machine
DPI	Deep packet inspection
McTLS	Multi-Context Transport Layer Security
IETF	Internet Engineering Task Force
AES	Advanced Encryptation Standart
RC4	Rivest Cipher 4
SHA-1	Secure Hash Algorithm 1
SHA-5	Secure Hash Algorithm 5
MD5	Message-digest algorithm
RSA	Rivest–Shamir–Adleman

DSA	Digital Signature Algorithm
CPU	Central Processing Unit
DES	Data Encryption Standard
ID	Identification
SMTP	Simple Mail Transfer Protocol
FTP	File Transfer Protocol
IP	Internet Protocol
MACs	Messages Authentication Codes
HMAC	Hash-based message authentication code
GMAC	Galois Message Authentication Code
OMAC	One-key MAC
GPU	Graphics processing unit
CRLs	Certificate Revocation Lists
CA	Certification Authority
PRNG	Pseudorandom number generator
CBC	Cipher Block Chaining
ECB	Electronic Codebook
OFB	Output Feedback
CAST5	Carlisle Adams/Stafford Tavares
IDEA	International Data Encryption Algorithm
PGP	Pretty Good Privacy
DH	Pretty Good Privacy
NAT	Pretty Good Privacy
NFs	Network functions
NFVI	Network functions virtualization infrastructure

EM	Element Management
OSS/BSS	Operation and Business Support Systems
EPC	Envoled Packet Core
SGW	Serving Gateway
MME	Mobility Management Entity
PGW	Packet Data Network Gateway
RGW	Gateways residenciales
DHCP	Dynamic Host Configuration Protocol
RAM	Random-access memory
DoS	Denied of service
VN	Virtual Network
InPs	Proveedor de infraestructura
SPs	Proveedor de servicios
ALG	Application Level Gateway
PEP	Performance Enhancing Proxy
ACK	Acknowledge
MTA	Mail Transfer Agents
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
HTML	Hypertext Markup Language
ICP	Internet Cache Protocol
HTCP	Hyper Text Caching Protocol
PNG	Portable Network Graphics
MAC	Media Access Control
RTT	Real Time Traffic

IDSec	Sistemas de detección de intrusos
LAN	Local Access Network

1 Introducción

El uso continuo de internet es indispensable en cualquier tarea común que se desee practicar entre las personas, por tal motivo, se ha introducido como necesidad que facilita labores cotidianas en la vida personal o laboral, a este incremento de necesidad se añade la importancia seguridad al momento que se adentra en el mundo de internet, por tal motivo, en los últimos años, la privacidad en internet se ha vuelto trascendental para cualquier labor, es por eso que las principales empresas dominantes en el mundo de las telecomunicaciones cubren sus datos bajo ciertas restricciones de seguridad, Facebook, Google, Amazon y Twitter es un ejemplo de ello donde solventan esta necesidad encriptando la información bajo HTTPS, a medida que pasa el tiempo, el tráfico encriptado cubre cierta parte del tráfico de internet [1], a partir del 2014 a la actualidad, ITSA[1] reporta estadísticas de alto nivel del tráfico que viaja a través de internet, estadísticas que están basadas por puerto, protocolos y distribución de paquetes, resaltando que a medida que pasan los años, las conexiones HTTPS irán aumentando cada vez más.

TLSv2/SSL es el estándar escogido y el más usado en la actualidad para añadir seguridad a través de internet, el cual consiste en agregar una capa de seguridad al tráfico HTTP y así obtener confidencialidad, integridad y encriptación extremo a extremo de los datos, esto conlleva que en la actualidad, los navegadores soporten una mezcla de HTTP y HTTPS, situación que no genera mayores inconvenientes, pero sí vulnerable a "stripping attacks" [2] que por defecto el atacante puede intervenir en la redirección HTTP a HTTPS. Así como "stripping attacks", también está "man in the middle" (MitM) que activa un proxy malicioso donde en dado caso que logre introducirse en la red, puede interceptar, modificar, bloquear o re direccionar todo el tráfico.

Con la acentuación de sesiones de internet y a los sistemas de computación, ha surgido numerosas cuestiones de seguridad dado el explosivo uso de las redes, estadísticas hechas por CERT [3] reporta que la intrusión a los sistemas de información y redes incrementan excesivamente año a año afectando sustancialmente las políticas de seguridad como CIA (Confidentiality, Integrity and Availability). Además, incluyendo la diversidad de contenido que viaja por internet y la necesidad de optimizar el contenido entre otros servicios [2][3][4] Existen servicios como

[1] http://stats.simpleweb.org/statistics.php?l=2&w=6&y=2017
[2] Palo Alto Networks. https://www.paloaltonetworks.com
[3] http://www.qosmos.com/products/deep-packet-inspection-engine/

detección/prevención de intrusos (IDS/IPS), que detectan si existen paquetes maliciosos en la red que puedan comprometer los servicios de seguridad, a su vez, el servicio "cachering" usado para mejorar las prestaciones de la red duplicando el contenido de descarga a contenidos más solicitados o "parental filtering" para prohibir el acceso a niños a contenido adulto en escuelas, bibliotecas o en hogares [5]. Estos servicios de red llevan a colación a middlebox, los cuales ofrecen muchos beneficios a usuarios, proveedores de contenidos y operadores de red. El gran inconveniente al que se enfrentan los middlebox son los datos que viajan con encriptación (ej. HTTPS) y debido a esto los middlebox no pueden realizar las tareas encomendadas.

El punto donde actualmente el mercado de las telecomunicaciones tiende a concentrarse es en mejorar la flexibilidad en los servicios ofrecidos por internet, por tal motivo NFV "Network Functions Virtualization" [5] ha surgido como una iniciativa promovida por el operador centrada al objetivo de mejorar la flexibilidad en los servicios de red y tener despliegue e integración dentro de los operadores de red, esto conlleva implementar funciones basadas en software en lugar de aplicar el concepto de soluciones hardware, por lo tanto, siguiendo esta tendencia, es posible reemplazar funciones de red implementadas en hardware con middlebox virtualizadas (VNF) establecidas por software que consiste en una porción de VM's (Virtual Machine) corriendo en diferentes procesos donde su finalidad sea realizar alguna función de red que sea instanciado a través de NFV, por lo tanto, esto permite acotar recursos económicos, reducir la dificultad de mantenimiento y actualización de dichas funciones.

Recientemente se ha presentado una solución de un protocolo [4] construido bajo TLSv2 que combina seguridad y funcionalidades de red para visualización y control extremo a extremo de los datos, con esto, los extremos de la sesión tienen control y conocimiento sobre qué elementos hacen parte de la sesión, además, permite escoger a los usuarios y proveedores de contenidos el tipo de datos visualizados en la sesión, asimismo, ofrece autenticación en integridad de los datos mientras se mantiene la sesión y en paralelo los permisos de lectura y escritura se distribuyen de manera independiente, por lo tanto, este protocolo fue construido como una extensión de OpenSSL [6] y establece su amplio despliegue.

---

[4] Radisys R220 Network Appliance. http://www.radisys.com/press-releases/radisys-new-carrier-grade-network-appliance-platform-delivers-highly-flexible

[5] University of Toulouse Internet Blacklists http://dsi.ut-capitole.fr/blacklists/

[6] OpenSSL C ryptography and SSL/TLS Toolkit https://www.openssl.org/

Utilizando [4] como extensión de TLSv2 y a fin de permitir soporte de middlebox para ofrecer un modelo de seguridad que permita a puntos finales de la sesión y a los proveedores de servicios incluir middlebox seguros, se establece como solución aplicar este método que controle y permitan el manejo de lectura y escritura del contenido transmitido en la sesión sin necesidad de aplicar funciones donde el cliente vea comprometida la seguridad en su información.

1.1 Motivación

A medida que avanza el tiempo, es cada vez más común encontrar que los contenidos transportados por la red adjunten cabeceras de seguridad debido a la cantidad de hechos relacionados con suplantación, robo y espionaje de información que cada día hace que las aplicaciones prevengan (a favor del cliente y servidor) comportamientos inusuales donde se pueda ver comprometida la seguridad en datos y evitar presencia indeseada en la sesión, es por eso que, una significativa parte en internet optó por usar encriptación de los datos para proveer confidencialidad y asegurar las comunicaciones.

Dada la nubosidad que genera tener las comunicaciones encriptadas, es complicado acceder a visualizar el contenido de datos, mitigando el valor añadido que tiene los servicios de red como "caching" o "scanners" de virus sobre la capa de aplicación, volviéndose ineficiente tener estos servicios cuando se trata de comunicaciones encriptadas. Muchos middlebox de red realizan "Deep packet inspection" (DPI), efectuando un conjunto de tareas de análisis de "payloads" en los datos, tareas que incluyen IDS y control parental. Sin embargo, hace varios años, estos paquetes son enviados bajo HTTPS y los middlebox no pueden realizar el análisis de datos porque estos datos están encriptados, por lo que la propiedad de tener privacidad y control se reduce solo en encriptación.

Generado este debate que nace a partir de tener privacidad o control. En este proyecto, aplicando un nuevo protocolo de seguridad que surge como extensión a TLSv2, se pretende analizar el comportamiento con los datos y la comunicación y así realizar un middlebox que permita abrir la brecha donde la privacidad y el control por parte de los servicios de red son por ahora esquivos a asociarse.

1.2 Objetivos

Como objetivo general se plantea ***implementar una VNF que soporte tráfico encriptado sobre McTLS y TLS,*** donde sea posible encontrar una solución que incorpore McTLS y TLS en la sesión, y que a su vez los datos añadan cabeceras encriptadas haciendo los datos privados pero que en paralelo pueda acceder al control de los mismos.

Para alcanzar el objetivo en cuestión, se necesita analizar de manera independiente conceptos y aspectos específicos como:

- Investigar y analizar el funcionamiento implementado por multi-context TLS y TLSv2, además, sus requisitos, procedimiento, manejo de los datos, métodos de seguridad y despliegue en la red.
- Determinar soluciones proxy para llevar McTLS a entornos reales.
- Realizar un análisis de entornos o escenarios reales donde McTLS optimice el uso de servicios en la red en favor de los puntos finales de la sesión.

1.3 Estructura del documento

El presente documento ofrece una descripción y análisis de un nuevo protocolo de seguridad llamado Multi-Context Transport Layer Security (McTLS) como alternativa a TLS para transportar los datos de forma segura por la red e implementarlo sobre una VNF o máquina virtual. El eje central de este documento se centra en McTLS y los métodos correctos para traducir los datos de un protocolo de encriptación a otro (McTLS a TLS), por lo cual se ha seleccionado varios capítulos que se describen a continuación:

- ***Capítulo 1 –*** *Introducción:* Se describe en un contexto los temas críticos que afectan actualmente internet como la seguridad y rendimiento que involucran a usuarios, empresas y proveedores de servicios. También resaltada estadísticas que revelan la necesidad de establecer una fórmula para mitigar o disminuir los problemas que día a día internet está teniendo debido a los ataques que se generan a través del mundo y la propuesta de un nuevo estándar que mejore la seguridad en las redes de internet.
- ***Capítulo 2 -*** *tecnologías de interés*: Proporciona información teórica acerca de conceptos útiles que son necesarios analizar para implementar el desarrollo de este proyecto, en este capítulo se nombra tecnologías como Transport Layer Security (TLS), Hypertext Transfer Protocol (HTTP), Hypertext Transfer Protocol Secure (HTTPS), Network Function Virtualization,

Middlebox y el protocolo de seguridad Multi-Context Transport Layer Security.

- ***Capítulo 3 –*** *Desarrollo:* Aquí se analizan los requisitos necesarios de McTLS para luego valorar ideas que sean útiles y acertadas al desarrollo del proyecto, a su vez se mostrarán ideas que no tuvieron éxito y la conclusión a la que se llegó para obtener la idea más acertada, también se analizan de forma individual los componentes que hacen parte de una sesión que usa McTLS y según los datos obtenidos de esa investigación se agrupa lo investigado para finalmente tener un planteamiento y así integrar McTLS y TLS en una misma sesión o comunicación.
- ***Capítulo 4 -*** *Caso demostrativo y resultado:* Inicialmente describe la arquitectura que se propuso para realizar los test de funcionalidad de lo que se había desarrollado, también se describe los componentes que hacen parte del entorno de pruebas y el role que adquieren dentro de la arquitectura, posteriormente se muestran resultados de los valores obtenidos en las simulaciones, realizando varios pruebas de funcionalidad y describiendo cada uno de los resultados obtenidos.
- ***Capítulo 5 -*** *Conclusiones y líneas futuras de investigación*: Da a conocer las reflexiones que se han obtenido durante el desarrollo de este proyecto, además, ideas donde este proyecto sea el punto de partida para mejoras, nuevas versiones o implementaciones de nuevas características que puedan complementar la integración entre McTLS y TLS.
- ***Bibliografía***.

2 Tecnologías de interés

En esta sección se desea analizar tecnologías como Transport Layer Security (TLS) como protocolo de encriptación de datos, conceptos de seguridad relacionados con criptografía, confidencialidad, integridad y autenticación de los datos, tipos y métodos de ataques más conocidos, conceptos de algoritmos criptográficos, tecnologías como Hypertext Transfer Protocol (HTTP), análisis de OpenSSL, tipos de certificados criptográficos, NFV como arquitecturas de nueva generación y middlebox como conjunto de herramientas que procesan datos y realizan acciones. Todas estas tecnologías y sistemas ayudarán como partida inicial al desarrollo de este proyecto.

2.1 Transport Layer Security

TLS (*Transport Layer Security*) es el protocolo que proporciona comunicación segura sobre internet, el cual permite comunicación entre las capas de aplicación de cliente y servidor, este protocolo está diseñado para prevenir entre otras cosas, falsificación, manipulación o espionaje de los datos.

Existe diferencias entre TLSv1.1 y TLSv2 las cuales están descritas en [8], sin embargo las más relevantes son: inserta *cipher-suite-specified*, el elemento *digitally-signed* es reemplazado por un solo Hash, lo que incluye un campo que especifica el tipo de algoritmo Hash usado.

Por defecto al usar TLS como protocolo de seguridad provee privacidad e integridad de los datos entre los extremos de la sesión, TLS es un protocolo que cualquier desarrollador puede utilizar para encriptar sus aplicaciones, TLSv2 usa RFC 5246 [7] propuesto por IETF [8] (*Internet Engineering Task Force*) como estándar de encriptación que se compone de dos capas: (i) *Protocol Record TLS* (ii) *TLS Handshake protocol*. Una de las ventajas de TLSv2 lo lleva a ser un protocolo de aplicación independiente, por lo que protocolos de capas superiores pueden usarlo de manera transparente.

TLS es el protocolo más utilizado para ofrecer capa de seguridad. Este es el protocolo de seguridad dentro del HTTPS, es el responsable de la capa de seguridad en cualquier aplicación bajo TCP.

La comunicación lógica inicia como lo muestra la siguiente figura, con el cliente enviando un handshake al servidor, el servidor responde y envía un certificado que previamente fue mencionado. **Un certificado es una pieza de datos que incluye una**

[7] https://tools.ietf.org/html/rfc5246#section-1
[8] http://www.ietf.org/

llave pública asociada con el servidor y otra información relevante como el certificado, tiempo de validez, etc.

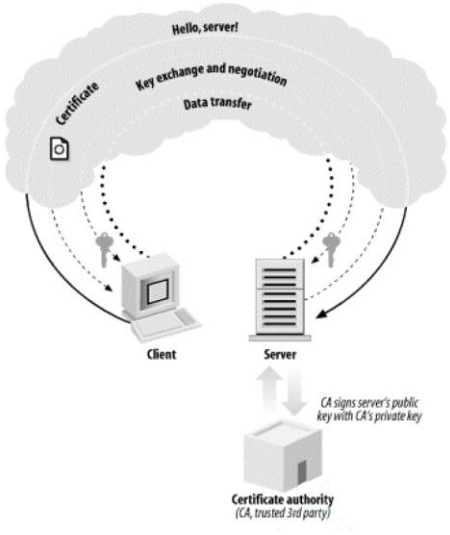

Figura 1. Handshake TLS.

Durante el proceso de conexión, el servidor proveerá identidad usando la llave privada para des-encriptar los datos que el cliente encripta con el servidor, el cliente necesita recibir los datos sin cifrar para proceder. Sin embargo, el servidor puede permanecer público. Así, un atacante necesitaría una copia del certificado y la llave privada. Sin embargo, un atacante podría interceptar mensajes servidor y presentar un certificado propio. Los datos en campos que pueden ser visibles en un certificado son de tipo: nombre de dominio asociado con el servidor y el nombre de la entidad certificadora. Teniendo en cuenta estos datos, el atacante podría establecer una conexión a través de un proxy al servidor de destino y posteriormente ponerse en escucha de lo que pasa entre el cliente y servidor haciendo una especie de man-in-the-middle (como lo muestra la siguiente figura). Es por eso que el cliente no solo debe realizar validación completa del certificado del servidor, sino que también debe asegurar que ese certificado es confiable.

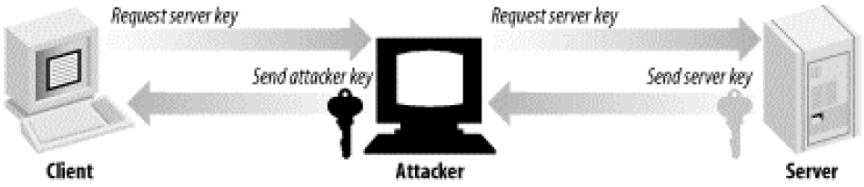

Figura 2 Man-in-the-middle.

La solución a este problema es involucrar un ente externo (*third party*) llamado **autoridad certificadora** que valide el certificado servidor usando una llave privada. El cliente puede validar la firma de la autoridad certificadora (asumiendo que este tiene la llave pública de la entidad certificadora). Si el testeo es exitoso, el cliente puede asegurarse que el certificado es proporcionado por una entidad certificadora confiable. Una vez validado del certificado, el cliente y servidor añaden seguridad usando algoritmos simétricos, una vez se haya completado la negociación, el cliente y servidor pueden compartir datos. En detalle, TLSv2 es un ligeramente más complejo, ya que los mensajes de autenticación son usados extensamente para asegurar la integridad de los datos. Además, durante la validación del certificado, una parte puede ir a la entidad certificadora donde "*Certificate Revocation Lists*" asegura que el certificado actual no ha sido robado o suplantado.

TLSv2 es un protocolo de altas prestaciones de seguridad, pero como en muchas aplicaciones, la eficiencia va ligada a la forma de uso que haya sido dada. Por lo que hay muchas aplicaciones que tiene fallos de seguridad por culpa de los desarrolladores que crean sus aplicaciones y el mal uso de TLSv2.

Una conexión a través de TLS es mucho más lenta que una conexión común en TCP/IP. Este es un problema con el que TLS debe lidiar. Cuando una sesión es establecida, el cliente y servidor intercambian una cantidad considerable de información que es requerida en la sesión. Este *handshake* inicial involucra criptografía fuerte en las llaves públicas, un ejemplo visto son las conexiones HTTPS que se ven actualmente a través de internet.

Protocol Record TLS tiene la función de encapsular protocolos superiores, uno de estos protocoles es *TLS Handshake Protocol*, además, permite autenticación entre el cliente y el servidor e inicia la negociación encriptada a través de las claves criptográficas antes de iniciar la transmisión.

Al ser un protocolo en capas, cada capa puede incluir un campo que almacene la longitud, descripción y contenido. El protocolo recoge los mensajes y los transmite, fragmenta los datos en bloques manejables, de manera opcional comprime los datos, aplica una MAC, encripta y transmite el resultado. Al otro extremo, los datos son recibidos, des-encriptados y verificados, en dado caso que hayan sido comprimidos, lo descomprime, re-ensambla y entrega los datos a las capas superiores, por lo tanto, *Protocol Record TLS* se compone de dos propiedades básicas:

- La conexión es privada, esta propiedad hace uso de criptográfica simétrica (AES [6], RC4 [7], etc.) para encriptación. Las llaves criptográficas son generadas exclusivamente por cada conexión y es basada por una negociación secreta

hecha por otro protocolo (ej. *TLS Handshake Protocol*), sin embargo, *Record Protocol* acepta conexiones sin encriptar.

- La conexión es de confianza, el envío de los datos incluyen un mensaje para validar la integridad, lo hace a través de una clave MAC, estas claves son generadas a través de funciones Hash (ej. SHA-1, SHA-5, MD5, etc.). *Record Protocol* puede trabajar sin esta función Hash, sin embargo, este modo es usado normalmente cuando otro protocolo esté usando *Record Protocol* como transporte para completar los parámetros de negociación.

Los parámetros criptográficos de una sesión son producidos por *TLS Handshake protocol*, el cual opera encima de la capa de TLS record, cuando un cliente o un servidor inicia la comunicación, agregan la versión del protocolo usado, selecciona los algoritmos criptográficos, opcionalmente se autentican uno con otro y usan la técnica de encriptación de llave privada para generar el intercambio secreto de datos, este protocolo involucra:

- Intercambio de mensajes *hello* de acuerdo a los algoritmos usados, intercambio de valores randómicos y revisión de la reanudación de la sesión.
- Intercambio de parámetros criptográficos necesarios para permitir al cliente y servidor acordar un *pre master* secreto.
- Intercambiar certificados e información criptográfica que permita al cliente y servidor autenticarse por sí mismos.
- Generar un *master* secreto a partir del *pre master* intercambiar los valores randómicos.
- Proveer parámetros de seguridad en la capa TLS Record.
- Permitir al cliente y servidor verificar que el otro extremo calculó los mismos parámetros de seguridad y que el *handshake* fue exitoso sin presencia de ataques que hayan manipulado la sesión.

TLS Handshake Protocol provee conexión segura que posee tres propiedades básicas:

1 La identidad de los participantes puede ser autenticada a través de simetría o a través de una llave pública (ej. RSA, DSA, etc.), esta condición permite el estado de opcional, sin embargo, lo recomendable es que al menos uno de los participantes la use.

2 La negociación compartida es secreta, esta negociación es secreta por lo que mitiga algún ataque en medio de la sesión.

3 La negociación es de confianza: ningún ataque puede modificar la negociación sin ser detectado por los extremos de la sesión.

Los objetivos principales de implementación TLS hacen referencia a que tenga seguridad criptográfica donde asegure establecer una conexión de confianza entre ambas partes (extremos) de la sesión, además, requiere interoperabilidad, donde independiente del programa, el protocolo debe ser capaz de desplegar aplicaciones que permitan intercambio de mensajes con parámetros encriptados sin conocer el tipo de lenguaje del otro extremo de la sesión, asimismo, describe extensibilidad, donde TLS busca proveer un marco de referencia el cual nuevas llaves públicas y métodos de incorporación masivos puedan ser incluidos cada vez que sea necesario, igualmente, general eficiencia, debido a que las operaciones criptográficas tienden a requerir recursos CPU (ej. Operaciones de llave publica), TLS incorpora una opción de ***caching*** **como esquema que reduce el número de conexiones necesarias para establecer la sesión**, por lo tanto, tiene bastante cuidado con reducir la actividad de red.

TLS ha sido diseñado para minimizar riesgos de ataques, sin embargo, no los mitiga completamente y no está exento de otra clase de ataques. Por nombrar algunos ejemplos: no tiene control de un ataque que bloquee el acceso al puerto que por defecto usa TLS o intentar obtener una negociación de manera desautorizada entre los extremos de la sesión, sin embargo, **TLS es seguro cuando ofrece *cipher suite***; si la negociación se hace a través de 3DES [9] con un intercambio de llaves RSA 1024 bits con un host cuyo certificado haya sido verificado, entonces, la sesión será segura.

Este objetivo es logrado gracias al protocolo de *Handshake*, el cual puede ser resumido de la siguiente manera: El cliente (extremo de la sesión) envía un mensaje de tipo *ClientHello,* donde el servidor (el otro extremo de la sesión) debe responder con un mensaje de tipo *ServerHello* o un mensaje de error *fatal error* en caso que la comunicación falle. Los mensajes de *ClientHello* y *ServerHello* son útiles para generar seguridad más robusta en la sesión, los mensajes establecen los siguientes atributos como, versión de protocolo, ID de sesión, *Cipher Suite* y método de compresión, adicionalmente, se generar y se intercambian dos valores randómicos como *ClientHello.random* y *ServerHello.random*. El actual intercambio de llaves utiliza hasta cuatro mensajes: (i) El certificado servidor, (ii) *ServerKeyExchange*, (iii) Certificado cliente y (iv) *ClientKeyExchange*. El método de intercambio de llaves puede ser creado especificando un formato para estos mensajes y definiendo el uso de los mismos para permitir al cliente y servidor compartir la llave secreta, esta llave secreta debe ser extensa, a partir de 46 bytes en adelante.

Cuando los mensajes *hello* hayan sido enviados y dado el caso que la autenticación haya sido exitosa, el servidor enviará el certificado en un mensaje *Certificate*,

[9] Singh, G. (2013). A study of encryption algorithms (RSA, DES, 3DES and AES) for information security. International Journal of Computer Applications, 67(19) doi:http://dx.doi.org/10.5120/11507-7224

adicionalmente, un mensaje de *ServerKeyExchange* puede ser enviado, este mensaje es usado en el caso de que el servidor no tenga certificado, o si este certificado es solo para firma. Si la autenticación del servidor es correcta, entonces el servidor solicitará el certificado del cliente, hecho esto, el servidor enviará un mensaje *ServerHelloDone*, indicando que la fase del *handshake* ha sido completada y el servidor estará a la espera de la respuesta del cliente. Si el servidor envía un mensaje de tipo *CertificateRequest*, el cliente estará obligado a enviar un mensaje de tipo *Certificate*. Posteriormente el cliente enviará un mensaje *ClientKeyExchange* y el contenido del mensaje dependerá del algoritmo seleccionado por el *ClientHello* y el *ServerHello* para la llave privada; en caso de que el cliente envíe un certificado con la capacidad de firma, un mensaje de *digitally-signed CertificateVerify* es enviado para verificar explícitamente la posesión de la llave privada en el certificado.

A partir de ahora, un mensaje de *ChangeCipherSpec* es enviado por parte del cliente, a su vez, el cliente copia el *Cipher Spec* pendiente dentro del *Cipher Spec* actual, luego el cliente inmediatamente enviará un mensaje de tipo *Finished* bajo un nuevo algoritmo. En respuesta a esto, el servidor enviará un mensaje de tipo *ChangeCipherSpec*, transfiere el *Cipher Spec* actual y envía un mensaje de *Finished* bajo un nuevo *Cipher Spec*. A partir de ahora el *Handshake* está completado, por lo que el cliente y servidor pueden empezar a intercambiar datos a la capa de aplicación, por lo tanto, en la siguiente figura se muestra de manera gráfica el intercambio de mensajes de un *Handshake TLS*.

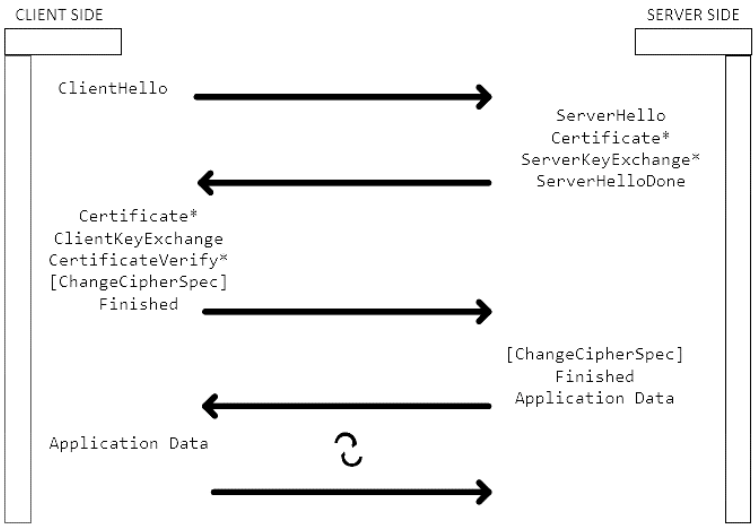

Figura 3. Flujo de mensajes de un Handshake.

En la figura anterior es preciso advertir que los mensajes señalados con (*) son mensajes condicionados que no siempre hacen parte del *Handshake*, es decir, cuando el cliente y el servidor deciden acotar el tiempo de *Handshake* el proceso de intercambio de mensajes se resume.

El cliente envía un *ClientHello* usando un ID de sesión que indica la sesión resumida, luego el servidor verifica esta sesión en cache y encuentra si existe una relación. En caso de que exista, el servidor re-establecerá la conexión bajo un estado de sesión específico, posteriormente enviará un *ServerHello* con el mismo ID de sesión, en este punto, el cliente y servidor deberán enviar mensajes etiquetados de tipo *ChangeCipherSpec* y proceder directamente a mensajes de tipo *Finished*. Una vez re-establecida la sesión, el cliente es aceptado para intercambiar datos a la capa de aplicación, sino existe relación entre el ID de sesión, el servidor generará un nuevo ID de sesión y a partir de ahí, el cliente y servidor realizarán el proceso completo de *Handshake*, por lo tanto, la siguiente figura expone el proceso resumido del *Handshake*.

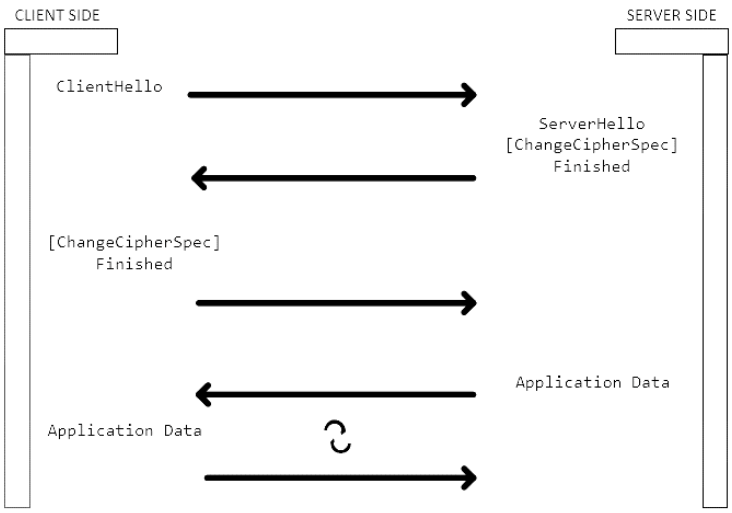

Figura 4. Flujo de mensajes de un Handshake abreviado

2.1.1 Llaves en claro

En una instalación típica de TLS, el servidor mantiene las credenciales resultando que los clientes puedan autenticarse al servidor. Adicionalmente, para que un certificado esté presente en una conexión real, el servidor debe mantener una llave privada la cual es necesaria para correcta comunicación. La llave necesita estar almacenada en algún sitio dentro del servido. La solución más segura es usar aceleración de criptografía hardware. Muchos de estos dispositivos pueden generar y almacenar material clave, a su vez, prevenir que la llave privada sea accesible por algún atacante que quiere hacerse con ella en la máquina.

2.1.2 Validación del certificado

CRLs (*Certificate Revocation Lists*) no es lo más útil para que los clientes realicen una validación adecuada al servidor, ciertamente, para que TLS trabaje bien, el cliente debe ser capaz de extraer la llave publica a partir del certificado y el servidor debe tener una llave privada que corresponda con esta llave publica, sin embargo, no hay mecanismos para realizar una validación adicional, por lo que *ataques man-in-the-middle* son factibles de realizar.

Para eso, los desarrolladores deben decidir cuales autoridades certificadoras son confiables y verificar que esos certificados son asociados con dichas autoridades certificadoras.

Incluso algunas aplicaciones que instalan CA y las usan para validarse contra el servidor a menudo fallan en el chequeo adecuado del contenido del certificado. Por lo que dichos sistemas son susceptibles a ataques mencionados anteriormente, ofreciéndole al atacante obtener de primera mano credenciales del cliente y robar sus credenciales firmadas por una CA. Por lo tanto, la mejor solución a estos problemas es acordar el tipo de autenticación que necesita el cliente. Muchas aplicaciones solo esperar que estas se comuniquen de forma segura con alguno grupo de servidores, en estos cosas, lo recomendado es examinar los campos apropiados en el certificado donde se valide los nombres de esos servidores haciendo una especie de "*Whitelist*"[10]. Adicionalmente, si la finalidad es asegurar los mecanismos de autenticación establecidos por la CA, se puede considerar crear un CA (asumiendo el control total entre el cliente y el servidor).

2.1.3 Entropía algorítmica

En TLS tanto cliente como servidor necesitan generar datos randómicos para las llaves y otros parámetros. Los datos son generados de tal forma que un atacante no pueda saber acerca de esos datos, es por eso que TLS genera estos datos a través de un número pseudorandomico generado por (PRNG). PRNG es un algoritmo determinístico que produce una serie de número aleatorios. El clásico PRNG no es adecuado usarlo para entornos donde la seguridad es crítica, para eso, TLS implemente criptografía PRNGs, el cual es usado en situaciones de riesgo informático.

[10] http://www.esecurityplanet.com/malware/whitelisting-why-and-how-it-works.html

2.1.4 Mensajes Hello

El ciclo de mensajes *Hello* es usado para intercambiar información segura entre cliente y servidor, cuando inicia una sesión, las capas de encriptación, hash y algoritmos de compresión son inicializadas en estado nulo, por lo tanto, este estado de conexión es usado para la renegociación de mensajes. Existen varios tipos de mensajes con etiqueta con este tipo de etiqueta.

- *Hello Request:* Cuando este mensaje es enviado, esta etiqueta puede ser enviada al servidor en cualquier momento, *HelloRequest* es una notificación de que el cliente debe iniciar de nuevo el proceso de negociación. En respuesta, el cliente deberá enviar un *ClientHello*. Este mensaje no está destinado a establecer sesión, solo es usado para iniciar una nueva negociación. El servidor no debería enviar un *HelloRequest* inmediatamente después que el cliente inicia conexión. Si el servidor envía un *HelloRequest* y no recibe un *ClientHello*, este puede cerrar la conexión con un mensaje de error, después de haber enviado un *HelloRequest* el servidor no deberá repetir la petición hasta el que *handshake* haya sido completado.
- *Client Hello:* Cuando un cliente se conecta por primera vez al servidor, es necesario enviar en un inicio un mensaje *ClientHello*, además, el cliente también puede enviar un *ClientHello* en respuesta a un *HelloRequest*. TLS permite extensiones de compresión dentro de un bloque de extensiones, la presencia de extensiones pueden ser detectados determinando si hay bytes sobrantes en el final del mensaje *ClientHello*.
- *client_version*: Es la versión del protocolo TLS del cual el cliente desea comunicarse en la sesión, por recomendación debería ser la última versión del protocolo.
- *Random*: Es la estructura randómica generada por el cliente.
- *Session_id*: Es el ID de la sesión donde el cliente quiere establecer conexión, este campo es vacío si la *session_id* no está disponible, o si por otro lado, el cliente desea generar nuevos parámetros de seguridad.
- *Cipher_suites*: Es la lista de opciones criptográficas que soporta el cliente.
- *Compression_methods*: Es la lista de métodos de compresión soportados por el cliente.

2.1.5 Server Hello

El servidor enviará este mensaje en respuesta de un *ClientHello*, dado que el servidor no encuentre relación entre los mensajes, el servidor enviará en el *Handshake* un mensaje de alerta.

- *Server_version*: Este campo contiene la versión del protocolo especificada por el servidor.
- *Random*: esta estructura es generada por el servidor, debe ser generada de forma independiente a partir del *ClientHello.random*.
- *Session_id*: Es el identificador correspondiente a la conexión, si el *ClientHello.session_id* no está vacío, el servidor buscará en la cache de sesiones una relación entre una y otra. Este campo contiene un ID de sesión diferente para cada conexión.
- *Cipher_suite*: Solo una opción criptográfica es seleccionada por el servidor a partir de la lista en *ClientHello.compression_methods*.

2.1.6 Server Certificate

El servidor debe enviar un mensaje de *Certificate* cuando hayan acordado el método de intercambio de llaves usadas para autenticar los certificados, este mensaje será enviado inmediatamente después del *ServerHello*.

El mismo tipo y estructura de mensaje debe ser usado por el cliente, hay que aclarar que el cliente puede no enviar certificados sino posee los certificados adecuados en respuesta a la petición de autenticación.

2.1.7 Server Key Exchange Message

Este mensaje será enviado inmediatamente después de mensaje *Server certificate* (o del *ServerHello* en caso de tratarse de una negociación anónima). Este mensaje es enviado por el servidor solo cuando el *Server certificate* no contenga suficientes datos para permitir el intercambio de *pre master* con el cliente.

2.1.8 Certificate Request

Un servidor puede opcionalmente solicitar un certificado a partir del cliente si selecciona apropiadamente el *cipher suite*, si este mensaje es enviado entonces inmediatamente reenviará el *ServerKeyExchange.*

2.1.9 Server Hello Done

Este mensaje es enviado por el servidor para indicar el final de la asociación del *ServerHello*, después de enviado este mensaje, el servidor esperará por la respuesta del cliente.

Este mensaje significa que el servidor realizó el envío de mensajes y confirmar el intercambio de llaves, con esto, el cliente puede proceder con la fase del intercambio de llaves.

En cuanto se recibe un *ServerHelloDone*, el cliente debería verificar que el servidor posee un certificado válido y si es necesario, chequear que el los parámetros de *server hello* son correctos.

2.1.10 Client Certificate

Este es el primer mensaje que el cliente puede enviar después de recibir un *ServerHelloDone*. Este mensaje solo es enviado si el servidor solicita un certificado. En caso que el cliente no posea certificados entonces el cliente deberá enviar un mensaje resaltando que no tiene certificados para verificar la sesión, en este caso el servidor puede ya sea continuar con el *Handshake* sin autenticar el cliente, o puede responder con un mensaje de alerta. También, si por algún aspecto el certificado haya sido cambiado (ej. Dado que la firma del certificado no sea conocida, CA), el servidor puede continuar con la *Handshake* o enviar un mensaje de error (alerta).

Este mensaje significa que el cliente trasmite la cadena de certificados hacia el servidor; el servidor usará esto cuando se haya verificado el certificado enviando un mensaje *CertificateVerify*. Este mensaje debe ser apropiado para la negociación verificado a través del algoritmo usado para el intercambio de claves. El certificado debe ser firmado usando un hash o firma de algoritmo adecuada.

2.1.11 Client Key Exchange Message

Este mensaje siempre es enviado por el cliente, esto debe enviar inmediatamente el mensaje del certificado del cliente. Por otra parte, debe primero enviar este mensaje y después recibir un *ServerHelloDone*. Con este mensaje se ajusta el *pre-master* ya sea directamente, por transmisiones con RSA o por *Diffie-Hellman*.

2.1.12 Certificate Verify

Este mensaje es usado para proporcionar verificación de un certificado cliente. Cuando se envía este mensaje, inmediatamente debe enviar el mensaje de intercambio de claves.

2.1.13 Finished

Un mensaje de *finished* siempre es enviado después de una cadena de *chiper spec* para verificar que el intercambio de claves y los procesos de autentificación hayan sido exitosos.

El *finished* de un mensaje es protegido con algoritmos de negociación y deben verificar que su contenido sea correcto. Una vez el destino haya enviado y recibido un mensaje *finished* verificando que la validación haya sido correcta es posible empezar a intercambiar datos de a nivel de aplicación sobre la conexión. Por otra parte, existe un error si un *finished* no es procesado por un mensaje *ChangeCipherSpec* en el momento adecuado del *Handshake*.

Un *ChangeCipherSpec* es enviado por el cliente y servidor para notificar que los datos serán protegidos bajo una nueva negociación, este mensaje es enviado durante el *Handshake* después de los parámetros de seguridad hayan sido acordados.

2.2 Criptografía

El primer objetivo de la criptografía es confirmar los datos importantes que viajan a través de un medio inseguro puedan ser seguros. Existen diferentes algoritmos criptográficos. Cada uno proporciona algunos o todos los servicios de aplicación que se exponen a continuación.

2.2.1 Confidencialidad

Los datos son secretos incluso si los datos viajan a través de un medio inseguro. En la práctica esto significa que atacantes potenciales podrían ser capaz de ver datos que en principio son ilegibles y que están bloqueados, pero ellos no deberían ser capaz de desbloquear los datos sin tener la información adecuada. En la criptografía clásica, los algoritmos de encriptación son secretos, en la criptografía moderna esto no es así, los algoritmos son de código libre y las llaves criptográficas son usadas para encriptar y des-encriptar los procesos o datos. La única cosa que necesita es que la llave sea secreta, adicionalmente, en algunos casos existen escenarios que no necesitan tener la llave secreta[11].

[11] Pravir Chandra, Matt Messier, John Viega. "Network Security with OpenSSL", Publisher : O'Reilly, Pub Date : June 2002, ISBN : 0-596-00270-X, Pages : 384

2.2.2 Integridad

La idea básica en la integridad de los datos es determinar una manera de verificar que una pieza de datos haya tenido alguna modificación en el camino sobre algún periodo de tiempo. Revisar la integridad puede ser usada para asegurar que los datos enviados sobre el medio no han sido modificados durante su recorrido. Según [8] muchos *checksum* existentes pueden detectar e incluso corregir algunos errores simples. Sin embargo, tales *checksum* son pobres intentos de modificaciones robustas de los datos. Es importante aclarar que la encriptación no asegura la integridad de los datos y tampoco asegurar que algún atacante pueda tener existo en cambiar algún bit de datos y así alterar la información (***bit-flipping***).

2.2.3 Autenticación

La criptografía puede ayudar a establecer identidad para propósitos de autenticación.

2.3 No repudio

Para garantizar la participación de una sesión, es necesario que existan dos partes involucradas (emisor y receptor) donde se puede distinguir dos tipos de no repudio: (i) **No repudio en origen:** garantiza que la persona que envía el mensaje no puede negar que él es el emisor debido a que el receptor tendrá información de quien es el emisor del mensaje. (ii) **No repudio en destino:** Pasa lo contrario, el receptor no pude negar que él es el receptor del mensaje, ya que el emisor tiene información de quien es el receptor del mensaje. En el mundo real no se puede asumir que un atacante no comprometa la sesión y modifique la encriptación de las llaves. **El protocolo TLS no soporta no-repudio**, pero existen una manera sencillas de añadirlo usando firmas digitales. Con este servicio se puede mitigar gran variedad de ataques de red, incluyendo las siguientes funciones.

2.3.1 Snooping

Un atacante mira el tráfico de red que está pasando por la red y capturas datos importantes en la sesión como información en tarjetas de crédito.

2.3.2 Tampering

Un atacante monitorea tráfico de red y altera los datos en tránsito (ej. Un atacante puede modificar el contenido de un email)

2.3.3 Spoofing

Un atacante forja a los datos de red, para suplantar la identidad del usuario final.

2.3.4 Hijacking

Una vez un usuario legitimo es autenticado, un ataque *spoofing* puede ser usado para *hijack* la conexión

Actualmente e investigando información adecuada es posible desplegar algunos de los ataques mencionados anteriormente, como la gran mayoría son herramientas usan la filosofía de *open source* es posible explotar alguno de estos ataques, como ejemplo existe ***dsniff*** [12]

Tradicionalmente los protocolos de red tal como HTTP, SMTP, FTP, SMTP y Telnet no proveen tipos de defensa adecuada para evitar estos ataques. Mientras que muchos protocolos proveen algún tipo de autenticación basado en *login*, muchos de ellos no cumplen lo que recomienda tener confidencialidad e integridad de datos.

TLSv2/SSL es una gran opción para los protocolos tradicionales de red, porque hace fácil añadir confidencialidad e integridad de manera transparente a servicios basados en TCP/IP. Esto puede proveer autenticación, lo que genera que los clientes estén confiados que están hablando con el servidor deseado.

2.4 Algoritmos criptográficos

TLSv2/SSL incluye muchos algoritmos criptográficos, por ejemplo, en el caso que desee encriptar cookies HTTP que serán ubicadas en un navegador final. **TLS no ayudará a proteger las cookies mientras estén siendo almacenadas en disco**, en situaciones como estas, OpenSSL exporta algoritmos criptográficos que son usados en el protocolo TLS.

En general, según [8] no es recomendable usar directamente algoritmos criptográficos. Ya que no es probable obtener una seguridad total a ataques

Algunos de los algoritmos más conocidos que se implantan muy a menudo son: **encriptación de llave simétrica, encriptación de llave pública, funciones *hash* criptográficas, código de mensajes de autenticación y firmas digitales**

[12] https://www.monkey.org/~dugsong/dsniff/

2.4.1 Encriptación de llave simétrica

Los algoritmos de llave simétrica encriptan y des-encriptan datos usando una sola llave, como muestra la siguiente figura, la llave y el dato en claro es pasado por un algoritmo criptográfico, el resultado puede ser enviado a través de cualquier medio (inseguro), permitiendo solo un receptor, y es quien tiene la llave original para des-encriptar el mensaje.

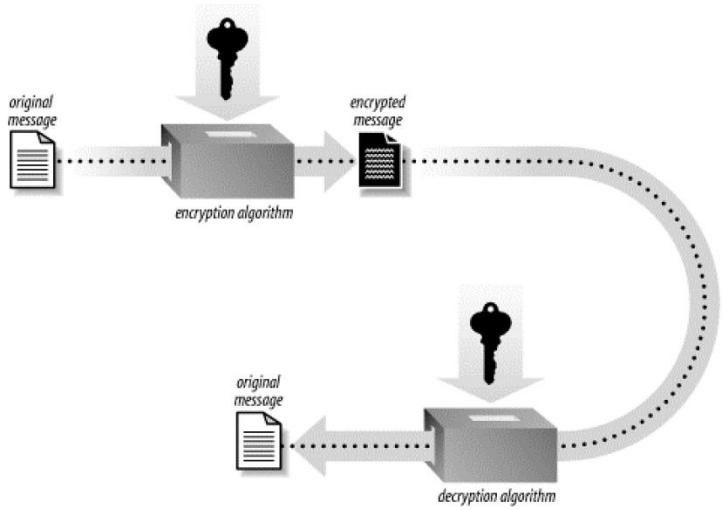

Figura 5. Criptografía de llave simétrica.

Una desventaja de este algoritmo es que la llave deber permanecer secreta todo el tiempo. En particular, el intercambio de llaves secretas puede ser de tedioso, ya que normalmente lo que se necesita es intercambiar llaves en el mismo medio físico que usa encriptación como medio de protección, enviando esta llave en claro se corre el riesgo de que cualquier atacante interesando en la información pueda interceptar la llave y copiar su estructura antes de empezar a enviar los datos.

Una solución al problema de distribución de llaves es usar un protocolo de intercambio de llaves. Para este propósito, OpenSSL provee el protocolo *Diffie-Hellman*, el cual permite un acuerdo de llaves sin necesidad de intercambiar llaves dentro de la red. Sin embargo, *Diffie-Hellman* no garantiza la identidad de la otra parte con que se quiera intercambiar llaves. Este es algún tipo de mecanismo de autenticación que es necesario para asegurar que no se ha intercambiado accidentalmente la llave con una fuente desconocida.

DES o Triple DES (3DES o DES3) es un cifrado simétrico disponible hace varios años pero un poco más conservador a diferencia de AES (*Advanced Encryptation Standart*) que eventualmente reemplaza a DES ya que es más rápido.

2.4.2 Encriptación llave pública

La forma más popular de hacer criptografía de llave pública es que cada parte tenga dos llaves, una debe permanecer secreta (*private key*) y la otra puede ser distribuida (*public key*). Las dos llaves tienen relación matemática. Como se muestra en la siguiente figura. Para que Alice envíe un mensaje a Bob usando encriptación de llave pública, Alice primero debe tener la llave pública de Bob, posteriormente ella enviará su mensaje usando la llave pública de Bob y lo entregará. Una vez encriptado, solo alguien que tenga la llave privada de Bob podrá des-encriptar el mensaje.

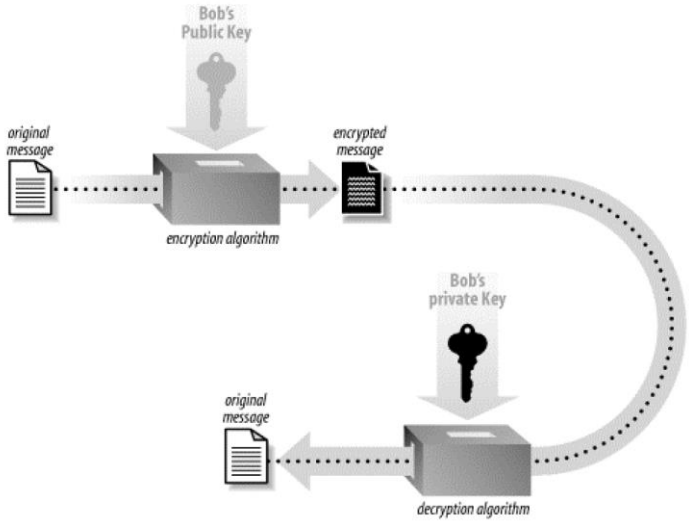

Figura 6. Criptografía de llave pública.

La encriptación de la llave pública soluciona el problema de distribución de llaves, esto asumiendo que solo hay una forma de encontrar la llave pública de Bob y asegurar que esa llave pertenece a Bob. En la práctica. La llave pública es enviada con un certificado y aquellos certificados son validados por un ente certificador. Regularmente**, un ente certificador es una organización dedicada a la investigación para aquellas personas que desean tener sus certificados validados.** La criptografía de llave pública generalmente está limitada por los recursos de máquina, no por el ancho de banda que posee el objetivo. Normalmente las máquinas servidor necesitan suficiente recurso hardware para manipular múltiples conexiones de manera simultánea.

Como resultado, muchos sistemas que usan criptografía de llave pública (incluida TLSv2), usan este recurso lo menos posible, generalmente, este tipo de criptografía es usada para acordar una clave de cifrado a un algoritmo simétrico y posteriormente realizar encriptación. Por otra parte, este tipo de información es utilizada cuando se da el intercambio de protocolos y cuando no sea requerido el no repudio.

RSA es el algoritmo de encriptación más popular, *Diffie-Hellman* es basado en criptografía de llave publica y puede ser utilizado para lograr el mismo fin (intercambiar llave simétrica) el cual es útil para encriptar y des-encriptar los datos. Para que este tipo de criptografía sea más eficiente es necesario mecanismos de autenticación de un ente certificador, aunque los esquemas de firmas digitales también proporcionan este tipo de autenticación. Es recomendable utilizar llaves con una longitud mayor a 1024 bits, usar un valor menos podría generar brechas de seguridad en la autenticación. En forma práctica, es recomendable usar longitudes de 1024 bits, pero si el deseo es tener seguridad robusta, entonces es posible usar longitud de 2048 bits.

2.4.3 Funciones de Hash criptográficos y códigos de mensajes de autenticación

Las funciones de hash criptográficos son esenciales para la comprobación de propiedades especiales. Los datos se evalúan por una función *hash* y el resultado es un valor checksum (*message digest)*, así que pasando los datos por la misma función *hash* dos veces será siempre evaluado con el mismo valor, sin embargo ese valor no ofrece mucha información acerca de los datos dentro de esa función, por lo que es casi imposible encontrar dos entradas que produzcan el mismo *hash*. Cuando se refieren a estas funciones, se habla que son funciones de una vía, es decir, que no debería ser posible tomar el valor de salida del *hash* y reconstruir el valor de entrada a partir de ese valor.

MD5 y SHA1 son una de las funciones *hash* más populares que ofrecen criptografía de una vía. MD5 tiene una longitud de 128 bits mientras que SHA1 tiene una longitud de 160 bits, el ciertos casos es recomendable usar MD5, pero en otros casos puede ser arriesgado, esto depende del tipo de aplicación que el desarrollador quiera implantar, así que lo esencial es usar funciones *hash* que superen una longitud de 160 bits

Las funciones *hash* pueden ser usadas en diversos casos. Frecuentemente hace parte como uso de solución de almacenamiento de contraseñas, otro ejemplo de utilidad se relaciona con los archivos, ya que OpenSSL puede enlazarse junto a MD5 para realizar el checksum de un fichero. Cuando el fichero es descargado, también es posible descargar el *checksum* del mismo con el fin de computar que el checksum del fichero descargado corresponde al *checksum* inicial. Infortunadamente en este caso no hay ningún secreto involucrado, ya que un atacante puede reemplazar el fichero con una versión modificada y reemplazar el *checksum* con uno que corresponda. Esto es posible porque el algoritmo es público y no hay información secreta en este. En caso que se esté usando una llave secreta con un distribuidor software, **el distribuidor podría combinar el archivo con la llave secreta para producir un *checksum* que un atacante no pudiese romper ya que él no tendría la llave secreta. Este tipo de esquema con**

llave secreta es conocido como MACs (*Messages Authentication Codes*). MAC es a menudo usado para proporcionar integridad en la transferencia de los mensajes. En efecto TLSv2 usa MAC para este propósito, aunque actualmente OpenSSL y TLSv2 soporta MAC tal como HMAC, GMAC y OMAC [13]

2.4.4 Firmas Digitales

En varias aplicaciones MAC no es un sistema útil de utilizar ya que requiere de compartir una llave secreta entre ambas partes. Lo ideal sería que fuese capaz de autenticarse sin necesidad de compartir llaves secretas o algún secreto entre las partes y además siga siendo seguro. Criptografía de llave pública hace posible esto. Por ejemplo, si Alicia firma un mensaje con su firma secreta, entonces nadie podría usar su llave pública para verificar que ella firmó el mensaje. RSA es usado para firmas digitales, en principio las llaves pública y privada son intercambiadas. Si Alice encripta un mensaje con su llave privada, nadie puede des-encriptar el mensaje, si Alice no encripta el mensaje, usando su llave publica para des-encriptar su mensaje podría resultar innecesario.

Hay también un esquema popular llamado DSA (*Digital Signature Algorithm*), el cual es soportado por TLSv2 y OpenSSL.

Si Alice está dispuesta a validar al certificado de Bob, ella puede firmarlo con su llave privada, una vez hecho esto, Bob puede adjuntar su firma al certificado. Ahora, existe el caso que él haya dado el certificado a Charlie, pero él no tiene conocimiento que Bob se lo ha dado, en casos como estos, Charlie puede validar la firma de Alice y de este modo demostrar que el certificado en efecto pertenece a Bob. Ya que las firmas digitales son una forma de criptografía de llave pública, además, se debe asegurar que la longitud de estas sea de 1024 bit o superior para aumentar la seguridad.

2.5 Hypertext Transfer Protocol y Hypertext Transfer Protocol Secure

HTTP fue introducido en el mercado en los años 90, pero desde entonces, las cosas en internet han cambiado significativamente convirtiendo este protocolo de comunicaciones en parte vital de la vida tanto para educación como comercio. HTTPS es la parte segura de HTTP ya que corre sobre una capa de seguridad que emplea TLSv2. Inicialmente fue dirigido para servicios que requerían confidencialidad de los datos entre cliente y servidor, como ejemplo común, esta aplicabilidad se puede ver en Facebook, Gmail, Youtube, GitHub, entre otros.

[13] http://stackoverflow.com/questions/29657347/what-mac-message-authentication-code-algorithms-supported-on-openssl

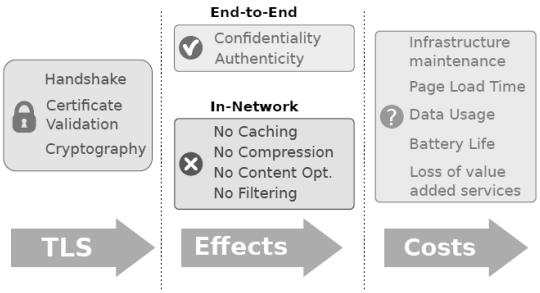

Figura 7. Adopción de TLS a HTTP.

HTTPS está al borde un nuevo despliegue ya que según [9] se discute que TLS en cualquier versión será usada para todas las conexión existentes, reflejando un diseño establecido por SPDY [10] el cual fue usado como punto de partida para HTTP 2.0

Dada a la creciente preocupación de la privacidad en internet, encriptar datos a través de HTTP suena una buena idea, sin embargo esta opción tiene su costo. Si se analiza la acogida que tiene HTTPS como servicio, el costo puede menguar debido a que el 50% del tráfico es transmitido de forma segura.

A su vez, TLS tiene un impacto en la latencia de los datos y proceso de CPU por las operaciones criptográficas que se deben hacer en los clientes, además, HTTPS requiere un *handshake* adicional entre el cliente y servidor. Por lo que conlleva a que HTTPS incremente costo de su desarrollo y verse afectado en consumo de batería de los dispositivos, esto lleva a depender de varios factores, uno de ellos es las preferencias personales y el tipo de utilidad dada.

Muchas han sido las variantes propuestas ante esta problemática: aceleradores hardware, arquitecturas GPU [11] o rebalanceo de RSA [12]. También TLS provee una negociación rápida como muestra la siguiente figura.

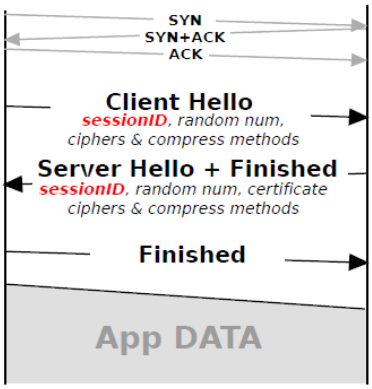

Figura 8. Mecanismo de negociación rápida

En este caso, una *Session ID* es usada para recuperar una sesión establecida previamente evitando el costo de crear una nueva sesión y reducir el *Handshake* a 1 RTT. [14]

El despliegue de HTTPS incrementa costos de infraestructura (computo, memoria, gastos generales en red) y adicionalmente costo económico ($1.999 por año), esto hace que HTTPS sea implementado solo para uso necesario.

2.6 Conceptos básicos de Open Secure Sockets Layer

OpenSSL (Open Secure Sockets Layer) es un trabajo derivado de SSLeay. SSLeay fue escrito originalmente por Eric A. Young y Tim J. Hudson en 1995. En diciembre de 1998 fue lanzada la primera versión de OpenSSL 0.9.1c, usando SSLeay 0.9.1b (el cual nunca fue lanzada), este fue el punto de inicio de OpenSSL el cual trabaja con dos herramientas: librerías criptográficas y una kit de herramientas SSL.

Las librerías de SSL (TLS) proveen una implementación de todas las versiones del protocolo TLS. Estas librerías proveen los algoritmos más populares para criptografía de llave pública, simétrica y algoritmos *hash*. También provee un generador de números pseudorandomico, soporte para manipulación de formato de certificados y administración de material clave, manipulación de buffer, entre otras.

OpenSSL es una tecnología open source, por lo cual es libre y está disponible para su uso en C, C++, Python, entre otros lenguajes de programación, también trabaja en diversos sistemas operativos, tanto en Unix OS, Mac OS X y todas la versiones de Windows.

Muchas aplicaciones están ya construidas para que soporten OpenSSL, por ejemplo, OpenSSH y Apache usan OpenSSL y requiere sus librerías para que pueda compilar. En caso que existan aplicaciones que no usen TLS de forma nativa, existe **Stunnel**[15] para proveer comunicación segura entre el cliente y servidor.

2.6.1 Librería OpenSSL

OpenSSL es principalmente una librería usada por desarrolladores para incluir soporte criptográfico en sus programas, a su vez, también es una herramienta que provee acceso a muchos funcionalidades por línea de comando. Por línea de comandos se hace fácil realizar operaciones comunes tal como *hash* MD5 de ficheros. La línea de

[14] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, Peter Steenkiste. "The Cost of the "S" in HTTPS".

[15] https://www.stunnel.org/index.html

comandos provee la habilidad de acceder a muchos niveles superiores de OpenSSL desde el Shell en Unix.

Cifrado simétrico

OpenSSL soporta una gran variedad de cifrado simétrico como *Blowfish*, CAST5, DES, 3DES, IDEA, RC2, RC4, RC5 y AES. Muchos de los cifrados simétricos soportan una variedad de diferentes modos, incluyendo CBC, ECB y OFB, el modo por defecto es siempre CBC si el modo no es especificado. En particular, es importante mencionar que nunca se debería usar ECB porque es mucho más complicado.

Diffie-Hellman

TLS contiene una gran variedad de diversos algoritmos criptográficos, incluyendo, algoritmos *message digests*, cifrado simétrico y criptografía pública. El uso de estos algoritmos se genera sin necesidad de intervención humana. Por ejemplo, para que un servidor emplee TLS, es necesario una llave privada y un certificado. El certificado contiene la llave pública que relaciona el servidor y llave pública. Esto debe ser creado como parte del proceso de configuración del servidor para que use TLS y esto debe ser configurado por quien configure el servidor.

TLS no es solo un protocolo que usa criptografía de llave pública, y algunos de los ejemplos más populares son SSH, PGP (*Pretty Good Privacy*) y S/MIME. Todos estos ejemplos usan criptografía de llave pública.

Diffie-Hellman (DH) usa *key agreement*. Es decir, es el intercambio de información sobre un medio inseguro que permite en cada una de las dos partes de la conversación computar un valor que es usado como una clave para realizar cifrado simétrico**. DH no puede ser usado para encriptación y autenticación. Esto solo provee protección ya que el intercambio de información toma lugar sobre un medio inseguro**, así que nunca debería ser usado para estos casos.

DH trabaja creando un conjunto de parámetros que son añadidos por ambas partes de la conversación. Estos parámetros consisten de un número randómico y un generador de valores. Estos parámetros son públicos y pueden ser añadidos antes de que comience la conversación. Usando este método, cada parte computa una llave pública y privada. La llave privada nunca es compartida en ningún caso. Las partes intercambian las llaves públicas y luego cada parte puede compartir información usando la llave privada.

Algoritmo de firma digital

Digital Signature Algorithm (DSA) es usando para crear y verificar firmas digitales. Provee autenticación pero no es usado para encriptación. DSA frecuentemente es usado en combinación con DH. Las dos partes de la conversación pueden intercambiar

DSA de llaves públicas antes de que empiece la conversación (o durante la conversación usando certificados) y usa clave DSA para autenticar la comunicación de DH.

Al igual que DH, DSA también requiere parámetros de cuáles llaves son generadas. No existe inconveniente si se usan estos parámetros para generar un par de llaves públicas. La llave pública es la única que se necesita para compartir datos con cualquier destino que desee verificar la autenticación de cualquier firma con una llave privada.

RSA

RSA es el algoritmo de llave pública más popular. La razón que lo hace tan popular es porque provee discreción, autenticación y encriptación en un solo paquete.

A diferencia de DH y DSA, RSA no requiere parámetros generados antes de que se generen las llaves. Lo que simplifica la cantidad de trabajo necesario para generar llaves, autenticación y encriptación

2.7 Certificados

Un certificado enlaza una llave pública con un nombre distintivo, un nombre distintivo es el nombre de una persona o entidad a quien le corresponde ese certificado. Esto es comparado con un pasaporte, que enlaza una foto con un nombre, fortaleciendo la identidad de la persona. Un pasaporte tiene un ente externo quien lo controla (el gobierno) que contiene información acerca de la persona a quien le corresponde el pasaporte.

Los certificados contienen un tipo de salvaguarda destinado a permitir la autenticidad del certificado para ser verificado, a su vez, ayuda a detectar ataques de falsificación o manipulación de datos. Como los pasaportes, los certificados también tienen fecha de caducidad, una vez caducado, un nuevo certificado debe ser emitido, y el anterior ya no debería ser seguro.

Un certificado debe ser firmado por los emisores de la llave privada, esto contiene casi toda la información necesaria para verificar esta validación, es decir, contiene información acerca del objetivo, emisores y la fecha del cual fue validado. El componente clave que falta es el certificado del emisor. **En el certificado emisor está el componente clave para verificar la validación de un certificado**, porque allí está la llave pública del emisor, el cual es necesario para verificar la firma en el certificado cliente.

Hay que tener en cuenta que cualquiera que tenga la llave pública del emisor puede verificar su autenticidad, por ende, este método es útil para prevenir manipulación. Además, también es importante destacar que los certificados emisores o llaves públicas pueden estar dentro de un certificado emisor. Por lo que esta información no es segura para autenticar el certificado, si fuera así, el elemento de confianza establecido como CA sería innecesario, cualquiera podría crear otro par de llaves para firmar un certificado y colocar esa llave publica en el certificado.

Los certificados son creados con un número único para el emisor del certificado. No existe dos certificados iguales así hayan sido emitidos por la misma persona ya que tienen diferente número serial. Este número serial es usado a menudo para identificar un certificado rápidamente.

2.7.1 Autoridades certificadoras

CA puede tener varios nombres, es conocido como autoridad certificadora, ente certificador, entidad certificadora, entre otros. Estos nombres se relacionan como una organización o compañía que certifica los emisores, por lo tanto tiene una gran responsabilidad para asegurar que los certificados emisores son legítimos. Además, una CA debe asegurar más allá de toda duda creada que todos los certificados emitidos tienen una llave publica que fue emitida por parte del que alega haberla emitido.

Hay quien dude de una CA, pero normalmente hay dos tipos de bases para afirmar que una entidad certificadora es confiable. Primero, una CA privada tiene la responsabilidad de emitir certificados solo a miembros de la organización y se confía solo del miembro perteneciente a esa organización. Una CA publica, tal como **VeriSing** [16]o **Thawte**[17] tiene la responsabilidad de emitir certificados a cualquier miembro del público y debe ser confiable para el público. Una CA debe ser confiable y para que esto sea así, la llave publica de los certificados emitidos deben ser extensamente distribuidos, para una CA publica, sus certificados son generalmente publicados para que cualquiera puede hacer uso de ellos, un ejemplo son los navegadores web como Chrome o Firefox

Autoridad certificadora privada

Una entidad certificadora privada es ideal para uso corporativo, por ejemplo, una compañía configura su propio CA para emails. Usando S/MIME como estándar de encriptación y autenticación de mensajes email. La compañía CA emitiría certificados para cada empleado y cada empleado debería configurar su S/MIME como cliente email y así mantener la compañía asegurada.

[16] https://www.verisign.com/?dmn=www.verisign.es
[17] https://www.thawte.com/

Autoridades certificadoras públicas

Una entidad certificadora publica normalmente emite certificados para sitios web que requieran encriptación o autenticación, tal como plataformas *e-commerce* donde los clientes deben compartir información secreta, en estas operaciones es fundamental que el cliente trasmita información al sitio web sin preocuparse de la integridad de sus datos.

Normalmente el trabajo es mucho más difícil para una entidad certificadora pública que una entidad privada, por lo tanto, los proveedores de certificados públicos cobran por prestar este servicio.

Extensión de certificados

El formato más común para certificados es el X.509 [18], fue introducido en 1988. Hay otras versiones del formato conocidas como: X509.v1, X509.v2 y X509.v3, la revisión más reciente del estándar fue hecha en 1996.

El X509.v3 está definido en 14 extensiones que han venido consolidando esfuerzos de desarrollo hecho por entes externos. De las 14 extensiones del estándar solo cuatro han sobrevivido y a las cuales dan soporte actualmente

2.8 Network Function Virtualization

Los operadores de redes se basan en un sin número de funciones intermediarias (NFs) tales como NAT, balanceadores de carga, firewalls, sistema de detección de intrusos (IDs), *proxies*, controles parentales, etc. Tradicionalmente, estas funciones son implementadas a través de hardware o más conocidas *"middle-boxes"* pero en este caso usando poder hardware, es decir, basadas en plataformas hardware que son costosas, difícil de mantener y actualizar. Así que, con la nueva ideología de virtualización en redes, es posible desplegar estos middle-boxes llevándolas a nivel software, convirtiéndose en VNF (*Virtual Network Functions*)[19].

Al hablar de VNF es imprescindible que exista un nuevo método de virtualización en redes como NFV, es decir, cuando se habla de VNF es necesario tener el concepto de NFV, por lo tanto, a pesar que el objetivo de esta memoria no aplique esta tecnología. Se considera necesario señalar algunos significados expuestos para NFV ya que *middle-boxes* pueden hacer parte de una red virtualizada.

NFV transforma la arquitectura e infraestructura de los operadores de red a estructuras virtualizadas. NVF implementa funciones de redes mediante técnicas de

[18] https://www.ietf.org/rfc/rfc5280.txt

[19] Sevil Mehraghdam, Matthias Kelle, Holger Karl, "Specifying and Placing Chains of Virtual Network Functions"

virtualización ejecutadas en hardware (servidores industriales, almacenamiento, switches, etc.), estas aplicaciones puede ser instanciadas sin la necesidad de equipo experto para ello, es decir, los operadores pueden correr un software *open source* que realice las tareas de firewall dentro de una máquina virtual x86 (VNF).

Con esta innovación se gana agilidad en las infraestructuras de los operadores de red ya que NFV traería beneficios sustanciales a la industria de las telecomunicaciones, reduciendo capital invertido, consumo de energía, consolidando las aplicaciones de red y aumentando el ciclo de innovación en los operadores de red.

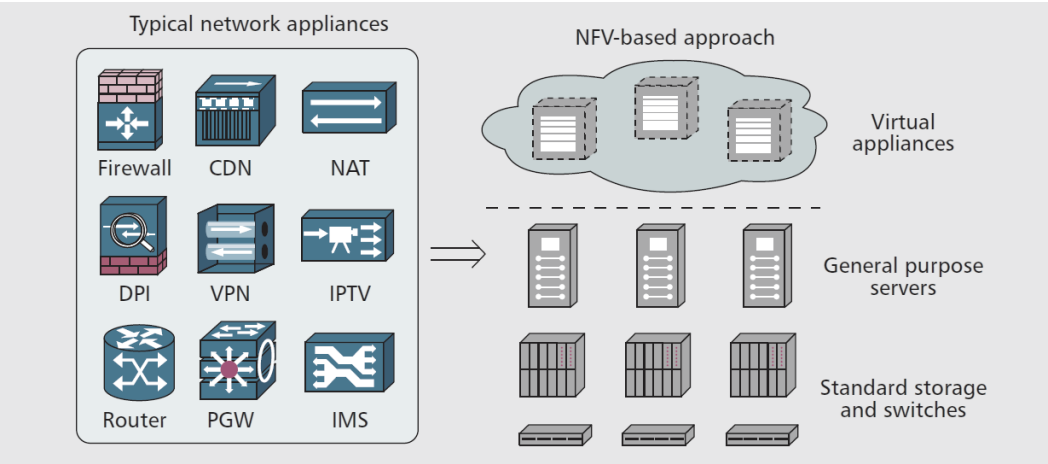

Figura 9 NFV

En algunos estudios realizados sobre NFV [14] citan que esta tecnología ha surgido como una iniciativa promovida por los operadores de red con el objetivo de incrementar la flexibilidad en los servicios de redes. Este logro significa implementar soluciones de red mediante software, evitando soluciones hardware para mejorar el costo de inversión. Este concepto aplica para numerosos elementos de red gracias a la programabilidad mejorada de las redes. NFV transforma el método con que los operadores diseñan sus redes aunque sin dejar de lado que son juntamente complementarios, por lo tanto, teniendo en cuenta esta definición. Cada elemento de red tiene un conjunto de diferentes funciones que puede ser potencialmente extraída y puestas en elementos externos que después pueden ser movidos, instanciados, duplicados y administrados de manera individual. Este propósito abre un amplio rango de posibilidades que pueden ser adoptados en diferentes escenarios.

NFV prevé la implementación de NFs como entidades implementadas por software que corran sobre infraestructuras NFVI (infraestructura de NFV), así como lo muestra la siguiente figura, NFV tiene tres principales dominios de trabajo.

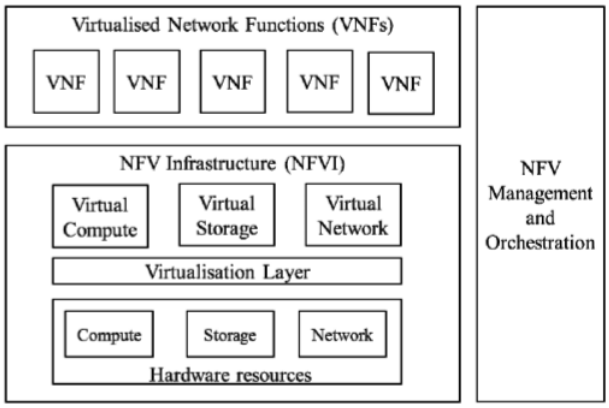

Figura 10 Framework NFV

4 Funciones de redes virtualizadas, es la implementación software de una NFs la cual es capaz de correr sobre NFVI.
5 NFVI, incluye la variedad de recursos físicos y cómo aquellos tienen la capacidad de ser virtualizados, por lo tanto NFVI permite la ejecución de VNFs.
6 Orquestación y administración de NFV, se convierten en factores trascendentales para orquestar y gestionar recursos software y hardware que soporta una infraestructura de virtualización, además de la gestión de VNFs.

Según [24] existen bloques funcionales dentro de la arquitectura NFV que son puntos de referencia esenciales de los cuales ya están presentes en despliegues actuales, sin embargo hay otros que podrían ser necesarios para soporte de virtualización y operación. Algunos de estos bloques son:

1 *Virtualised Network Function* (VNF).
2 *Element Management* (EM).
3 *NFV Infrastructure* (Hardware y recursos de virtualización | Capa de virtualización)
4 Gestión de la infraestructura virtualizada.
5 Orquestador NFV.
6 *Service, VNF and Infrastructure Description*.
7 *Operation and Business Support Systems* (OSS/BSS).

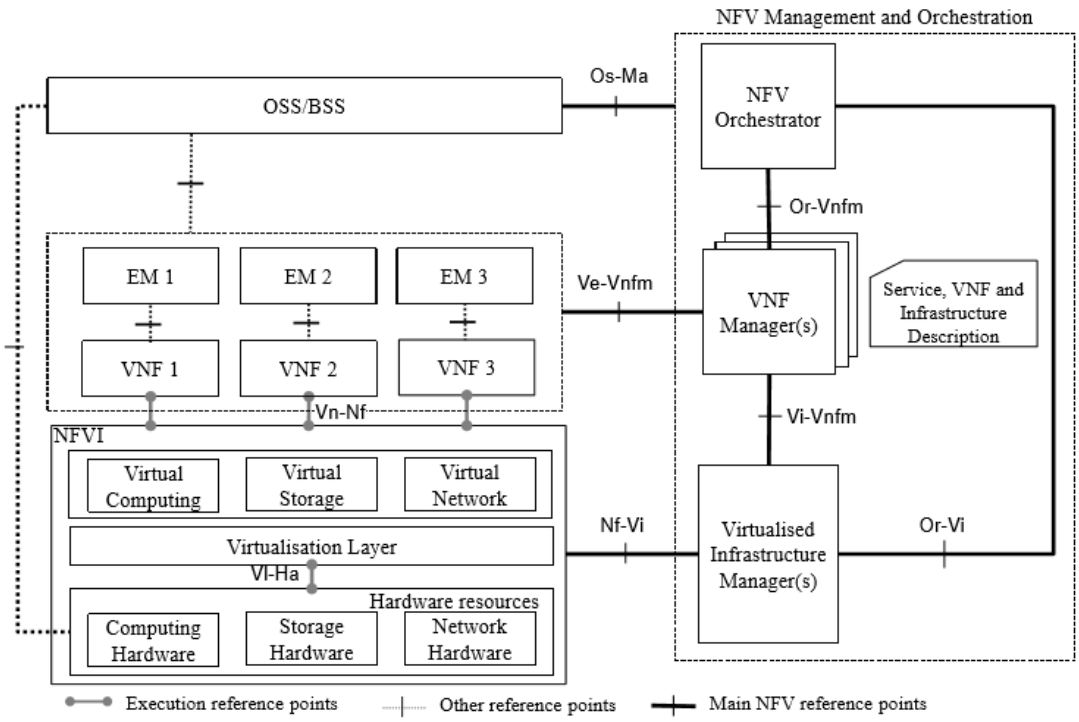

Figura 11 Arquitectura NFV.

La anterior figura muestra la arquitectura de NFV que está representada en bloques funcionales y puntos de referencia. Los puntos de referencia principales están representados con línea continua y son el alcance de NFV. Estos son los puntos esenciales de estandarización. Lo que está en línea punteada son puntos de referencia que podrían necesitar extensiones para manipular las funciones de red virtualizada, sin embargo estas puntos de referencia no son el enfoque principal de NFV.

2.8.1 Virtual Network Function

Una VNF es una función que se encuentra en la red capaz de realizar tareas específicas dentro de la red y está hecha de forma virtualizada. Alguno de los ejemplos de una NFs son 3GPP, *Envoled Packet Core* (EPC), elementos de red como: *Mobility Management Entity* (MME), *Serving Gateway* (SGW), *Packet Data Network Gateway* (PGW), *gateways* residenciales (RGW), DHCP, firewalls, etc. La lista de casos de uso para VNF puede ser encontrada en ETSI GS NFV 001 [25].

El comportamiento funcional de NFV son ampliamente independientes de si esta virtualizada o no. **Una VNF puede ser compuesta de múltiples componentes internos, es decir. Una VNF puede ser desplegada sobre múltiples VM (Virtual Machine), donde cada VM es un componente VNF.** Sin embargo hay casos donde toda la VNF es desplegada en una VM.

2.8.2 EM

Realiza funciones de gestión para una o varias VNFs, es decir, gestión para: fallos, configuración, contabilidad, rendimiento y seguridad.

2.8.3 Infraestructura NFV

Son todos los componentes hardware y software el cual construyen el ambiente donde las VNF son desplegadas, gestionadas y ejecutadas. Desde la perspectiva VFN, la capa de virtualización y recursos hardware parece una entidad que provee VNF con los necesarios recursos de virtualización.

2.8.4 Recursos Hardware

Para NVF, los recursos hardware incluyen, procesamiento, almacenamiento y red. La conectividad y almacenamiento de VFNs son a través de una capa de virtualización (*hypervisor*), los recursos de almacenamiento pueden ser diferenciados entre redes compartidas de almacenamiento y almacenamiento ubicado en el propio servidor.

Los recursos de red son abarcados por funciones de switching como: Routers, enlaces por cable o inalámbricos. Por otra parte, los recursos de red pueden abarcar diferentes dominios. Sin embargo, NFV solo sigue dos tipos de dominios:

1 Redes NFVI-PoP: Red que interconecta recursos de *computing* y almacenamiento compuestos en una NFVI-PoP. Incluye también dispositivos de switching y routing para permitir la conectividad al exterior.
2 Red de transporte: interconecta NFVI-PoP unas con otras de diferentes operadores.

2.8.5 Gestión de la infraestructura virtualizada

Desde el punto de vista de NFV. La gestión de la infraestructura virtualizada comprende funcionalidades que son: controlar y gestionar la interacción de VFN con computing, almacenamiento y recursos de red bajo esta autoridad. De acuerdo a la lista de recursos hardware especificados en la infraestructura, la gestión de la infraestructura virtualizada realiza:

1 Inventario del software (*hypervisor*), *computing*, almacenamiento y recursos de red dedicados a la infraestructura NFV.
2 Ubicar los facilitadores de virtualización, es decir, VM con *hypervisor*, recursos de computación, almacenamiento y conectividad de red.

3 Administrar recursos hardware para las VMs, mejorar eficiencia energética, etc.
4 Recoger información de fallos en la infraestructura.
5 Recoger información relacionada con monitoreo, optimización, planeación, etc.

2.8.6 Orquestador NFV

El orquestador NFV es el encargado de la gestión de la infraestructura y recursos software, además de realizar servicios de red sobre NFV.

2.8.7 VFN Manager

Es el responsable del ciclo de vida de las VNFs, con funciones como: instalación, actualización, escalabilidad, terminación, etc.

Múltiples VFN Manager pueden ser desplegadas; también puede ser desplegada una VNF Manager por cada VNF instanciada o solo una como master.

2.8.8 Service, VNF and Infrastructure Description

Este conjunto de datos provee información con respecto al modelo desarrollado de la VNF. Estos modelos o descriptores son usados internamente dentro de la gestión y orquestación NFV. Los bloques funcionales manipulan información contenida en los descriptores tanto como sea necesario.

2.8.9 Operation and Business Support Systems (OSS/BSS).

OSS hace el acuerdo con gestión de redes, gestión de fallos, gestión de configuración y servicios. Mientras que BSS hace acuerdo con clientes, gestión de productos y orden, etc.

2.8.10 Puntos de referencia.

Capa de virtualización – Recursos Hardware (VI-HA)

Crea un ambiente de ejecución para VNFs y colecciona estado de información relevante de recursos hardware para gestionar las VNFs sin ser dependientes de cualquier plataforma hardware.

VNF – NFV Infraestructura (Vn-Nf)

Representa el ambiente de ejecución provisto por NFVI a VFN. No asume ningún protocolo de control, esto está en el alcance de NFV para garantizar ciclo de vida independiente del hardware.

Orquestación NFV – VNF Manager (Or-Vnfm)

Hace referencia a puntos relacionados con peticiones; autorización, validación, reserva y ubicación para la VNF Manager.

Envía información de configuración al VNF Manager, así, la VNF puede ser configurada apropiadamente.

Gestión de infraestructura virtualizada – VNF Manager (Vi-Vnfm)

Hace referencia a puntos como recursos solicitados por la VNF Manager, configuración de recursos hardware e información de estado, es decir, eventos intercambiados.

NFVI – Gestión de infraestructura virtualizada (Nf-Vi)

Hace referencia a puntos como asignación específica de recursos virtualizados en respuesta a los recursos solicitados, además de reenvío de información de aquellos recursos virtualizados, configuración de recursos hardware e información de estado, es decir, eventos intercambiados.

OSS/BSS – Orquestación y administración NFV (Os-Ma)

Hace referencia a puntos tales como peticiones para servicios de red, para gestión de la VNF con respecto a su ciclo de vida, reenvío de información de estado, intercambio de políticas de gestión, intercambio de análisis de datos, intercambio de información sobre inventarios y capacidad de NFVI.

VNF/EM – VNF Manager (Ve-Vnfm)

Hace referencia a puntos como, gestión de la VNF con respecto a su ciclo de vida, intercambio de información de la configuración, intercambio de información necesaria para la gestión de servicios de red.

2.8.11 Máquina virtual

Una máquina virtual [19] es una implementación software que emula una máquina real, la siguiente figura muestra la arquitectura que maneja una máquina virtual (VM), es instanciada a través de un *hypervisor* que está instalado en un host físico del cual toma recursos como CPU, RAM, espacio en disco duro, entre otras. En un nivel superior, hay un software o sistema operativo como sistema invitado donde están alojadas las aplicaciones de cada máquina, cada una es independiente y es administrada por el hypervisor.

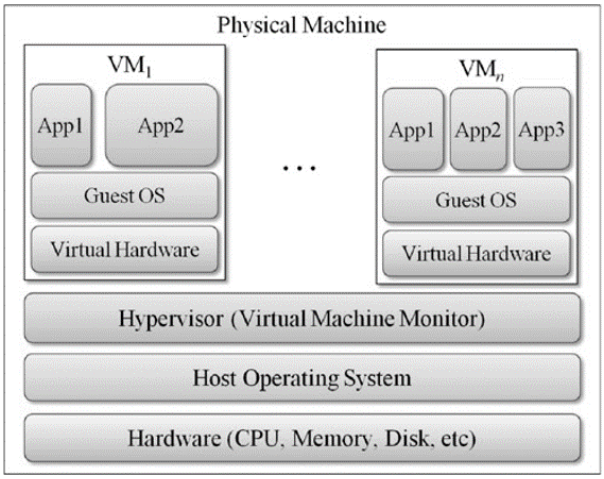

Figura 12 Arquitectura de máquina virtual

Un *hypervisor* es la unión de hardware y software que permite compartir recursos físicos entre host y VM. Es una aplicación que corre en el host y por lo tanto es susceptible de sufrir riesgos cuando incrementa el volumen y la complejidad de la aplicación.

Existen amenazas y atacantes que afectan la seguridad en hosts físicos, por lo que las redes virtuales se encuentran menos amenazadas, lo que hace a las VM tener menos ataques pero sin estar exentas de ser atacadas [20], sin embargo, la virtualización de redes podría traer vulnerabilidades como DoS para atacar el ambiente virtualizado y afectar la comunicación entre VM, en estimaciones, existen un 60% de VM en producción que es menos segura que los hosts físicos, y un 30% de despliegues con VM que involucran incidentes de seguridad [21].

Una red virtualizada, es una ambiente que permite comunicación entre múltiples VM en un mismo entorno físico. Cada red virtual (VN) en un ambiente virtualizado de red (NVE) contiene nodos virtuales y enlaces virtuales, es decir. Una red virtual es un subconjunto fundamental de recursos de red físicos.

La virtualización de redes propone desacoplar funcionalidades de red en ambientes separados, dividiendo el rol tradicional de los ISP (proveedores de servidos de internet) en dos: proveedor de infraestructura (InPs), quien será el encargado de administrar la infraestructura física, y proveedor de servicios (SPs), quien creará redes virtuales agregando recursos a partir de múltiples proveedores y ofrecer servicios extremo a extremo.

Básicamente, una red virtualizada es un ambiente de red que permite a múltiples SPs formar dinámicamente múltiples redes virtuales heterogéneas que coexistan una a otras de manera aislada.

Una VM puede ser usada para producción, sin embargo, hace difícil mantenerla segura. Por lo tanto, muchas investigaciones se han desarrollado en pro de prevenir ataques contra VM ofreciendo monitorización dedicada a hosts virtualizados [22] compartiendo un hypervisor similar, el monitoreo no solo incluye IDS (sistema de detección de intrusos), sino que también puede integrar servicios como chequeo, sistemas *honeypot*, análisis forense, entre otros.

2.9 Middleboxes

Según [15] fue un término creado por Lixia Zhang como un fenómeno particular el cual estaba surgiendo en internet. Middlebox es definido como un dispositivo intermediario que realiza funciones fuera de lo normal en comparación de un Router IP que enruta paquetes compuestos de datagramas desde una fuente a un destino.

La palabra intermediario normalmente es usada en tráfico HTTP del cual un middlebox puede ser una VFN o VM (Virtual Machine) que realice alguna función de red. Puede ser un dispositivo que termine un flujo de paquete IP y posteriormente origine otro, o ser un dispositivo que transforme diversos flujos IP en otro. Las funciones de enrutamiento estándar (ej. Router) no son consideradas funciones middlebox, ya que las funciones de un Router son transparentes para los paquetes IP, es decir, que un Router tiene como función determinar la ruta del paquete y reenviarlo (mientras en paralelo actualiza campos necesarios para el proceso de reenvío). Es por eso que no son clasificados como middlebox.

Hoy el modelo de extremo a extremo (modelo *Hourglass*) sufre desviaciones causadas por inclusión de numerosas middlebox en la red que realizan otras funciones distintas a la de enrutar paquetes IP. Visto de esta manera, las middlebox desafían la transparencia de la capa de red[20], visto de otra manera, las middlebox desafían las comunicaciones extremo a extremo diluyendo este significado como una característica necesaria para todas las sesiones de comunicación. Además, desafían a los antiguos protocolos ya existentes, ya que estos protocolos fueros diseñados sin necesidad de usar middlebox y en algunas partes de la sesión podrían verse afectados por la presencia de middlebox.

[20] https://tools.ietf.org/html/rfc2275

2.9.1 Tipos de middlebox

Actualmente puede existir gran variedad de middlebox que realicen funciones fuera de los mencionados a continuación, sin embargo los más comunes son descritos en los siguientes apartados.

NAT

NAT (*Network Address Translator*)[21] [22] [23] es una función hecha a menudo en el Router que asigna dinámicamente una dirección pública a un host de una red privada sin que la dirección del host sea conocida. Un NAT es una práctica que viene siempre acompañada por un ALG (*Application Level Gateway*), ALG es una aplicación que permite a una red con IPv6 comunicarse con una red con IPv4 y viceversa, aunque en [24] resaltan esto como un característica en el Gateway que habilita parsear carga útil en la capa de aplicación para tomar decisiones sobre ella, pero también existe aplicativos de profunda incepción donde el Gateway analiza tráfico en la capa de aplicación, por lo que esto también incluye a la capa de transporte hasta el punto de retocar el *checksum*.

NAT-PT

A diferencia del NAT común, NAT-PT[25] normalmente es hecha en el Router, que realiza NAT entre una dirección IPv6 y una red con dirección IPv4, adicionalmente en esta traslación también incluye un formato cabecera entre redes IPv6 e IPv4.

Sockets Gateway

Socks Gateway [26] es un mecanismo con estado para autenticación firewall el cual el cliente debe comunicarse primero con el *SOCK server* del firewall antes que el cliente sea capaz de recorrer el firewall, es el *SOCK server* que determina el origen y numero de puerto usado para salir del firewall, es similar al NAT, sin embargo Socks Gateway no requiere de ALG.

IP Tunnel extremo a extremo

Corresponde a realizar conexiones virtuales entre un extremo a otro extremo de la sesión, utiliza los servicios básico de IP para configurar túneles de un punto a otro los cuales podrían estar situados en diferentes partes del mundo. Tener un túnel hace que se cree una nueva conexión virtual bajo la infraestructura de internet, de este modo se abre un sin número de nuevos servicios haciendo que los paquetes lleguen a su destino intactos y tal cual como fueron enviados por lo que sigue el principio de extremo a

[21] https://tools.ietf.org/html/rfc2663
[22] https://tools.ietf.org/html/rfc2993
[23] https://tools.ietf.org/html/rfc3022
[24] https://kb.juniper.net/InfoCenter/index?page=content&id=KB13530&actp=METADATA
[25] https://www.ietf.org/rfc/rfc2766.txt
[26] https://www.ietf.org/rfc/rfc1928.txt

extremo donde nadie de manera sospechosa interfiere en la sesión. Sin embargo en caso que falle en el túnel esto generaría la perdida de comunicación entre los participantes de la sesión causando por consiguiente que la sesión sea abortada, sin embargo en algunos casos cuando el problema no va a mayores, IP tunnel tiene la capacidad de reestablecer la sesión de forma automática.

PEP TCP

Un PEP (*Performance Enhancing Proxy*) es usado para mejorar el rendimiento de los protocolos de internet sobre redes de comunicación. Existen muchos tipos de PEP que realizan diferentes características que afectan TCP. PEP no necesariamente tiene que ser limitado a mejorar el rendimiento de la red ya que puede ser relacionado a otro tipo de rendimiento como por ejemplo [17], que aborda el tema de mantener las conexiones TCP vivas durante periodos de desconexión en redes inalámbricas.

PEP puede ser implementado en cualquier capa del protocolo TCP y la aplicabilidad depende de ambiente en que se use, por ejemplo, en la capa de transporte PEP puede ser usado en ambientes donde los ACK se agrupen y causen segmentos de datos indeseados, un PEP puede desarrollarse con la capacidad de modificar el espacio de los ACK mejorando el rendimiento de la red. En la capa de aplicación, PEP puede ser usado como cache o MTA para intentar mejorar el rendimiento de los servicios ofrecidos.

Como lo indica [16], PEP debería ser usado solo para ambientes específicos y en circunstancia donde el concepto de extremo a extremo (*end-to-end*) no sea necesario de implementar. En el caso que PEP sea utilizado en el medio, el usuario o el administrador de la red deberían ser conscientes de la existencia de PEP y acordar las funciones que aportaría como middlebox. Estableciendo que el control de la aplicación es hecha por parte del usuario.

Balanceadores de carga

Un balanceador de carga [18] es un dispositivo que actúa como aplicación de tráfico a través de un número de servidores. Son comúnmente utilizados para incrementar la capacidad (concurrencia de usuarios) y rentabilidad de aplicaciones mejorando la sobrecarga de las aplicaciones disminuyendo la carga en los servidores.

Los balanceadores de carga están ligados a la capa cuatro y siete. En la capa cuatro los balanceadores actúan sobre datos encontrados en la red que sean encapsulados sobre IP, TCP, FTP, UDP. En la capa siete, los balanceadores distribuyen las peticiones basadas sobre datos encontrados, como por ejemplo en las cabeceras HTTP, cookies, etc.

Los balanceadores aseguran rentabilidad y disponibilidad, monitoreando las aplicaciones y solo enviando peticiones a servicios que pueden responder de manera oportuna.

IP Firewalls

Es una aplicación que vista de la forma más sencilla puede ser un Router que permita o rechace paquetes basados estrictamente en campos IP o de transporte, por ejemplo, que una de sus funciones sea deshabilitar tráfico entrante para ciertos puertos, deshabilitando cualquier tráfico para ciertas subredes, etc.

Aplicaciones firewall

Es un dispositivo situado entre el cliente y el servidor donde su principal función es analizar los mensajes que tiene como destino la capa de aplicación, dependiendo de la composición del paquete, se aplican las técnicas asociadas para anular o dejar pasar la petición. Otra de sus funciones es analizar y evitar que mensajes indeseados puedan daños en las aplicaciones como servidores web. Por lo tanto, puede ser implementado para proteger las subredes, realizar un chequeo exhaustivo de las peticiones, usar una metodología diseñada para minimizar *bugs*, ejecutarse en ambientes aislados, etc.

Proxies

Según [27] un proxy es un sistema intermedio que actúa entre el cliente y servidor a propósitos de mejorar el rendimiento entre la sesión. Un proxy debe implementar los requerimientos acordados entre el cliente y servidor.

Un proxy transparente es un proxy que no modifica las peticiones que recibe, su única función es dejar pasar la petición sin realizar ningún cambio al paquete.

Un proxy "no" transparente, es un proxy que modifica la petición recibida y le genera un cambio a los paquetes para proporcionar algún servicio añadido a la sesión, por ejemplo, un transformación de protocolo, reducción del protocolo, filtrado, anonimato, *caching*, autenticación de acceso, etc.

Es común que un proxy sea relacionado con un firewall cuando un firewall no permite la salida de trafico HTTP, sin embargo, HTTP hace uso de un proxy de manera voluntaria y conocida, es decir, el cliente debe ser configurado para hacer uso del proxy.

[27] https://tools.ietf.org/html/rfc2616#section-15.7

Caches

Es una técnica común usada en varias áreas de las tecnologías de la información e internet para mejorar rendimiento, escalabilidad, y calidad de servicio con el fin de almacenar temporalmente (*caching*) documentos como HTML e imágenes. Mejora el ancho de banda, reduce la carga en el servidor y el desfase de la comunicación haciendo copia de documentos que pasan a través de él.

Cache forma un rol importante en pro de mejorar en rendimiento web manteniendo objetos web que son visitados con más frecuencia. Ayuda a mejorar la perspectiva de usuario reduciendo latencia. Por ejemplo, reducir el tiempo desde que se hace una petición hasta que se recibe el contenido solicitado. Esto minimiza uso de ancho de banda [23].

En caso de existir una red de servidores cache, el contenido copiado es ofrecido por el servidor cache más cercano al usuario, para esto es necesario un protocolo de interconexión, existe una solución como ICP (*Internet Cache Protocol*)[28] para desplegar soluciones de este tipo sobre internet aunque una alternativa para ICP puede ser HTCP[29].

Anonimato

Esta técnica puede ser implementada de varias maneras con el fin de ocultar la dirección IP de los datos enviados o recibidos. Aunque esta implementación puede ser distinta, esto es una práctica muy similar a NAT o ALG.

2.10 Multi-Context Transport Layer Security

Así como los desarrolladores de HTTP tuvieron que adaptar y definir cómo encriptar datos transmitidos por HTTP a través de TLS, también los desarrolladores de protocolos deben estandarizar sus aplicaciones para que usen McTLS (Multi-Context Transport Layer Security) como protocolo de encriptación; quizás porque el cambio más significativo que traería McTLS sería los contextos, donde la aplicación necesita decidir cuantos contextos debería usar, permisos a establecer y middlebox a permitir.

2.10.1 Casos de uso para Multi-Context TLS

- **Compresión de datos Proxy:** Actualmente Google hace uso de proxy de compresión de datos para mejorar el rendimiento de ancho de banda dentro de un canal de comunicación usando aplicativos desarrollados que se alojan en datacenters de Google, sin embargo como lo indica [32], los datos que son enviados a través de HTTPS por cuestiones de seguridad no es posible

[28] https://www.ietf.org/rfc/rfc2186.txt
[29] https://tools.ietf.org/html/rfc2756

realizar esta acción. Con McTLS los usuarios podrían enseñar a sus navegadores a realizar funciones de compresión de datos a contextos con acceso de escritura a respuestas sobre HTTP. Por lo que en dado caso, el servidor web y el navegador pueden ponerse de acuerdo para que existan dos contextos donde datos con extensión .png, .img o cualquier imagen tenga acceso al proxy de compresión de datos y contenido HTML, CSS u otros no tengan acceso al proxy en cuestión.

- **Filtrado Parental:** Aplicando McTLS a esta función, es posible realizar configuración a dispositivos para permitir que sus filtros tengas acceso de solo lectura a *headers* (cabeceras) solicitadas por HTTP.

Según [4], todos los requisitos que se nombraron en apartados anteriores son abarcados y además han añadido dos características claves a TLS.

Contextos encriptados

Con TLS solo existen dos partes en la sesión (cliente y servidor), por lo tanto este requisito en TLS no tendría sentido. Ya que no tendría lógica restringir el acceso de los datos a un extremo (cliente o servidor). Con la inserción de middlebox de confianza los extremos pueden limitar ese acceso a solo una porción de los datos transmitidos o establecer permisos de solo lectura a uno de ellos. Esto se logra a partir de contextos encriptados o contextos (para TLS). **Un contexto es un conjunto de encriptación simétrica y mensajes de autenticación de código (MAC) que controla quién tiene permisos de lectura y escritura de los datos enviados en un contexto.**

Por otra parte se puede decir que la definición de contexto se relaciona a una sección de datos enviados a través de un canal de comunicación o también puede se le puede relacionar a la configuración de los permisos que tendrá un middlebox. Si esto se lleva a parte práctica, en el caso de HTTP podrían existir varios contextos donde el paquete de *request* puede ser dividido en: *headers* y *body* y el paquete de respuesta puede ser dividido en: *headers* y *body*.

Claves de contextos

El cliente y servidor debe realizar el intercambio de llaves con cada middlebox que acepta como confiable, luego los extremos deben generar la mitad de cada contexto y enviar encriptado con la llave simétrica la mitad de ellos a cada middlebox según el contexto al cual desean que tengan acceso. El middlebox solo tendrá acceso al contexto si recibe ambas partes de la llave, asegurando que el cliente y servidor son conscientes de cada middlebox y de los permisos dados. El servidor puede desistir de este control si así lo desea (para evitar computación extra en la sesión).

Controles de acceso lectura/escritura

Cada contexto tiene su propia llave de encriptación (llamada $K_{readers}$ en McTLS). Los que poseen esta llave les constituye acceso de lectura a la información.

Los accesos de escritura son controlados limitando quién puede generar una MAC válida. McTLS sigue tres propósitos MAC: "extremos de la sesión, lectores, escritores"[4], por lo tanto cada registro McTLS acarrea consigo tres llaves MAC que son $K_{endpoints}$ (compartida por los extremos de la sesión), $K_{writers}$ (compartida por los extremos de la sesión y middlebox con acceso a escritura) y $K_{readers}$ (compartida por extremos de la sesión y middlebox con permisos de escritura y lectura). Asimismo cada contexto tiene su propia llave de lectura ($K_{writers}$) y escritura ($K_{readers}$) pero solo hay una llave ($K_{endpoints}$) en la sesión donde solos los extremos (dispositivos finales) tienen acceso de todos los contextos.

Generación de MAC

Cuando un extremo de la sesión crea un registro, este incluye tres MAC con su correspondiente llave, cuando un middlebox con permiso de escritura modifica un registro inmediatamente genera una nueva MAC con $K_{writers}$ y $K_{readers}$ y posteriormente envía la llave ($K_{endpoints}$) MAC sin modificaciones, cuando un middlebox con acceso a lectura envía un registro, no modifica ninguna de las llaves MAC.

Chequeo de MAC

Cuando un extremo de la sesión recibe un registro, chequea la $K_{writers}$ MAC para confirmar que no se realizaron modificaciones, en estos chequeos la $K_{endpoints}$ MAC descubre si existió alguna modificación sospechosa. Cuando un middlebox con acceso a escritura recibe un registro, chequea la $K_{writers}$ MAC que no se realizaron modificaciones sospechosas en ella. Cuando un middlebox con acceso a lectura recibe un registro, este usa la $K_{readers}$ MAC para chequear si algún atacante o elemento extraño de la sesión ha realizado alguna modificación en el contexto.

Con el esquema de tres MAC, los middlebox con acceso a lectura no pueden detectar cambios realizados por otro middlebox de lectura (suplantado por un atacante). **Este problema es establecido porque una llave compartida no puede ser usada por una entidad para vigilar otra entidad en los mismos niveles de privilegios. Esto es un problema cuando hay dos o más middlebox de lectura para un contexto, es decir. Un middlebox de lectura no detecta si otro middlebox de lectura ha realizado algún cambio al contexto recibido. Sin embargo esto no es un problema ya que cuando llegue a un salto donde haya un middlebox con acceso de escritura podrá detectar el cambio realizado.** Por lo tanto en [4] proponen varias soluciones para fijar este problema, una de ellas es que los middlebox de lectura, escritura y

extremos de la sesión compartan sus llaves simétricas; los middlebox de escritura y extremos de la sesión creen y añadan una MAC por cada middlebox de lectura existente, la segunda solución es que los extremos de la sesión y middlebox de escritura generen una firma digital en lugar de MAC, diferente a las llaves $K_{writers}$ MAC, los middlebox de lectura podrían verificar estas firmas.

Handshake

En [4] realizan una explicación del handshake que ofrece McTLS donde muestran las diferencias más destacadas con TLS mostrando que tiene la misma forma, incluso manteniendo los 2-RTT que conserva TLS.

1. *Setup*: cada componente de la sesión genera valores randómicos y un par de llaves usando DH (*Diffie-Hellman*); los middlebox generan dos pares de llaves. Una para el cliente y otra para el servidor. Asimismo los extremos también generan sus valores secretos.
2. *Client hello*: Así como en TLS, en McTLS se inicia la sesión con un mensaje denominado *Client hello* compuesto por valores randómicos. Además, este mensaje tiene consigo extensiones McTLS denominadas *"MiddleboxListExtension"* que contiene la lista de los middlebox que fueron incluidos en la sesión, la lista de contextos encriptados y los permisos de acceso de todos los middlebox sobre los contextos. Por lo tanto, el cliente establece comunicación TCP hacia el middlebox y posteriormente el middlebox reenvía el mensaje *Client hello* hacia el servidor o hacia el siguiente salto.
3. *Certificate & Public Key Exchange*: Tal como en TLS, el servidor responde con mensajes compuestos de valores randómicos, certificado y una llave pública dentro del certificado. El middlebox hace lo mismo enviando los mismos componentes al servidor y cliente, el cliente envía una llave pública al middlebox el cual la guarda y la reenvía al servidor. El middlebox engloba estos mensajes sobre mensajes denominados *ServerKeyExchange* y *ClientKeyExchange*; asimismo usa diferentes par de llaves para el cliente y servidor y así prevenir ataques [31]
4. *Shared Key Computation:* El cliente computa dos valores secretos con ayuda del servidor y el(los) middlebox, que a su vez son usados para generar una llave simétrica compartida con el middlebox y servidor. Por otra parte, el cliente genera llaves parciales con valores secretos que solo el cliente conoce para los contextos con permiso de lectura y/o escritura. El servidor realiza los mismos procedimientos del cliente [4]. Cuando el middlebox recibe el mensaje de *ClientKeyExchange*, él mismo computa las llaves públicas compartidas con el cliente y servidor. Estas llaves serán usadas después para des-encriptar los contextos compartidos del cliente y servidor.

5. *Context Key Exchange:* El cliente envía las llaves parciales de los contextos hacia el middlebox, el middlebox puede recibir diferentes llaves dependiendo de los permisos que se la haya concedido por parte del cliente, estos mensajes son enviados de manera encriptada y autenticada bajo las llaves compartidas del cliente, el servidor y el middlebox.
6. *Context Key Computation:* Este proceso tiene como propósito asegurar que un middlebox solo tenga acceso a contextos que el cliente y servidor hayan permitido.
7. *Finished:* El Handshake culmina con el intercambio de mensajes *Finished* (así como en TLS).

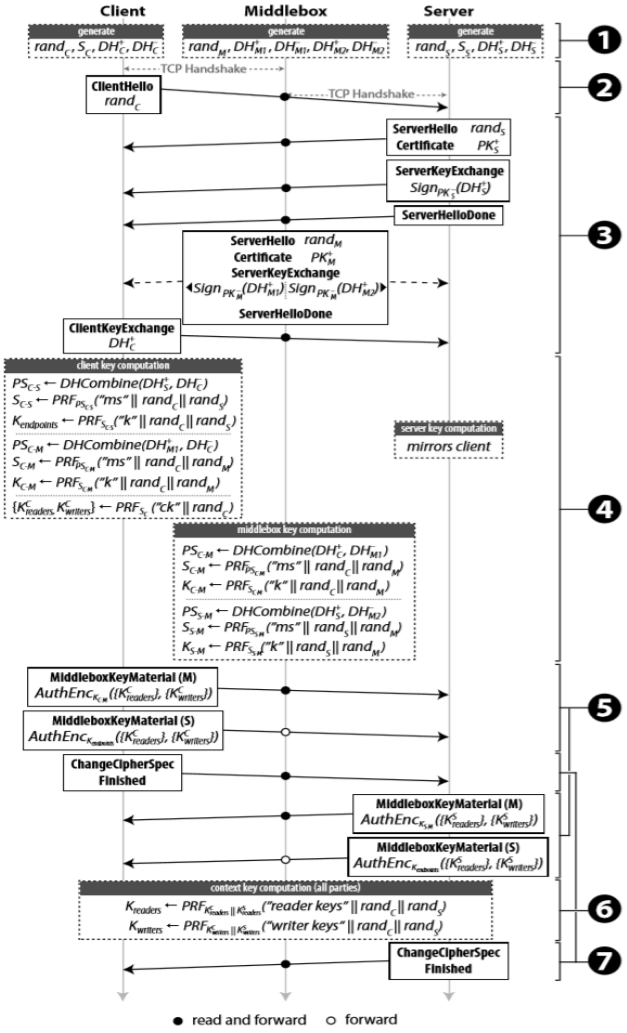

Figura 13. Handshake McTLS.

Existen algunas diferencias entre el handshake de TLS y McTLS. Sin embargo en [4] resaltan que estas diferencias no perjudican la seguridad del handshake de McTLS.

- Por cuestiones de seguridad los middlebox no pueden negociar parámetros como: *cipher siute* o número de contextos permitidos.
- El contexto del servidor no es incluido en los mensajes *Finished* del cliente, ya que requeriría un RTT extra, sin embargo, estos contextos son enviados de forma encriptada.
- El cliente no puede des-encriptar los contextos que el servidor y el middlebox envía y viceversa. Según [4] esto requeriría establecer una llave simétrica a tres vías entre los extremos y cada middlebox existente en la sesión, ya que el middlebox necesita las claves de cada uno de los extremos para realizar permisos sobre cada contexto.
- El middlebox no puede revisar el *hash* del mensaje *Finished* en el handshake porque no conoce las $K_{endpoints}$ de los extremos. Para los desarrolladores de McTLS esto no era necesario ya que así evitaban sobrecarga innecesaria en el protocolo [4]. Esto conlleva a que se considere un ataque de denegación de servicio porque los middlebox observarían una secuencia de mensajes incongruentes en el *handshake*. Sin embargo. Incluso aunque el middlebox no sea capaz de detectar el ataque. Los extremos sí podrían detectarlo y terminarían la sesión, además el contenido va encriptado bajos las llaves que los extremos compartieron con los middlebox, dando como resultado que ninguna fuente desconocida pueda ver o modificar el contenido de los contextos enviados en la comunicación.

2.11 Desarrollo

Como se citó en secciones anteriores. Los middlebox son aplicaciones que están situadas en medio de la comunicación entre cliente y servidor donde pueden existir uno o más aplicativos embebidos dentro de VFNs denominadas middlebox con el fin de manipular o monitorizar el flujo de datos.

El enfoque dado en esta investigación está destinada a nivel de aplicación del middlebox donde también es conocido como proxies**. El cual es definido como una aplicación que accede a los datos para realizar acciones como, sistemas de detección de intrusos (IDSec), filtro de contenidos para niños, caching de datos, compresión de información (tarea comúnmente hecha por proxies) y/o proveer escaneo de virus para empresas.**

	Request		Response	
	Headers	*Body*	*Headers*	*Body*
Cache	○		●	●
Compression			●	●
Load Balancer	○			
IDS	○	○	○	○
Parental Filter	○			
Tracker Blocker	●		●	
Packet Pacer			○	
WAN Optimizer	○	○	○	○

(● = read/write; ○ = read-only)

Figura 14. Patrones de permisos (lectura/escritura) en capa de aplicación para middleboxs

A medida que el mundo de las comunicaciones fue avanzado y desarrollándose se han encontrado diversidad de herramientas útiles que proveen funcionalidades de red que mejoran la transmisión extremo a extremo dando como resultado ayudar a los usuarios, proveedores de servicios y operadores de red a aumentar el procesamiento y almacenamiento de datos en red. Funciones como cacheo o compresión de datos ayudan a mejorar el *delay* de la red hacia los usuarios [26][27], reducir el uso de datos para operadores de red y usuarios [28][29], además de reducir consumo de energía en los dispositivos de lado cliente [29][30], al mismo tiempo que una función como cacheo de datos mejora sustancialmente la efectividad en la red.

La idea de desarrollar middlebox con TLS viene de las propiedades que TLS puede proporcionar como protocolo. Ya que otorga autenticación, seguridad e incluye integridad en los datos transmitidos, por tal motivo se ha diseñado un nuevo protocolo llamado McTLS (Multi-Context TLS), así que es necesario introducir las ideas claves y describir los componentes del protocolo. [30]

2.12 Requisitos del protocolo Multi-Context Transport Layer Security

El requisito fundamental de este protocolo es mantener las propiedades que el estándar TLS otorga, es decir:

1 Autenticación: los destinos finales deben tener la capacidad de autenticarse a todos los middlebox. Semejante a TLS, es necesarios que los clientes en todas las entidades existan en la sesión, sin embargo los servidores no deben establecer este ítem como obligatorio. (ej. para reducir sobrecarga).

2 Seguridad en los datos: solo los extremos y middlebox confiables pueden tener acceso (lectura/escritura) a los datos.

[30] https://mctls.org

3 Integridad de los datos: todos los miembros de la sesión deberán tener la capacidad de detectar modificaciones o alteraciones hechas por terceras partes, además, los extremos de la sesión deben examinar si los datos fueron originados por otro extremo de la sesión (middlebox de confianza, servidor…).

La inserción de middlebox trae consigo dos nuevos requisitos como:

4 Visualización y control explícito: El protocolo debe asegurar que los middlebox de confianza son añadidos a la sesión con consentimiento del cliente

5 Privilegios: los middlebox deberán tener el mínimo nivel de acceso requerido para realizar sus tareas, deben tener acceso a una porción de datos relevante para ejecutar sus funciones, por ejemplo, si alguna ejecución no requiere de modificación de los datos, entonces los permisos deberán ser de solo lectura.

Además de estos requisitos, el protocolo también debe reunir funciones substanciales como sobrecarga, latencia, computación, estado de conexión, sobrecarga a nivel de usuario, etc.

2.13 Valoración de ideas

En este apartado de la memoria, el desarrollador quiso definir middlebox como un proxy para tener una idea general de lo que iba a implementar. Como primer requisito para realizar un proxy que soportara McTLS en entornos reales, debía hallar la manera de integrar TLS en algún punto de la sesión por temas de compatibilidad, ya que en ambientes reales McTLS no estaría implementado.

Uno de las formas que estaba construido este protocolo tenía establecido que esta aplicación sólo tenía acceso a los contextos si recibía ambas mitades de las *keys* de cada extremo de la sesión sin olvidar que toda la comunicación sería encriptada.

Middlebox puede leer y modificar (si así lo establece el usuario) los datos transmitidos/recibidos sobre TLS, para esto, McTLS debe tener las mismas propiedades que TLS y así tomar los datos de las capas superiores (aplicación).

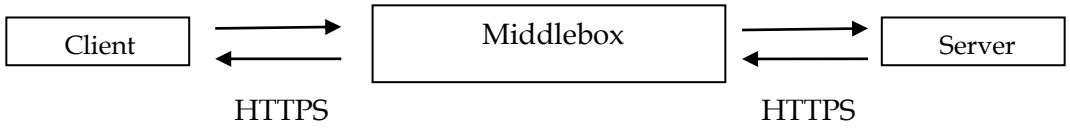

Figura 15. Estructura deseada

Para llegar a la estructura como se muestra en la anterior figura, primero se debía definir en cual punto de la sesión se debía aplicar TLS o McTLS, también era necesario idear varias hipótesis como:

- **¿El cliente debe ser compatible TLS o McTLS?**
- **¿El servidor debe escuchar y ser compatible a través de TLS o McTLS?**

Estas dos preguntas se establecieron como punto de partida para el desarrollo de este aplicativo. Con estas dos preguntas se generaron dos diseños, el primero constaba que el cliente fuera compatible con TLS, seguido de que se creara un proxy inverso que intercambiara la sesión de TLS a McTLS y por último que el servidor web escuchara a través de McTLS.

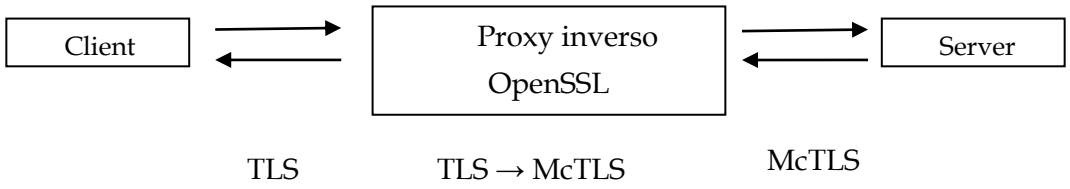

Figura 16. Diseño 1

Debido a que la finalidad del aplicativo era que se simulara como si fuese un entorno real, se concluyó que el Diseño 1 no satisfacía estos requisitos, ya que el servidor (Google, Facebook, Amazon, etc.) debían implementar una nueva capa de encriptación para los datos del cual era inviable para las grandes industrias tecnológicas que comandan el mercado, por tal motivo fue descartada esta alternativa.

El segundo diseño basado en las preguntas, consistía en establecer una sesión donde el cliente hablara McTLS, que usara un proxy inverso donde intercambiara la sesión de McTLS a TLS y posteriormente el servidor web fuese un aplicativo que escuchara a través de TLS.

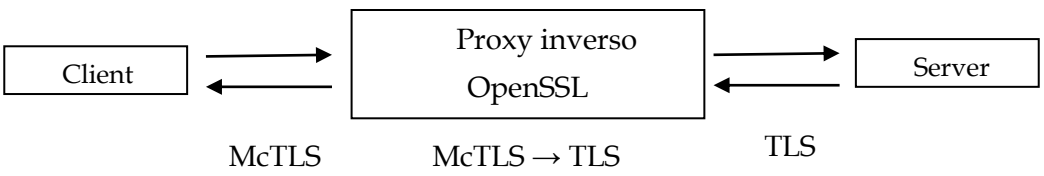

Figura 17. Diseño 2

El segundo diseño se veía con buenos ojos ya que el servidor era transparente al tipo de comunicación/petición que recibía desde el cliente, así el servidor no tendría que cambiar el tipo de capa de encriptación por el cual escucha, y las transferencias continuarían siendo cifradas.

2.14 Análisis de componentes Multi-Context Transport Layer Security

En el capítulo anterior se valoraron algunas ideas para desarrollar un entorno aceptable y compatible al nuevo protocolo McTLS e integrarlo en un ambiente de pruebas que se acercara a un entorno real. Se optó por la idea que se muestra en Figura 17. Diseño 2.

Para empezar con el desarrollo, era necesario conocer y analizar el funcionamiento de cada elemento perteneciente a McTLS, es decir, Cliente McTLS, Middlebox (Proxy) y servidor.

2.14.1 Cliente McTLS

Para analizar el funcionamiento de un cliente McTLS, se ha considerado detallar el código proporcionado por [33]. En este estudio, las abreviaturas SPP son relacionadas con las estructuras creadas a partir de McTLS.

De estos estudios se evidenció que el cliente realiza los siguientes procedimientos para iniciar una conexión mediante McTLS:

1. Generar certificados: el cliente debe obtener varios requisitos de certificados, como son keyfile (cliente.pem), una CA(root.pem), una firma creada a partir del algoritmo DH (dh1024.pem) y un valor radómico (random.pem).
2. Ajustar la cantidad de proxies a usar: El cliente establece la cantidad de proxies que participan en la sesión.
3. Inicializa un contexto: estos contextos en SPP se crear a partir de un método llamado SPP_method().
4. A partir del contexto se crea una estructura SSL: La estructura SSL es establecida por medio de SSL_new()
5. Crear una estructura SPP para cada proxie: esta función es creada a partir de SPP_generate_proxy().
6. Adaptar la cantidad de slices que se generan: Dividir los datos en pequeños trozos para luego seleccionar cual tipo de información es legible o manipulable, y cuáles no.

7. Crear los slices sobre SPP: Se crean los slices a partir de una estructura llamada SPP_generate_slice().
8. Asignar lectura o escritura a los slices: existen dos funciones, SPP_assign_proxy_read_slice() y SPP_assing_proxy_write_slice(). La primera tiene la capacidad de leer los datos transmitidos y la segunda para escribir datos sobre una estructura previamente creada.
9. Implementar un socket TCP para iniciar conexión con el middlebox: La forma con la que crea un socket TCP es a partir de la función socket().
10. Realizar conexión entre sockets, es decir, conectar un socket con otro: Usa la función connect() para realizar esta tarea.
11. Crear una conexión SPP y adjuntarla al socket: Inicia una conexión a través de SPP (McTLS) por medio de la función SPP_connect().
12. Hace un chequeo para comprobar relación del CN (Common Name) con el hostname del cliente: esta comprobación es realizada por medio de la función SSL_get_verify_result().
13. Generan funciones que preparan una petición http GET: cuando el formato GET es creado, se incluye la cabecera a formato SPP con SSP_write_record().
14. Envía y recibe datos: Los datos son enviados a partir de SPP_write_record() y son recibidos, los datos pueden ser leídos mediante la función SPP_read_record().

2.14.2 Middlebox McTLS

El desarrollo de este middlebox era compatible con dos capas de encriptación, McTLS Y TLS, sin embargo, era incompatible para los siguientes casos:

1. Cuando el cliente iniciaba la sesión por TLS, el middlebox estaba escuchando por McTLS y el servidor lo hacía a través de TLS.

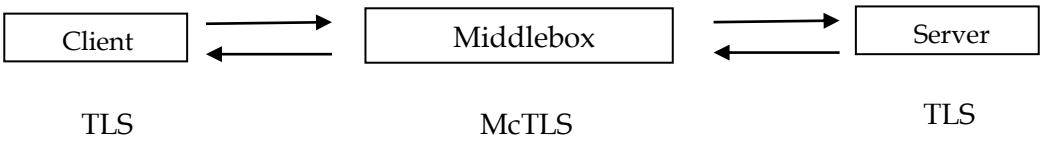

Figura 18. Sesión 1.

2. Cuando el cliente iniciaba la sesión por TLS, el middlebox estaba escuchando por McTLS y el servidor lo hacía a través McTLS.

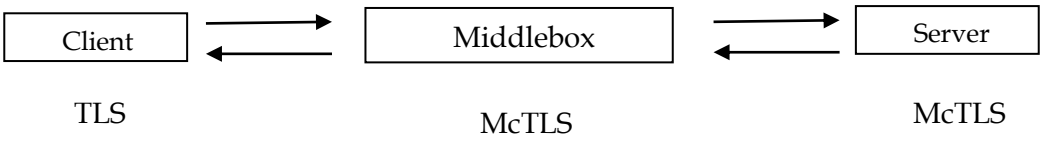

Figura 19. Sesión 2.

3. Cuando el cliente iniciaba la sesión por McTLS, el middlebox estaba escuchando por TLS y el servidor lo hacía a través McTLS.

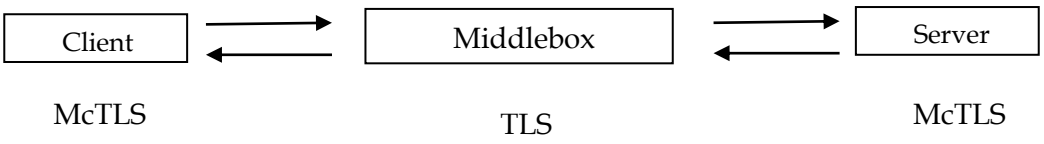

Figura 20. Sesión 3.

4. Cuando el cliente iniciaba la sesión por McTLS, el middlebox estaba escuchando por TLS y el servidor lo hacía a través TLS.

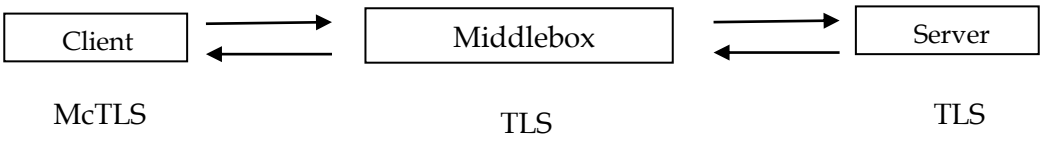

Figura 21. Sesión 4.

Cuatro pruebas fueron suficientes para concluir que la sesión no sería exitosa si las partes involucradas no acordaban una misma capa de encriptación durante la sesión.

Realizada estas pruebas, se fue centrando la idea al funcionamiento del middlebox para cuando estuviera en modo McTLS, de los análisis, se descubrieron los siguientes resultados de su funcionamiento:

1. Se crean certificados: Como el cliente, el proxy debe tener una keyfile (server.pem), una firma creada a partir del algoritmo DH (dh1024.pem) y un valor randómico (random.pem).
2. El proxy por defecto abre una conexión mediante la IP local y el puerto 8423.
3. Se crea conexión encriptada creando un contexto: Este contexto es generado a partir del método SPP_proxy_method(), este método solo puede ser usado para el(los) middlebox.
4. Cargar los parámetros DH desde un fichero (dh1024.pem)

5. Crear un socket, que escuche conexiones entrantes a través del puerto 8423: Esta función es realizada mediante TCP_connect(), escucha las conexiones entrantes por medio de listen().
6. Aceptar conexiones creadas anteriormente: Esto se crea con la función accept().
7. Crear un socket BIO para añadirlo a una estructura SSL: un socket BIO es posible crearlo a través de la función BIO_new_socket()
8. Crear una estructura SSL para una conexión: Con la función SSL_new() se crea una estructura por medio de un contexto.
9. Conectar el socket BIO, añadirlo a una estructura SSL para que sea posible leer y escribir operaciones TLS/SSL: Es posible realizar esta acción por medio de SSL_set_bio().
10. Aceptar una conexión SPP y enviar los datos recibidos al siguiente salto: Existe una función que acepta conexiones mediante McTLS, los desarrolladores la han nombrado SPP_proxy().
11. Lo siguiente es abrir otra conexión para enviar los datos previamente recibidos al siguiente salto o al destino (servidor): Se realiza un proceso similar que ha sido nombrado en procedimientos anteriores, es decir, se crea una conexión SSL saliente [create_ssl_connection()], se crea y se inicializa un nuevo contexto [initializate_ctx()], se crea una nueva estructura SSL [SSL_new()], se crea un socket TCP y se conecta ese socket al servidor o al siguiente salto [TCP_connect()], se crea otro socket BIO [BIO_new_socket()], finalmente se conecta el socket BIO a una estructura SSL para leer y escribir operaciones TLS/SSL [SSL_set_bio()].
12. Se crea una función para operar y tratar los datos recibidos que luego sean enviados al destino: la función handle_data() manipula los datos recibidos a través de funciones SPP, estas funciones pueden ser para leer, escribir o reenviar datos sin ser manipulados, SPP_read_record() es usada para leer datos recibidos, SPP_write() esa usada para escribir datos (en bytes) a una nueva conexión SPP_formard_record() es usada para reenviar los datos al siguiente salto.
13. Finalmente cierra la conexión previamente aceptada.

2.14.3 Servidor McTLS

En vista de continuar con la investigación, se optó por analizar el código que efectuaba de servidor McTLS en [33] y estos fueros los resultados del análisis:

1. Crear certificados: Crear un keyfile (Server.pem), una firma creada a partir del algoritmo DH (dh1024.pem), una CA(root.pem) y un valor randómico (random.pem).

2. Crear e inicializar un contexto a partir de los certificados: usa el método SPP_method(), luego crea un contexto a partir de método creado previamente con SSL_CTX_new().
    a. Establecer una lista de chipers solo para SSL: por medio de la función SSl_CTX_set_chiper_list() se establece una lista de chipers disponibles por OpenSSL.
    b. Carga el certificado server.pem dentro del contexto: mediante SSL_CTX_use_certificate() se carga el certificado dentro de un contexto.
    c. Establecer una contraseña cuando se carga/almacena el certificado (servidor.pem): mediante la función SSL_CTX_set_default_password_cb() se establece la contraseña por defecto al contexto que se ha creado previamente.
    d. Verificar el contexto: Con la función SSL_CTX_load_verify_locations() se verifica el contexto, dentro de esta función se usan variables como la CA (root.pem) y el contexto, si el valor es cero (0) hay algún fallo de verificación, si el valor es uno (1) es que la verificación ha sido exitosa.
3. Cargar los parámetros DH desde un fichero (dh1024.pem)
4. .Crear un socket, que escuche conexiones entrantes: Esta función es realizada mediante TCP_connect(), escucha las conexiones entrantes por medio de listen().
5. Aceptar conexiones creadas anteriormente: Esto se crea con la función accepet().
6. Crear un socket BIO para añadirlo a una estructura SSL: un socket BIO es posible crearlo a través de la función BIO_new_socket().
7. Crear una estructura SSL para una conexión: Con la función SSL_new() se crea una estructura por medio de un contexto.
8. Conectar el socket BIO, añadirlo a una estructura SSL para que sea posible leer y escribir operaciones TLS/SSL: Es posible realizar esta acción por medio de SSL_set_bio().
9. Espera a iniciar el handshake TLS/SSL: El handshake puede ser ejecutado con la función SSL_accept().
10. Empieza la manipulación de lectura y escritura de los datos: Con la función http_serve_request() se establece la lectura y escritura de los datos, usa variables como la estructura SSL que previamente se ha creado, la aceptación de la conexión abierta anteriormente (paso 4) y el tipo de protocolo de encriptación a usar (SPP):
    a. Se realiza lectura de los datos: con SPP_read_record() se establece la lectura de los datos (por slice) recibidos para posteriormente quedar almacenados en un buffer.
11. Envío los datos a través de conexión SPP/SSL: Mediante una función SendData() se retornan los datos al middlebox, esta función está compuesta por

parámetros como una estructura SSL, el buffer de información, el tipo de protocolo, etc.

   a. Mediante SPP_write_record() se escribe el contenido al destino dentro estructura SSL.

12. Cierra conexiones y el proceso acaba.

2.15 Desarrollo de middlebox

En el capítulo anterior se hizo un análisis del funcionamiento de un esquema cliente servidor con un middlebox entre cada componente, de los análisis surgió la idea de lo que se necesitaba para cumplir el objetivo propuesto.

Para completar el objetivo deseado, habría que crear un middlebox que fuese capaz de recibir conexiones McTLS y traducirlas a TLS. Así, en un extremo se tendría al cliente enviando peticiones por una capa de encriptación McTLS y al otro extremo, se tendría un servidor esperando conexiones mediante TLS, para esto primero se analizó las diferencias entre crear un socket con TLS/SSL y un socket usando McTLS.

2.15.1 Socket SSL

1. Crear un socket TCP con la función socket().
2. Conectar el socket al siguiente salto mediante la función connect() usando el puerto 443.
3. Inicializar OpenSSL.
4. Crear una conexión SSL y adjuntarla al socket.
5. Enviar datos mediante la función SSL_write() y recibir los datos mediante la función SSL_read().
6. Cerrar el socket con SSL_shutdown() o SSL_free()

2.15.2 Socket McTLS

1. Ajustar la cantidad de proxies a usar.
2. Generar una estructura SPP para cada proxie mediante la función SPP_generate_proxy().
3. Ajustar la cantidad de slices a generar.
4. Generar los slices sobre SPP mediante SPP_generate_slice().
5. Asignar lectura y/o escritura a los slices mediante SPP_asing_proxy_read_slice() y/o SPP_assing_proxy_write_slice().
6. Crear socket TCP a través de la función socket().
7. Conectar el socket al siguiente salto mediante connect().
8. Crear una conexión SPP y adjuntarla al socket mediante SPP_connect().

9. Enviar los datos con SSP_write_record(), recibir los datos a través de SPP_read_record().

2.15.3 Unión de ideas McTLS/TLS

Teniendo claras las funciones que iba a realizar cada módulo, era necesario tener unos objetivos bien definidos, de los cuales se enumeran a continuación:

1. Es necesario crear un socket que escuche conexiones entrantes a través del puerto 8423.
2. Aceptar esas conexiones entrantes en el socket.
3. Crear un socket BIO a partir del socket previamente creado.
4. Crear una estructura SSL y un contexto SSL para la conexión McTLS.
5. Aceptar una conexión SPP/McTLS.
6. Conectar el socket BIO y permitirle lectura y escritura de operaciones TLS/SSL.
7. Crear una nueva conexión SSL, para lograr esto es necesario:
    a. Crear un nuevo contexto y una nueva estructura en conexiones SSL.
    b. Crear un nuevo socket y conectar el socket al servidor.
    c. Crear un BIO socket a partir del socket anterior.
        i. Conectar el socket BIO, permitirle lectura y escritura de operaciones TLS/SSL.
    d. Iniciar manipulación de los datos
        i. Leer los datos del buffer mediante SSL_read().
        ii. Escribir los datos leídos en el buffer para posteriormente enviarlos con SSL_write().
    e. Finalizar/esperar nuevas conexiones.

Según los parámetros propuestos anteriormente, se realizó un esquema para mayor compresión de la idea que se había planteado.

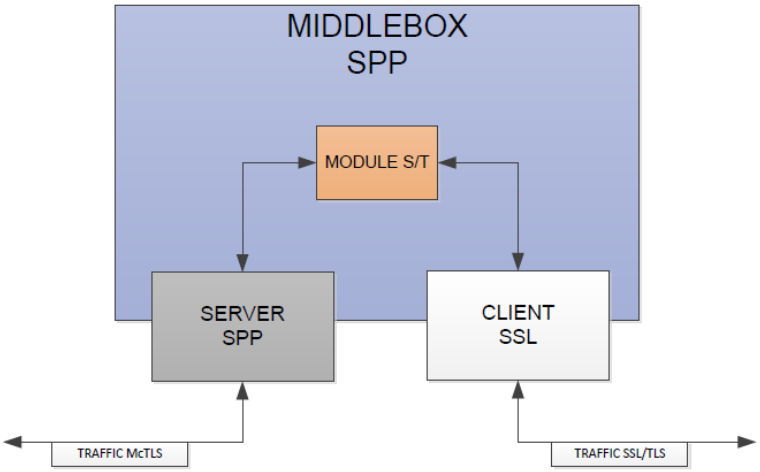

Figura 22. Middlebox propuesto

En la figura anterior se muestra el nuevo módulo que se integró en el desarrollo de McTLS como capa de seguridad para el transporte de los datos, consiste en un módulo llamado ***Molude* S/T** que tiene la capacidad de recibir tráfico encriptado mediante McTLS, luego procesarlo, ajustarlo a estándares TLS, y posteriormente enviar el tráfico mediante una capa de seguridad conocida como TLS.

Aparte, también se analizó el procedimiento para generar una estructura SSL que se adjunta a un socket TCP, en la raíz principal de una estructura SSL está un método que se genera a partir de funciones. En programación, (ej: C++, C) un método TLS puede ser traducido a TLSv2_method(), para McTLS, y SPP_method() para SPP. [31]

Un contexto agrupa variables indispensables que crean una estructura SSL/SPP, un contexto puede contener valores relacionados con certificados, llaves, contraseña y tipos de protocolo a usar (SPP, SSL). En programación, un contexto tiene formato similar a initialize_ctx(KEYFILE,PASSWORD,PROTOCOL)

Debido a que se está trabajando con certificados, es necesario créalos y usarlos según sea el objetivo de uso, a continuación se mostrará algunos ejemplos de cómo se generan certificados que luego son usados en las comunicaciones seguras.

[31] https://www.openssl.org/docs/man1.0.2/ssl/ssl.html

Para crear un *root CA* [8]:

```
$ openssl req -newkey rsa:1024 -sha1 -keyout rootkey.pem -out
rootreq.pem $ openssl x509 -req -in rootreq.pem -sha1 -extfile
myopenssl.cnf \ >  -extensions v3_ca -signkey rootkey.pem -out
rootcert.pem   $ cat rootcert.pem rootkey.pem > root.pem $
openssl x509 -subject -issuer -noout -in root.pem subject=
/C=US/ST=VA/L=Fairfax/O=Zork.org/CN=Root CA issuer=
/C=US/ST=VA/L=Fairfax/O=Zork.org/CN=Root CA
```

```
$ openssl req -newkey rsa:1024 -sha1 -keyout serverCAkey.pem
out \ >  serverCAreq.pem $ openssl x509 -req -in serverCAreq.pem
-sha1 -extfile \ >  myopenssl.cnf -extensions v3_ca -CA root.pem
-CAkey root.pem \ >  -CAcreateserial -out serverCAcert.pem $ cat
serverCAcert.pem serverCAkey.pem rootcert.pem > serverCA.pem $
openssl x509 -subject -issuer -noout -in serverCA.pem subject=
/C=US/ST=VA/L=Fairfax/O=Zork.org/OU=Server Division/CN=Server CA
issuer= /C=US/ST=VA/L=Fairfax/O=Zork.org/CN=Root CA
```

Si se quiere crear el certificado del servidor y firmarlo con la autoridad certificadora (CA) del servidor:

```
$ openssl req -newkey rsa:1024 -sha1 -keyout serverkey.pem out
\ >  serverreq.pem $ openssl x509 -req -in serverreq.pem -sha1 -
extfile myopenssl.cnf \ >  -extensions usr_cert -CA serverCA.pem
-CAkey serverCA.pem \ >  -CAcreateserial -out servercert.pem $
cat servercert.pem serverkey.pem serverCAcert.pem rootcert.pem >
\ >  server.pem $ openssl x509 -subject -issuer -noout -in
server.pem subject=
/C=US/ST=VA/L=Fairfax/O=Zork.org/CN=splat.zork.org issuer=
/C=US/ST=VA/L=Fairfax/O=Zork.org/OU=Server Division/CN=Server CA
```

Para crear el certificado cliente y firmarlo con el *root CA*: [8]

```
  $ openssl req -newkey rsa:1024 -sha1 -keyout clientkey.pem out
\ >  clientreq.pem $ openssl x509 -req -in clientreq.pem -sha1 -
extfile myopenssl.cnf \ >  -extensions usr_cert -CA root.pem -
CAkey root.pem \ >  -CAcreateserial -out clientcert.pem $ cat
clientcert.pem clientkey.pem rootcert.pem > client.pem $ openssl
x509  -subject  -issuer  -noout  -in  client.pem  subject=
/C=US/ST=VA/L=Fairfax/O=Zork.org/CN=shell.zork.org       issuer=
/C=US/ST=VA/L=Fairfax/O=Zork.org/CN=Root CA
```

Para crear dh512.pem y dh1024.pem [8]:

```
  $ openssl dhparam -check -text -5 512 -out dh512.pem

  $ openssl dhparam -check -text -5 1024 -out dh1024.pem
```

A partir de estos procesos generados, se crea una estructura SSL/SPP que servirá para crear una conexión que cifrará los datos. En programación, una estructura SSL/SPP puede ser creada a partir de SSL_new(ctx); donde ctx es el contexto que se ha creado previamente. [32]

Figura 23 Estructura SSL

[32] https://www.openssl.org/docs/man1.0.2/ssl/SSL_new.html

También se descubrió que un socket TLS se componía de varias capas superiores que lo cubrían para generar una comunicación segura, en la siguiente imagen es posible observar que el núcleo de una comunicación es el socket, un socket es una forma de comunicación que se encarga de escuchar conexiones, aceptarlas y conectar (si es el caso) a otro socket, a partir del socket se manipulan funciones para generar una estructura SSL y luego aplicar acciones de lectura o escritura para finalizar con la conexión y realizar el handshake.

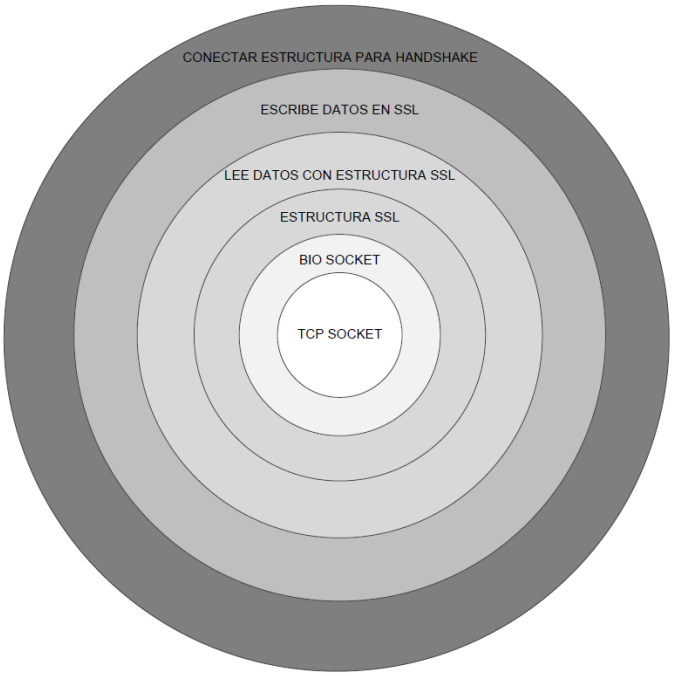

Figura 24. Capas Socket

3 Caso demostrativo y resultados

Con el fin de desarrollar un middlebox que trabajara de forma híbrida entre dos protocolos de seguridad (McTLS y TLS) y ver cómo era posible llegar agrupar un middlebox de otro protocolo de seguridad dentro de las condiciones normales que se manejan hoy en día en internet, mitigando el impacto y los cambios para que éste middlebox fuese adaptable con destinos (servidores) que vayan mediante TLS, fue necesario seguir y entender el código proporcionado por [36] de un prototipo de middlebox que escucha McTLS.

[36] es un prototipo de un escenario que puede escuchar McTLS durante toda la sesión o enlace, también fue adaptado para que cifrara mensajes a través de SSL/TLS y

en texto plano. La limitante era que el middlebox embebido en este prototipo no es capaz de combinar dos protocolos de seguridad en una misma sesión, pero fue una buena base para iniciar la idea con la que fue desarrollado este proyecto.

3.1 Componentes

El caso demostrativo consta de varios componentes que se dividen dentro de toda la comunicación, los componentes varían dependiendo de la cantidad de middlebox que quiere el cliente desplegar cuando inicie sesión con un servidor.

3.1.1 Cliente McTLS

Es el encargado y el responsable de iniciar la sesión cifrada de los datos, este cliente usa McTLS. En este caso, el cliente fue desarrollado en código C para simular un entorno capaz de iniciar una sesión enviando paquetes encriptado por medio de McTLS y enviándolo a un servidor que sólo escuche y cifre datos en TLS/SSL.

El cliente tiene la capacidad de decidir cuantos Middlebox quiere que participen en la sesión y tiene la autoridad de decretar cuál de esos middlebox puede leer, escribir o solo ser usado como *forwarding* de los datos.

3.1.2 Middlebox SPP/TLS

Es la caja mágica que dependiendo de los permisos que le haya asignado el cliente, puede manipular los datos que viajan por la red, es posible de allí sacar provecho de los datos obtenidos desde el cliente (si el cliente permite el acceso a escritura a un(os) middlebox de lectura para realizar prevención de funciones o ataques que viajan por la red y así tomar contramedidas preventivas o de acción).

3.1.3 Servidor TLS

En este caso demostrativo, se llegó a la idea de usar un servidor TLS con un servidor web implementado (Apache) que estuviera sobre una capa de seguridad como TLS donde escucha por el puerto 443.

La siguiente imagen ilustra los tres componentes que fueron implementados en este caso.

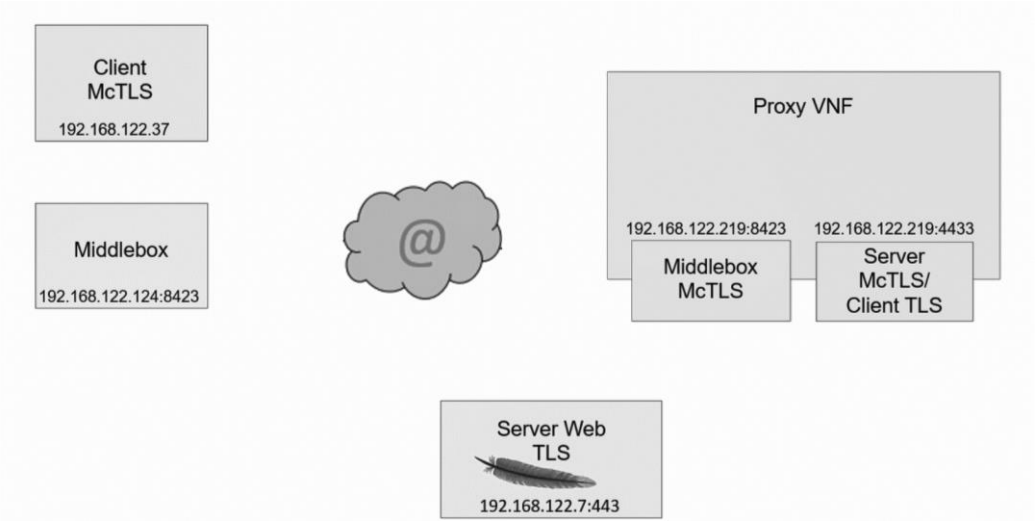

Figura 25. Escenario de caso demostrativo

La idea principal fue simular un entorno donde un cliente (Client McTLS) hiciera una petición web a un servidor web (Server Web TLS), en este caso un servidor Apache, que muestra contenido web a usuarios que la visitan. En el entorno mostrado anteriormente los elementos estaban interconectados en una misma Red LAN, que a efectos de funcionamiento del middlebox no se tendría en cuenta.

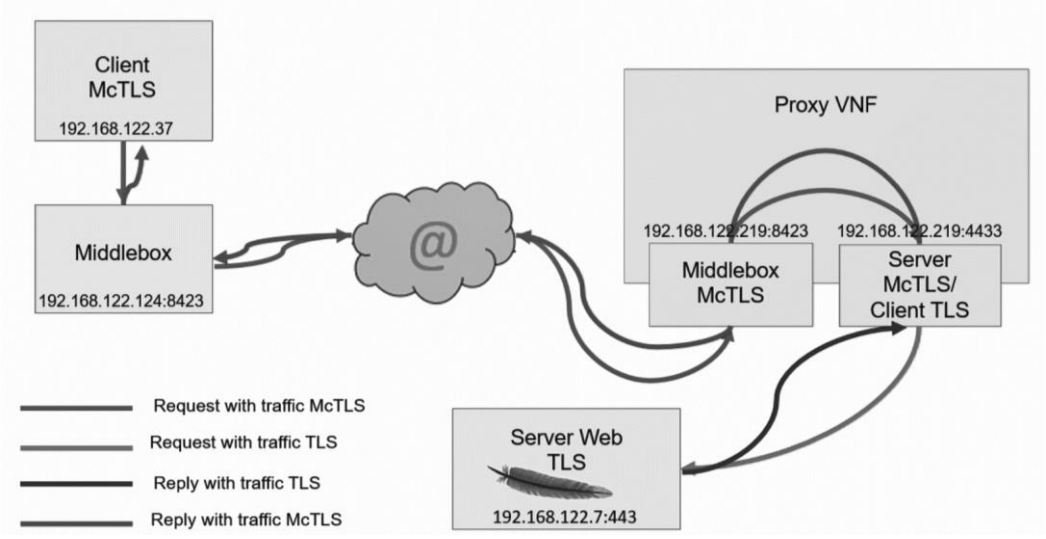

Figura 26. Sesión McTLS entre cliente y servidor.

En el proceso de petición/respuesta los paquetes cambiarían de protocolo de seguridad en el transcurso de que la petición fuese realizada, en este caso, antes de iniciar la comunicación, el cliente (Client McTLS) permite que existan dos middlebox en la sesión.

3.2 Primer caso demostrativo

La primera demostración que se realizó consistía que el cliente hiciera una petición al servidor web, con dos middlebox entre medias, la demostración se basaba en que el cliente permitiera el permiso de lectura y escritura de datos a los middlebox.

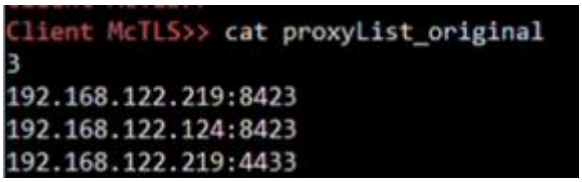

Figura 27. Lista de middlebox.

Figura 28. Petición https desde cliente con permiso de lectura

Con la sentencia anterior, el cliente indicaba (-u) la URL la cual quería establecer una comunicación a través de SPP (-c), con –s a 1 indicaba que iba a realizar un *slice* de los datos, es decir, los datos no se iban a trocear. Con –r a 2 se indicaba el permiso de lectura a los dos middlebox que se encontraban participando en la sesión, con –o a 3, el cliente se establecía a realizar una petición https.

Cuando el cliente hacía la petición, recibía con éxito la respuesta emitida del servidor.

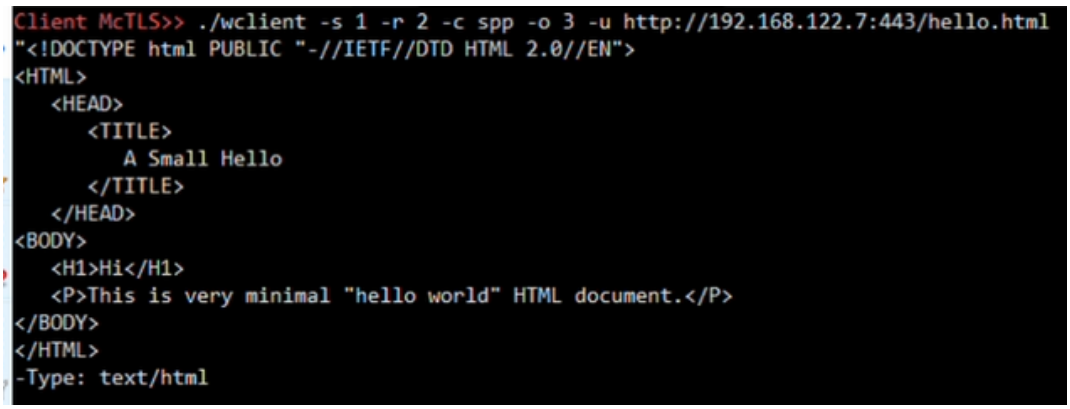

Figura 29. Respuesta del servidor al cliente.

En el otro lado de la sesión, los middlebox eran conscientes de la petición y la respuesta que había recibido el cliente, ya que éste había dado permisos de lectura a los dos middlebox.

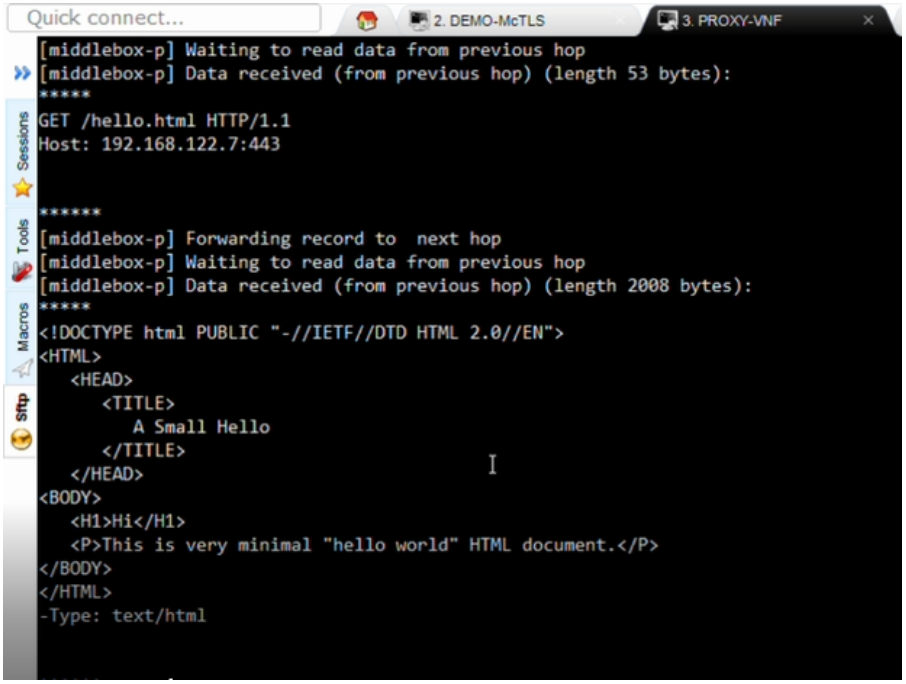

Figura 30. (1) Datos leídos por el proxy VNF

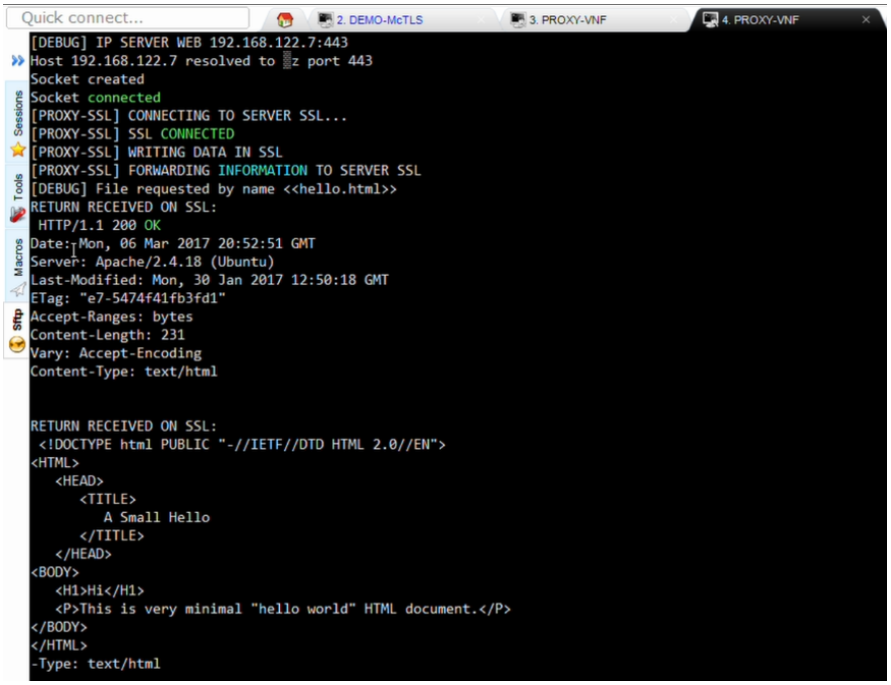

Figura 31. (2) Datos leídos por el proxy VNF

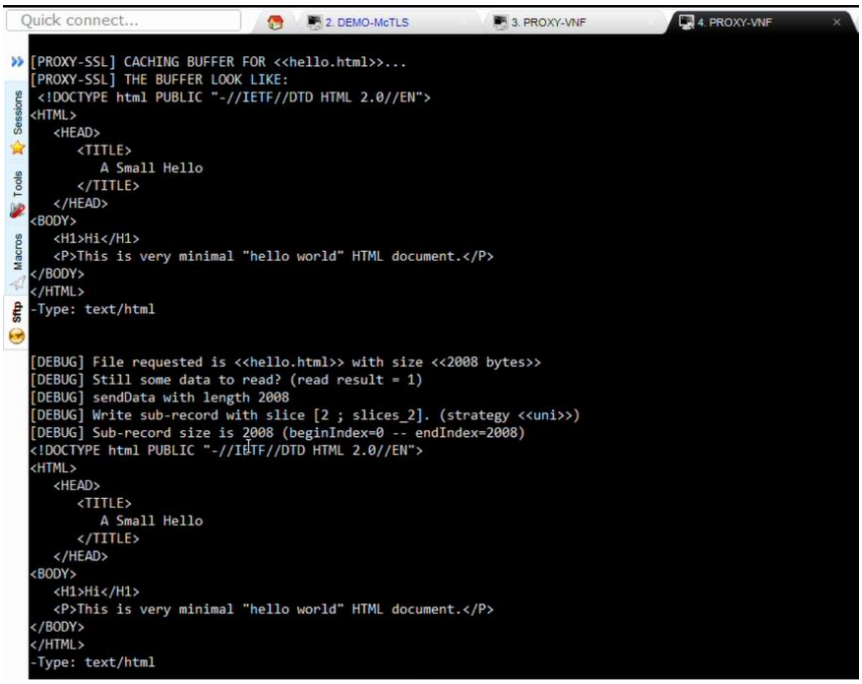

Figura 32. (3) Datos leídos por el proxy VNF

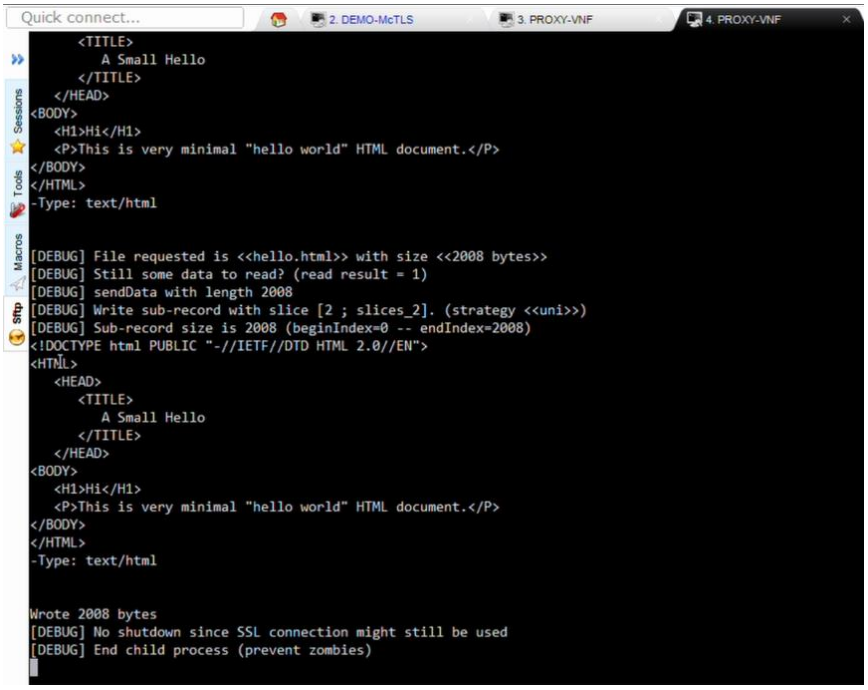

Figura 33. (4) Datos leídos por el proxy VNF

En las imágenes (1-4) de Datos leídos por el proxy VNF que se citaron anteriormente, es posible observar que proxy VNF, recibió y leyó los datos enviados y recibidos del cliente, además realizó la conversión de SPP a SSL para que los datos fuesen transmitidos al servidor mediante TLS, también se observa la petición web que ha realizado el cliente (GET) y la respuesta que ha enviado el servidor.

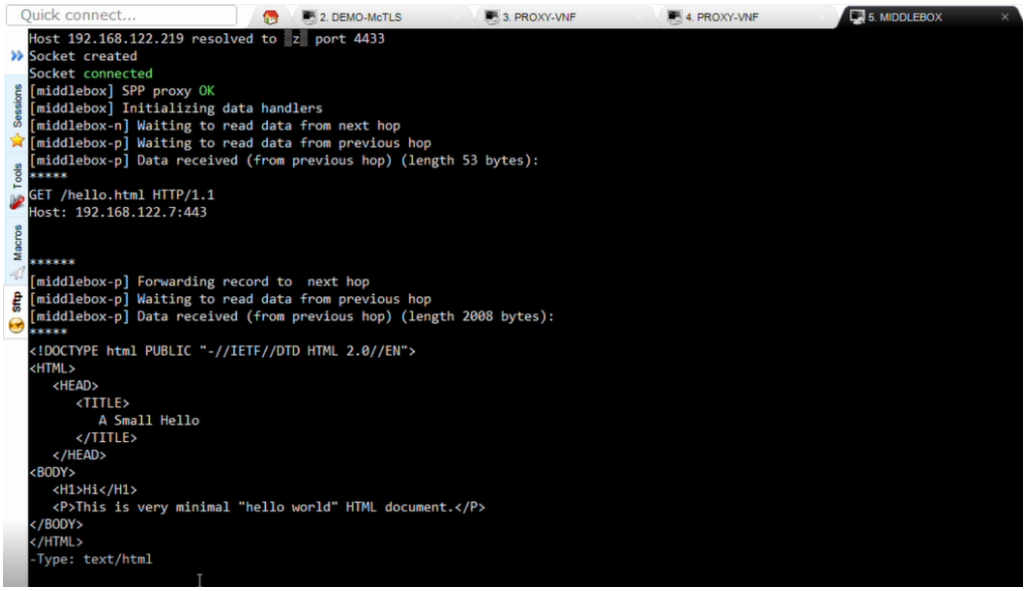

Figura 34. Middlebox 1.

En la imagen anterior se muestra que el primer middlebox que aparece en la sesión McTLS entre cliente y servidor puede también leer los datos enviados y recibidos del cliente.

3.2.1 Datos Wireshark (1)

Para comprobar que los datos fueron enviados de forma encriptada y por puertos seguros, en la misma red LAN se implantó una herramienta llamada Ettercap [37] para simular un ataque *main-in-the-middle* y capturar los paquetes a través de wireshark [38].

En primera instancia, se quiso analizar si los paquetes que se estaban enviando, estaban siendo enviados/recibidos mediante el puerto 80, así se cercioraba que los datos estuviesen viajando por los puertos deseados.

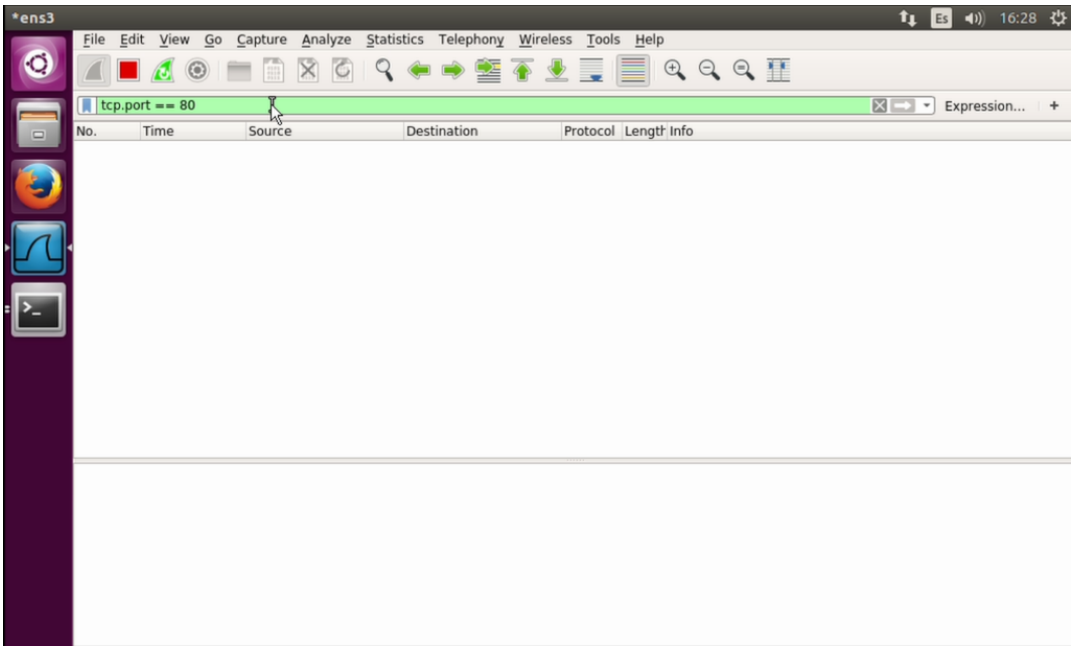

Figura 35. Captura de tráfico puerto 80

Cuando se hizo la captura del tráfico por el puerto 4433 (puerto donde escuchaba el proxy VNF), fue posible ver tráfico de paquetes con sus cabeceras e información que para cualquier atacante podría ser útil, sin embargo, analizando los paquetes con mayor detenimiento se observó que los datos eran ilegibles ya que iban cifrados por el canal.

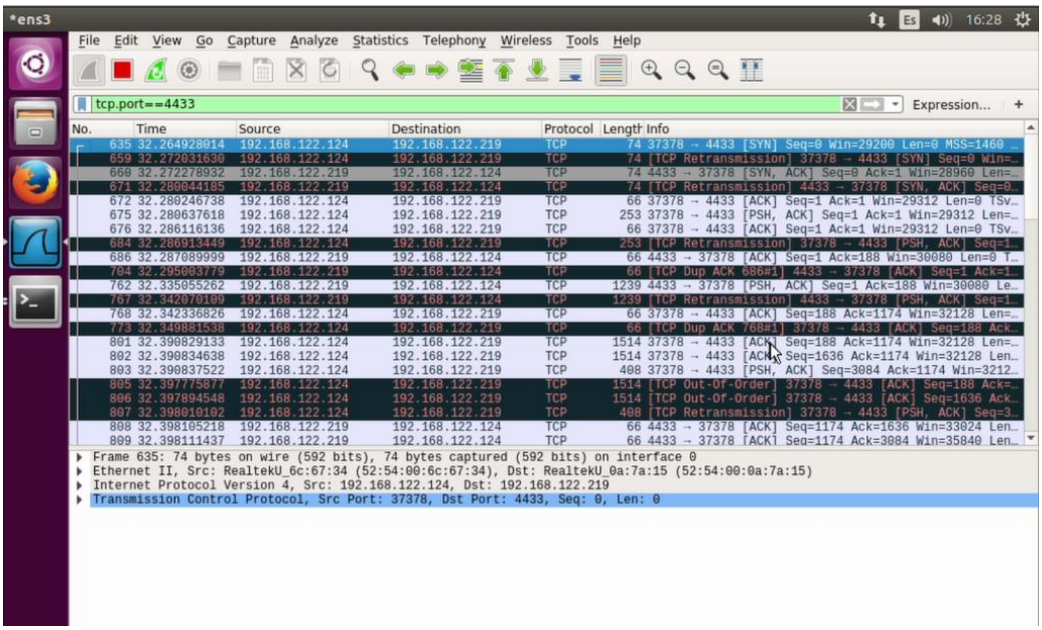

Figura 36. Captura de tráfico puerto 4433

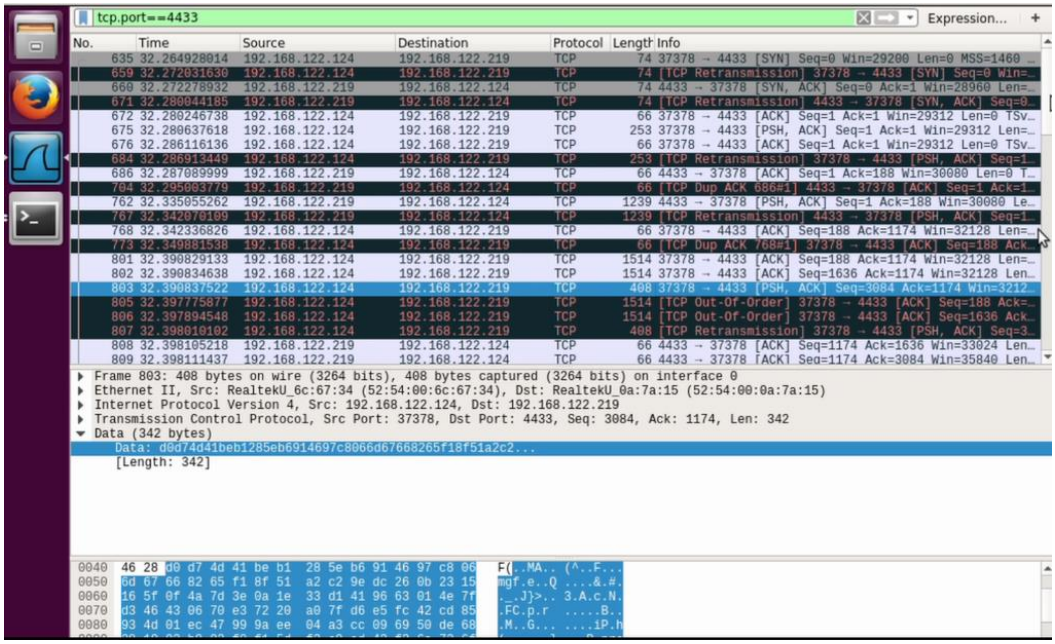

Figura 37. (1) Análisis de paquetes.

Como se ve en la imagen anterior, el número de paquete 803 corresponde al datagrama que contiene la información relacionada con la petición del contenido deseado, en este paquete, vemos que el campo Data tiene una longitud de 342 y los datos que se visualizan son números y no el contenido de la información que fue enviada desde el primer middlebox (que previamente había recibido el contenido del cliente).

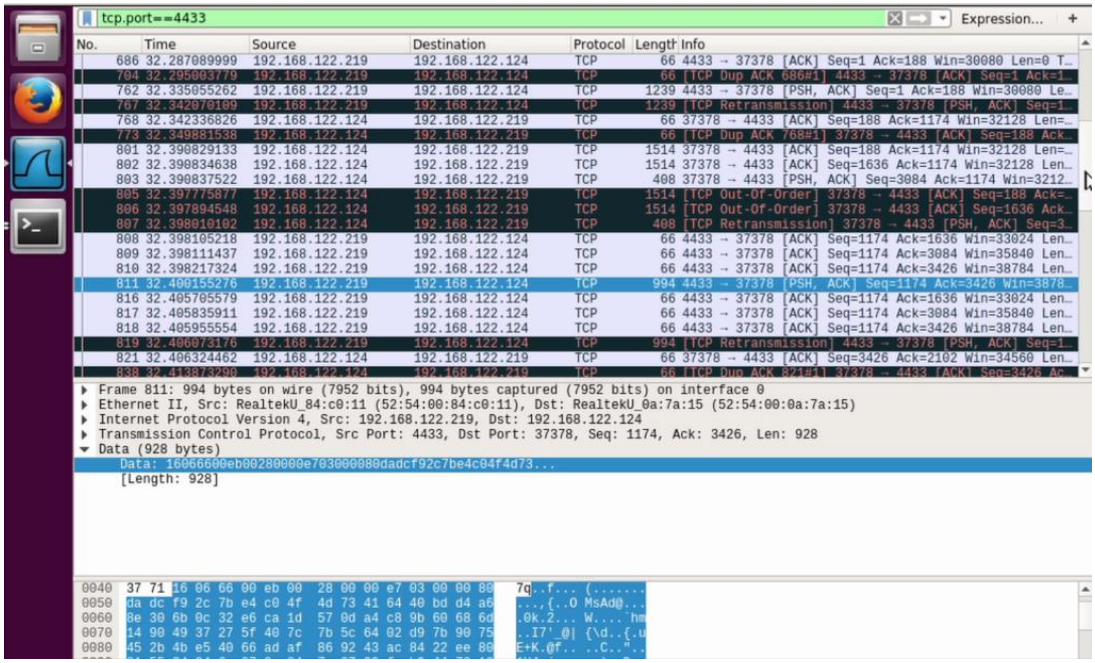

Figura 38. (2) Análisis de paquetes.

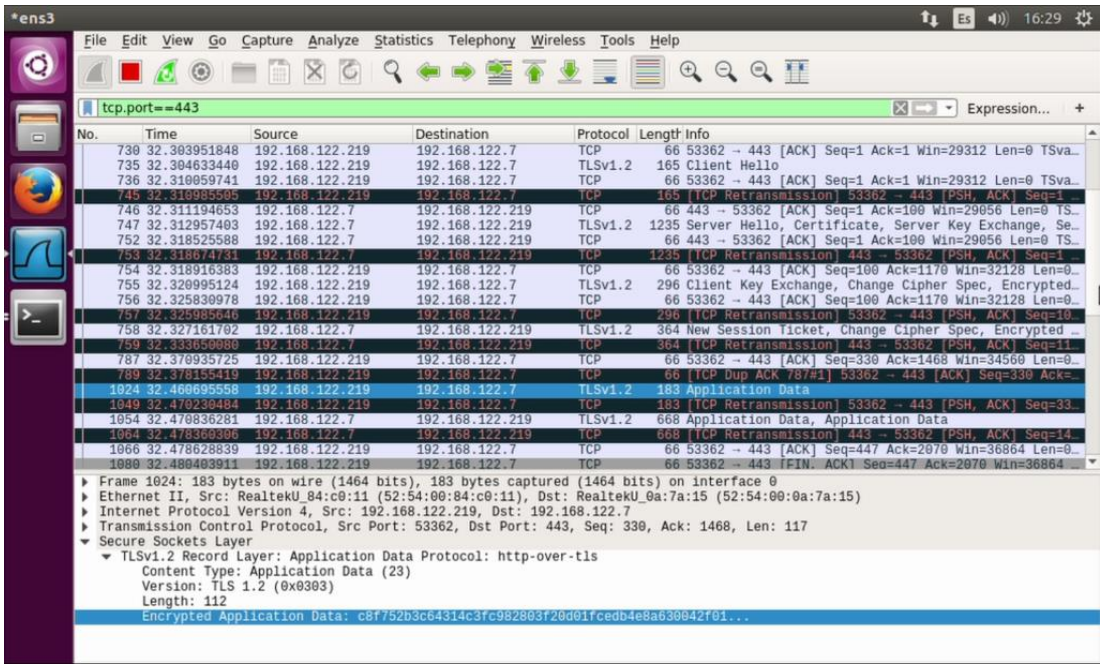

Figura 39. (3) Análisis de paquetes.

El siguiente análisis fue verificar que el tráfico de respuesta también fuese cifrado. El paquete 811 y 1024 de las imágenes anteriores corresponden a los paquetes que pasan por el proxy VNF y que previamente el servidor ha generado. Como se ve en la imagen anterior, el tráfico que proviene desde el Proxy VNF hacia el middlebox, va encriptado, ya que el campo Data muestra el contenido cifrado.

3.3 Segundo caso demostrativo

El segundo caso que se quiso analizar consistía en hacer la misma petición al servidor usando las mismas herramientas que en el primer caso, con la diferencia que ahora el cliente sólo iba a permitir acceso de lectura al proxy VNF.

Siguiendo el mismo procedimiento, la petición realizada por el cliente fue la siguiente.

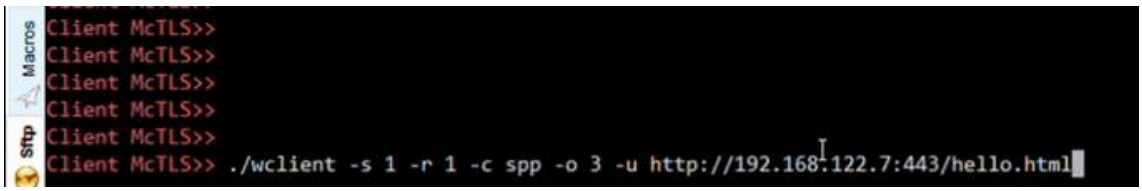

Figura 40. Petición https desde el cliente con permiso de lectura limitado

Los parámetros que ha puesto el cliente siguen siendo los mismo de caso anterior, solo que el campo (-r) ya no equivale a dos sino a uno, dando a entender que sólo el proxy VNF tendría acceso de lectura.

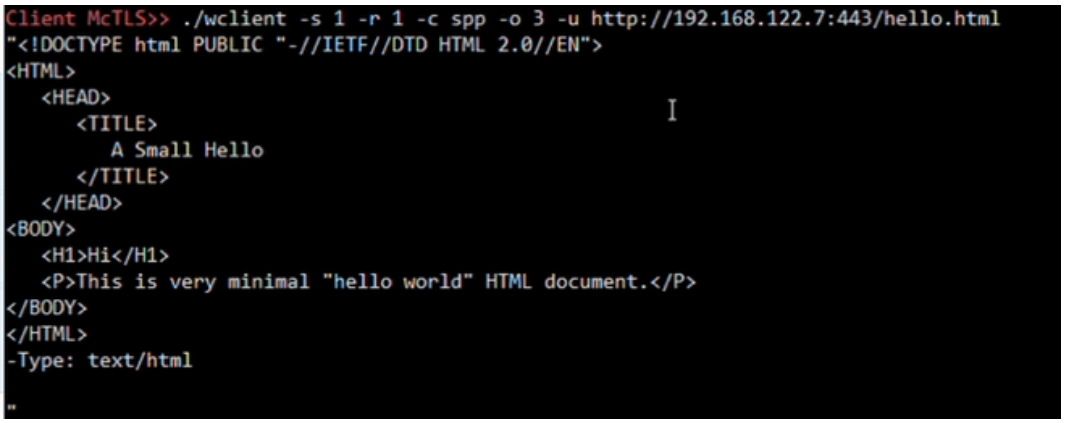

Figura 41. Respuesta https.

La respuesta se ha realizado con éxito, mostrando el contenido que el cliente había solicitado.

El contenido en el proxy VNF no varía, ya que se le ha asignado previamente desde el cliente accesos de lectura, por lo tanto puede ver el contenido transmitido, en la siguiente imagen se puede visualizar las cabeceras y el tipo de petición que ha realizado el cliente (GET).

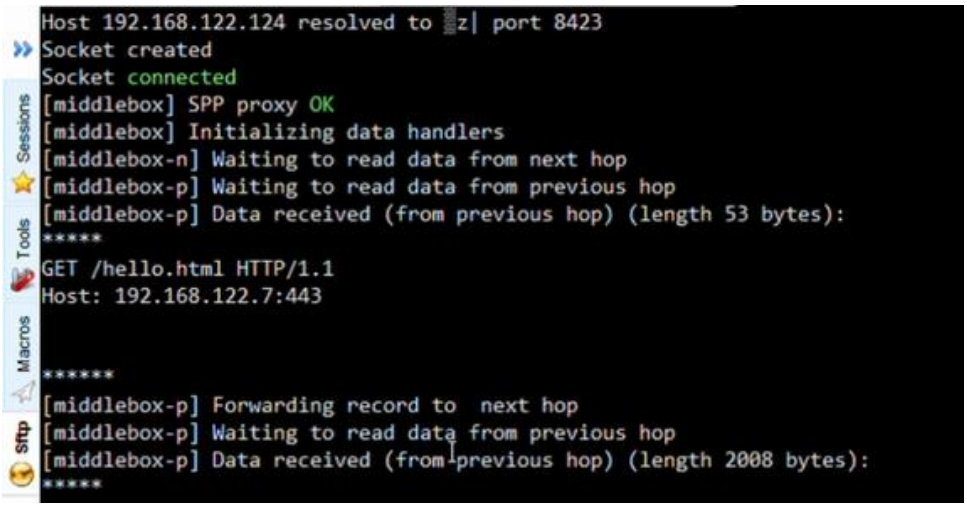

Figura 42. GET de contenido.

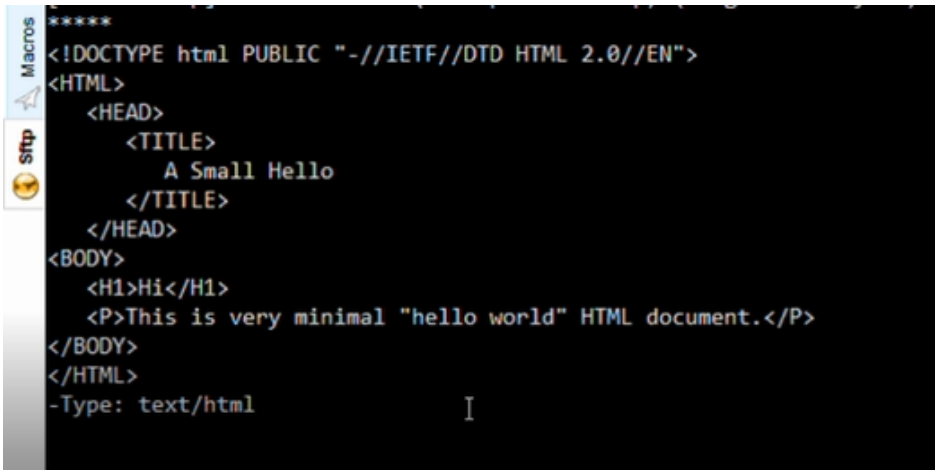

Figura 43. Respuesta del contenido.

Ahora desde el proxy VNF se puede visualizar el contenido enviado desde el servidor hacia el cliente.

Desde el middlebox se podrá visualizar que el contenido está cifrado ya que desde el cliente no se le dio permisos de lectura, por lo que actuará de proxy y sólo enviara el contenido al siguiente salto (proxy VNF) sin revisar los datos.

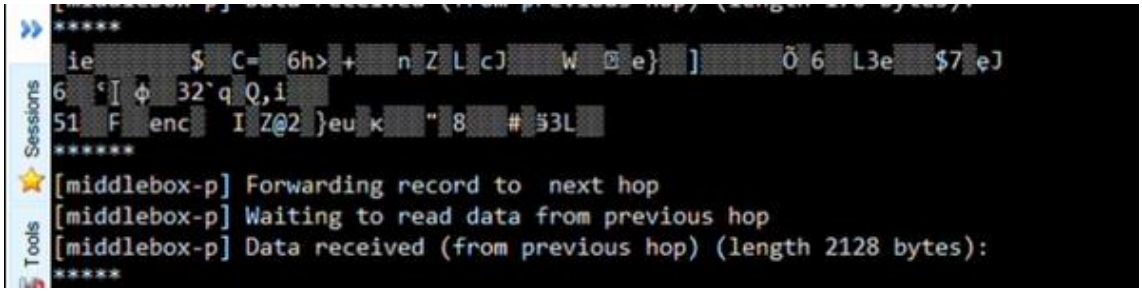

Figura 44. GET de contenido visto desde middlebox.

La cabecera con el tipo de petición que se veía en el proxy VNF no se puede visualizar de la misma manera en el middlebox ya que se prohibió leer los datos enviados y recibidos.

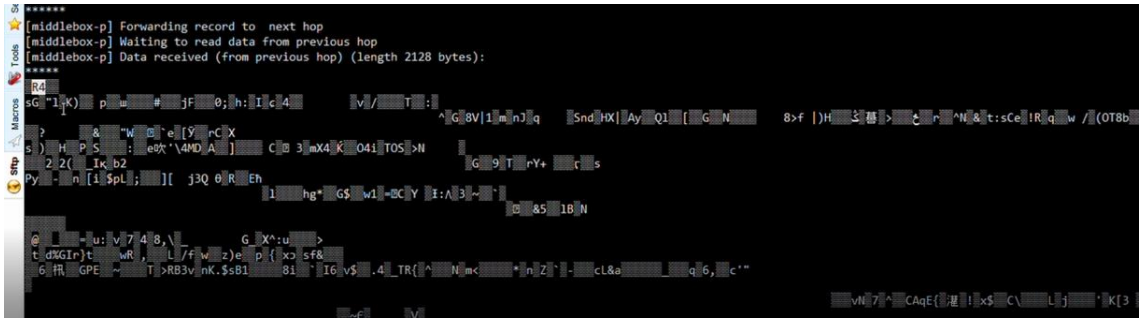

Figura 45. Respuesta de servidor vista desde middlebox.

La figura anterior muestra que el contenido que solicitó el cliente tampoco puede ser visto claramente desde el middlebox.

La conversión de SPP a SSL realizó el mismo procedimiento que el caso anterior, en las imágenes (Datos leídos por el proxy VNF [1-4]) del caso anterior es posible ver las pruebas, que fueron idénticas a las de este segundo caso.

3.3.1 Datos Wireshark (2)

En las pruebas tomadas desde Wireshark no reflejan mayor cambio que el primer caso, los análisis realizados fueron los mismos, tomando tráfico por el puerto 80 para verificar que no se estaba enviando nada por este puerto, seguido de las capturas realizadas a través de puerto 4433 (puerto donde escuchaba el proxy VNF) donde se quiso verificar que los datos fuesen cifrados y el atacante no podría verlos y posteriormente desde el puerto 443 para cerciorarse que los datos estarían viajando al servidor mediante SSL.

Figura 46. Captura de tráfico del puerto 80.

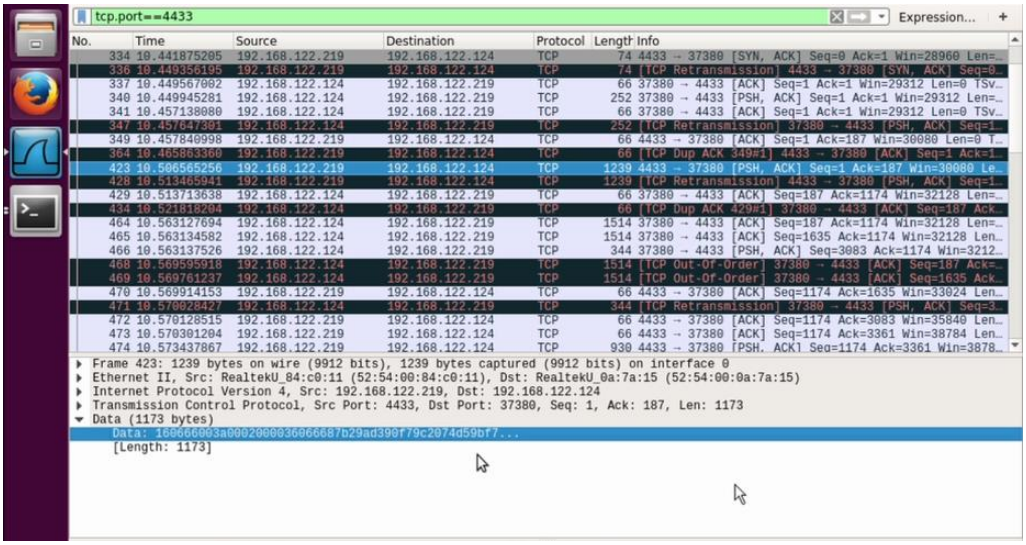

Figura 47. Captura de tráfico puerto 4433.

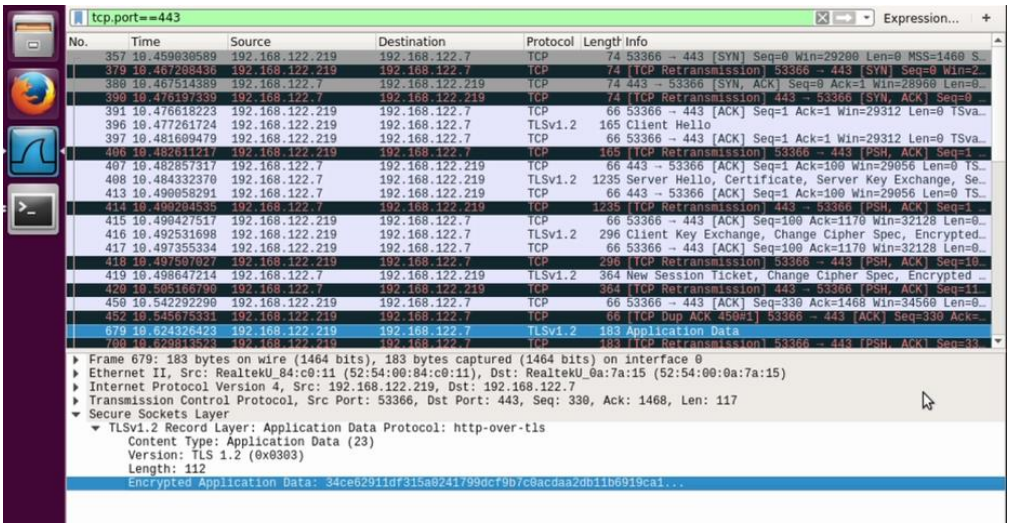

Figura 48. Captura de tráfico puerto 443.

4 Conclusiones y líneas futuras de investigación

Una vez realizada el análisis e investigación del proyecto descrito en las secciones anteriores, en este capítulo se explican las conclusiones generadas a lo largo del desarrollo e integración de McTLS con TLS, además se plantean posibles líneas de investigación futuras, donde sea posible abarcar en ciertos aspectos que quedan pendientes después de la realización de este proyecto.

4.1 Conclusiones

El descubrimiento de McTLS en favor de la seguridad, las redes, optimización y los sistemas finales ha llevado a los proveedores de servicio a impulsar de una u otra forma a que se estandarice esta tecnología para beneficio mutuo. Debido a esto, McTLS ha sido el impulso necesario para que hoy en día, se haya convertido en un estándar aprobado por la ETSI (*middlebox security protocol*) [39].

McTLS hace parte de una pequeña parte de millones de funcionalidades que se implementan en middlebox para optimizar y proteger los sistemas de información, por lo que, a razón de los componentes y características que ofrece McTLS, podemos analizar que es una puerta de entrada para mitigar y/o adelantarse a las amenazas cibernéticas en favor de todos los usuarios que usan internet, ya que McTLS abre la posibilidad de monitorizar y analizar el tráfico encriptado y a partir de ahí generar acciones de contramedidas a amenazas que afectan diariamente a servidores y usuarios en internet.

Fue posible concluir que existe la posibilidad de implementar McTLS e integrarla con conexiones TLS, además, esta integración hace que los servidores que proveen contenidos no tengan que cambiar el formato ni protocolo de encriptación que han usado durante años (TLS), por lo que hace más llamativo este desarrollo.

Aunque el middlebox tendrá cierto control sobre la información que viaje a través de la red, será un control que previamente haya sido aceptado por el usuario, ya que los middlebox por sí mismos no podrán ejercer ninguna acción, por defecto sino se aplica ningún permiso por el usuario, el(los) middlebox solo actuaran de proxy, por lo que depende de los permisos que el cliente aplique, por eso fue posible observar que no se está violando la privacidad del usuario, ya que el mismo usuario (cliente) es el que permite o no permite que los datos que envía por la red puedan ser manejados por los proveedores de servicio para ayudar a mejorar disminuir amenazas y optimizar las comunicaciones.

A nivel de código se pudo analizar que TLS y McTLS van relacionados en la forma como trabajan y encriptan la información, esto dio como resultado que se realizara un estudio de tráfico para verificar que los datos que viajaran a través de McTLS fueran encriptados, los estudios mostraron que la información viaja encriptada del mismo modo que TLS, también se analizó que en ningún momento la información puede ser vista por terceros o sistemas que no hayan sido aceptados previamente en la sesión, por lo que la confidencialidad podría permanecer.

Debido a la cantidad de procesos que se añaden durante la comunicación entre el cliente y servidor, McTLS aparte de factible también genera sobrecarga en cuestión de latencia, tiempo de carga y sobrecarga de los datos.

4.2 Líneas futuras de investigación

Este proyecto fue simulado en un entorno que incluía máquinas virtuales donde se ejecutaban aplicaciones embebidas en procesos dentro de un sistema invitado, estas aplicaciones son los middlebox, que pueden traducirse a cajas que en su interior hacen alguna función específica, según esto, hace pensar que es posible integrar estos middlebox en la tecnología NFV siendo un middlebox una VNF que en su interior tengan herramientas software como la propuesta de McTLS que se planteó en este proyecto. Por lo tanto, un análisis futuro sería integrar estos middlebox en un entorno NFV donde estos middlebox sean lanzados una vez se inicia una sesión según las políticas establecidas previamente y así pueda ser integrado con otras VNFs que realicen alguna acción una vez el middlebox haya visto la información que se transmite entre el cliente y servidor.

Sería interesante analizar el comportamiento de un middlebox y alguna tecnología que ya exista en el mercado como SQUID, que a pesar de usarse como proxy en sesiones TLS, también puede realizar otras más funciones o acciones a datos que no son encriptados, es decir, se podría integrar un middlebox como la función que es capaz de leer o manipular datos que vienen encriptados bajo McTLS y traducirlos a datos que puedan ser tratados por TLS y usar SQUID para que aplique filtrados más específicos y así ampliar la utilidad de middlebox como herramienta de seguridad.

Este proyecto hizo un enfoque específico en el desarrollo y análisis de integrar McTLS y TLS en una misma sesión mediante middlebox. Sin embargo, la otra parte que también usa el protocolo McTLS es el cliente, el cual tiene un análisis muy general, ya que los navegadores de internet (Google, Firefox, etc.) son navegadores que usan conexiones TLS para iniciar comunicación segura de los datos, así que un buen punto de línea de investigación futura sería el cliente McTLS donde se pueda desarrollar y analizar el cliente como aplicación, sistema operativo o la posibilidad de integrar McTLS en los navegadores web.

Bibliografía

[1] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste,"The Cost of the "S" in HTTPS". Carnegie Mellon University, Politecnico di Torino, Telefónica Research.

[2] Michael Kranch, Joseph Bonneau "Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning", Princeton University.

[3] CERT, <http://www.cert.org>, Cabrera JBD, Gosar J, Lee W, Mehra RK, On the statistical distribution of processing times in network intrusion detection, In: 43rd IEEE conference on decision and control, Paradise Island, Bahamas, 2004, pp. 75-80.

[4] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego Lopez, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste, "Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS". Carnegie Mellon University, Case Western Reserve University, Telefónica Research.

[5] Jordi Ferrer Riera, Eduard Escalona, Josep Batalle, Eduard Grasa, Joan A. García-Espín, "Virtual Network Function Scheduling: Concept and Challenges" Distributed Applications and Networks Area.

[6] Título: The Design of Rijndael: AES - The Advanced Encryption Standard Information Security and Cryptography. Autores: Joan Daemen, Vincent Rijmen. Edición: ilustrada. Editor: Springer Science & Business Media, 2013. ISBN: 3662047225, 9783662047224. N.º de páginas: 238 páginas.

[7] Nadhem AlFardan, Royal Holloway, University of London; Daniel J. Bernstein, University of Illinois at Chicago and Technische Universiteit Eindhoven; Kenneth G. Paterson, Bertram Poettering, and Jacob C.N. Schuldt, Royal Holloway, University of London. "On the Security of RC4 in TLS".

[8] Pravir Chandra, Matt Messier, John Viega. "Network Security with OpenSSL", Publisher : O'Reilly, Pub Date : June 2002, ISBN : 0-596-00270-X, Pages : 384

[9] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, Peter Steenkiste. "The Cost of the "S" in HTTPS".

[10] The Chromium Projects. Spdy. http://www.chromium.org/spdy.

[11] K. Jang, S. Han, S. Moon and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In NSDI 2011

[12] C. Castelluccia. Improving Secure Server Performance by Rebalancing SSL/TLS Handshakes. In USENIX Security Symposium, 2005.

[13] Sevil Mehraghdam, Matthias Kelle, Holger Karl, "Specifying and Placing Chains of Virtual Network Functions"

[14] Josep Batalle, Jordi Ferrer Riera, Eduard Escalona and Joan A. García-Espín. "On the implementation of NFV over an OpenFlow infrastructure: Routing Function Virtualization"

[15] B. Carpenter, IBM Zurich Research Laboratory, S. Brim. RFC 3234 "Middleboxes: Taxonomy and Issues".

[16] Border J, Kojo M, Griner J, Montenegro G, Shelby Z. "Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations", IETF RFC 3135, June 2001

[17] K. Brown, S. Singh, "M-TCP: TCP for Mobile Cellular Networks," ACM Computer Communications Review Volume 27(5), 1997.

[18] Load Balancer F5. https://f5.com/glossary/load-balancer

[19] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, Kuang-Yuan Tung. "Intrusion detection system: A comprehensive review".

[20] N.M Mosharaf Kabir Chowdhury, Raouf Boutaba. "A survey of network virtualization".

[21] Nikitasha P, Jyotiprakash S, Subasish M, Prasanna PS. "A security framework for virtualization based computing environment. International Journal of Engineering Science and Technology 2011;3:6423-9.

[22] Guofu Xiang, Hai Jin, Deqing Zou, Xinwen Zhang, Sha Wen, Feng Zhao: "VMDriver: A Driver-based Monitoring Mechanism for Virtualization"

[23] P. N. Vijaya Kumar, Dr. V. Raghunatha Reddy. "Novel Web Proxy Cache Replacement Algorithms using Machine Learning Techniques for Performance Enhancement".

[24] European Telecommunications Standards Institute 2014. "ETSI GS NFV 002 V1.2.1 (2014-12)"

[25] European Telecommunications Standards Institute. ETSI GS NFV 001 V.1.1.1 (2013-10). "Network Function Virtualization (NFV) Use Cases"

[26] P. Lepeska. Trusted proxy and the cost of bits. https://www.ietf.org/proceedings/90/slides/slides-90-httpbis-6.pdf, 7 2014

[27] X. Xu, Y. Jiang, T. Flach, et al. investigating transparent web proxies in cellular networks.

[28] S. Woo, E. Jeong, S. Park, et al. Comparison of caching strategies in modern cellular backhaul networks. MobiSys '13, pages 319–332, New York, NY, USA, 2013. ACM.

[29] F. Qian, S. Sen, and O. Spatscheck. Characterizing resource usage for mobile web browsing. MobiSys '14, pages 218–231, New York, NY, USA, 2014. ACM.

[30] M. Hoque, M. Siekkinen, and J. K. Nurminen. On the energy efficiency of proxy-based traffic shaping for mobile audio streaming. CCNC, pages 891–895. IEEE, 2011

[31] A. Menezes and B. Ustaoglu. On reusing ephemeral keys in diffie-hellman key agreement protocols. International Journal of Applied Cryptography, 2(2):154–158, 2010.

[32] Data Saver. https://developer.chrome.com/multidevice/data-compression

[33] GitHub McTLS. https://github.com/scoky/mctls

[34] Openssl, Cryptography and SSL/TLS Toolkit. https://www.openssl.org/docs/man1.0.2/ssl/ssl.html

[35] Openssl, Cryptography and SSL/TLS Toolkit. https://www.openssl.org/docs/man1.0.2/ssl/SSL_new.html

[36] mcTLS prototype implementation based upon OpenSSL. https://github.com/scoky/mctls

[37] Ettercap. https://www.ettercap-project.org/index.html

[38] Wireshark. https://www.wireshark.org/

[39] ETSI, middlebox security protocol, https://www.etsi.org/etsi-security-week-2018/middlebox-security