Requisitos de Seguridad para TLS

1. INTRODUCCIÓN A TLS

TLS (Transport Layer Security Protocol) es un protocolo de seguridad criptográfico que permite proteger las comunicaciones. Utiliza algoritmos y funciones criptográficas para proporcionar los siguientes servicios de seguridad a los datos transmitidos:

- Protección de la **confidencialidad** mediante el uso de algoritmos de cifrado simétrico. Las claves para este cifrado se generan a partir de un secreto pre-compartido (*pre_master_secret*) que se establece entre cliente y servidor en las primeras fases del protocolo, utilizando métodos de intercambio de claves (*Key Exchange*).
- Protección de la **integridad y autenticidad de origen** mediante el uso de funciones HMAC (*Hash-based Message Authentication Code*).
- **Autenticación** mediante el uso de certificados de clave pública X.509.
- **Protección anti-reenvíos**, mediante números de secuencia añadidos al mensaje.
- ***Forward Secrecy*** (secreto hacia adelante), asegurando que, si la clave se descubre en un momento dado, no podrá ser usada para descifrar ninguno de los datos trasmitidos anteriormente.
- **Control** de sesión a través de un sistema de alertas avanzado, que notificará al usuario inmediatamente en el caso de que algún error o incidencia ocurra.

TLS es un estándar definido por IETF (*Internet Engineering Task Force*), especificado por primera vez en 1999. La versión actual es TLS 1.3 especificada en la RFC 8446, de agosto 2018.

TLS es un protocolo cliente-servidor. Uno de los extremos de la comunicación actuará como cliente y el otro, como servidor TLS.

TLS tiene un uso muy extendido. El más conocido es la protección del tráfico HTTP entre un cliente web (*web browser*) y un servidor o aplicación web. Este uso de TLS para proteger el tráfico HTTP constituye el protocolo HTTPS. Además, también se utiliza para proporcionar cifrado y otros servicios de seguridad a los datos transmitidos por otros protocolos, como SMTP (email), IM (mensajería instantánea), VoIP (voz sobre IP), FTP (transferencia de ficheros), etc.

2. VERSIONES TLS

Secure Sockets Layer, o SSL, era el nombre que originalmente le dio *Netscape* cuando desarrolló el protocolo a mediados de 1990. En 1996 se publicó una nueva versión completamente renovada que recibió la denominación de SSL 3.0 y estableció las bases para lo que luego se conocería como TLS, que apareció cuando la siguiente versión se publicó en 1999 y el IETF (*Internet Engineering Task Force*) la convirtió en un estándar y le dio el nombre de *Transport Layer Security,* o TLS.

El uso de SSL 3.0 (SSLv3) no está recomendado. Así se indica en la RFC 7568 - *Deprecating Secure Sockets Layer Version 3.0,* de 2015. Esta RFC considera SSLv3 como un protocolo inseguro dada la larga serie de ataques que ha sufrido, tanto a sus mecanismos de acuerdo de claves, como a los de cifrado.

SSLv3 fue reemplazado por TLS 1.0, 1.1, 1.2 y, finalmente, por la última versión, TLS 1.3.

Versión	RFC	Fecha	Última actualización
TLS 1.0	RFC 2246	Enero 1999	Febrero 2013
TLS 1.1	RFC 4346	Abril 2006	Enero 2020
TLS 1.2	RFC 5246	Agosto 2008	Enero 2020
TLS 1.3	RFC 8446	Agosto 2018	Marzo 2020

Tabla 1. Versiones TLS

TLS 1.0 y 1.1 son versiones bastante obsoletas de TLS y presentan muchas vulnerabilidades.

TLS 1.2 introduce muchas mejoras respecto a las versiones anteriores. En especial, la capacidad de usar *SHA-2* para los cálculos de HMAC y para su uso en funciones pseudoaleatorias (PRF). También introduce el uso de los modos de cifrado AEAD (*Authenticated Encryption with Additional Data*)., que proporcionan simultáneamente confidencialidad, integridad y autenticidad.

TLS 1.3 representa un cambio significativo y tiene por objetivo abordar las amenazas que han surgido desde la especificación de TLS 1.2. Entre los cambios más significativos se encuentra el rediseño del *Handshake*, haciéndolo más rápido y eficiente, y la incorporación de mejoras criptográficas. Las mejoras de TLS 1.3 se pueden consultar en los apartados 6.5 y 8.

Sin embargo, muchas extensiones que se definieron en TLS 1.2 no se pueden utilizar en TLS 1.3, por lo que el uso de esta nueva versión por parte de los sistemas y aplicaciones puede requerir un tiempo de adaptación.

Requisito 1:

Se deberá usar la versión del protocolo **TLS 1.2 o superior** y deshabilitar el uso de cualquier versión inferior.

3. USOS DE TLS

VPN SSL (TLS)

Una **VPN SSL**[1] proporciona un canal de acceso seguro a los recursos de una organización. Está formada por varios *Gateways* o Servidores VPN a los que los usuarios se conectan, a través de navegadores web, o a través de un *software* específico instalado en el equipo del usuario (cliente VPN). Todo el tráfico entre el navegador o cliente, y el servidor VPN, estará protegido y cifrado a través del protocolo TLS. Se contemplan dos (2) casos de uso:

- **VPN SSL Portal.** El usuario remoto se conecta mediante una conexión TLS a un "Portal Web". Este normalmente realiza funciones de *proxy*. A partir de este Portal se accede de forma segura a un conjunto limitado de aplicaciones y servicios de la red interna de la organización, para los que el usuario tenga autorización.
- **VPN SSL Túnel.** En este caso es necesario instalar un cliente VPN en el equipo del usuario. Este proporciona un túnel de capa 3, a través del cual los usuarios pueden acceder a la red interna de la organización.

Requisito 2:

Para aquellos **sistemas bajo el alcance del Esquema Nacional de Seguridad (ENS)**, deberán utilizarse productos recogidos en el apartado de **Productos Cualificados** para la categoría correspondiente en la familia **Redes Privadas Virtuales SSL dentro del Catálogo de Productos de Seguridad TIC (CPSTIC).**

Del mismo modo, en el caso de productos que vayan a ser utilizados en sistemas clasificados deberán utilizarse los incluidos en el apartado de Productos Aprobados para el nivel de clasificación del sistema considerado.

En ambos casos, deberá tenerse en cuenta el Procedimiento de Empleo Seguro publicado por el CCN.

HTTPS

HTTPS se trata de la extensión segura del protocolo HTTP. Este es utilizado típicamente por el navegador y los servidores web, para comunicarse e intercambiar información. Cuando el transporte de datos se protege con TLS, entonces se denomina HTTPS.

Un servidor HTTPS debe contar con un certificado de clave pública (normalmente llamado certificado SSL) que el cliente *web* pueda validar, autenticando con ello al servidor. El servidor también puede solicitar un certificado al cliente, pero no suele ser frecuente en los escenarios más habituales de uso de HTTPS.

Existen otros protocolos que utilizan HTTPS. Por ejemplo, el protocolo EST (*Enrollment over Secure Transport*)[2]. Este protocolo especifica cómo transferir mensajes de forma segura utilizando HTTPS,

[1] Aunque el protocolo empleado para la VPN sea TLS, estas se siguen conociendo como VPN SSL.
[2] RFC 7030.

entre un cliente y una CA (*Certificate Authority*). Estos mensajes son, por ejemplo, la petición de certificado CSR (*Certificate Signing Request*).

El funcionamiento básico de HTTPS se explica en la siguiente imagen.

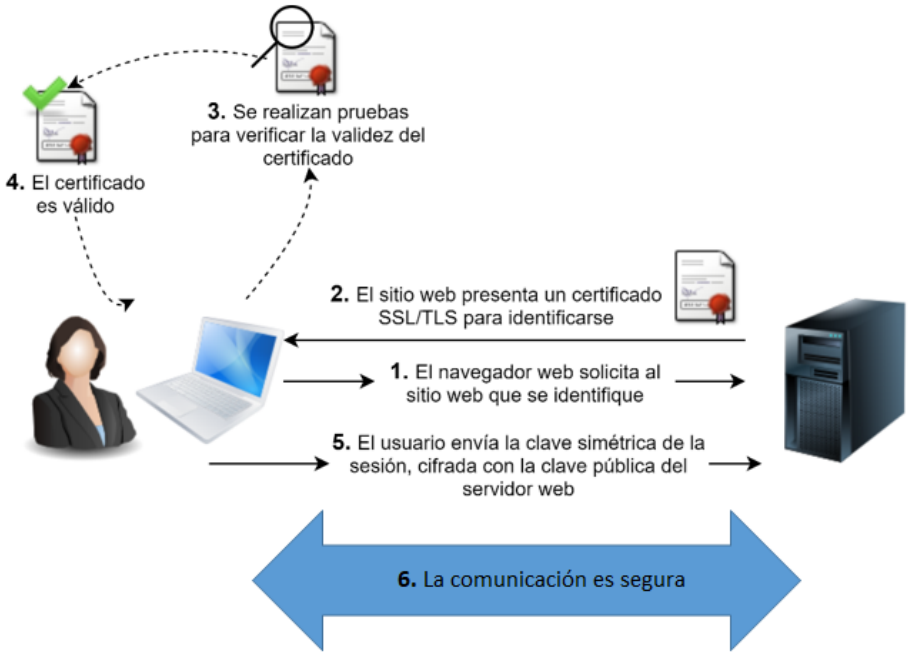

Figura 1. Funcionamiento de HTTPS

Otros protocolos

A continuación, se indican diversos protocolos que hacen uso de TLS o que están basados en TLS.

FTPS[3]

FTP es el protocolo de transferencia de ficheros más extendido. El uso de TLS le aporta la capa de seguridad y cifrado necesario. Hay dos modos de funcionamiento para FTPS:

- **FTPS (FTP implícito sobre TLS).** Se trata de un modo de FTP sobre TLS que está ya obsoleto. El cliente FTP se conecta a un puerto distinto al TCP 21 (canal de control de FTP). Antes de intercambiar ninguna información con el servidor FTP, se realiza la negociación TLS. A partir de entonces, la autenticación y la transferencia de archivos van protegidos con TLS.
- **FTPES (FTP explícito sobre TLS).** Es el modo de FTP sobre TLS más moderno y extendido en la actualidad. El cliente establece una conexión FTP estándar a través del puerto 21 con el servidor y, una vez conectado, solicita explícitamente la negociación TLS. A partir de entonces se establecerá la conexión cifrada para la autenticación y transferencia de ficheros. FTPES soporta TLS 1.2 y TLS 1.3.

DTLS

DTLS (*Datagram Transport Layer Security)* es la adaptación de TLS para operar sobre UDP, para comunicaciones a través de datagramas. Está diseñado para proporcionar **garantías de seguridad**

[3] FTPS es distinto a SFTP, que es FTP basado en SSH (*SSH File Transfer Protocol*).

similares a TLS, manteniendo las características de las comunicaciones con datagramas: una baja latencia, alta velocidad de transferencia de datos y tolerancia a las pérdidas de comunicación.

DTLS es apropiado, por tanto, para aquellos protocolos que requieren muy poca latencia, pero son tolerantes a pérdidas de paquetes o a que estos lleguen desordenados. Este podría ser el caso de VoIP, SIP (*Session Initiation Protocol*), aplicaciones de video *streaming*, etc.

TLS *Syslog*

TLS *Syslog* es el protocolo *syslog* sobre TLS. Permite proteger registros de auditoría enviados por redes no confiables. Permite incluir varios mensajes *syslog* en un sólo paquete TLS, o un mensaje *syslog* en múltiples paquetes TLS. El dispositivo que envía los registros es el cliente TLS. El dispositivo receptor de los registros (servidor *syslog* remoto) es el servidor TLS. El servidor debe disponer siempre de un certificado, de forma que el cliente lo pueda autenticar. Es recomendable configurar una autenticación mutua de forma que el cliente disponga de un certificado para que el servidor también lo pueda autenticar.

4. FUNCIONAMIENTO DE TLS: *HADSHAKE* & *RECORD PROTOCOLS*

TLS se compone de cuatro (4) protocolos. Los dos (2) principales son el protocolo *Handshake* y el protocolo *Record*. A través del *Handshake* se negocian los parámetros de la conexión, se realiza la autenticación de cliente y servidor, y se generan las claves necesarias para el cifrado y autenticación de los mensajes. *Record Protocol* es el encargado de la transmisión y recepción de los mensajes, usando las claves y algoritmos negociados. Los otros dos (2) protocolos son: *Change Cipher Spec* y *Alert*. El primero de ellos, realmente solo está compuesto de un mensaje, y se utiliza para indicar el cambio en la estrategia de cifrado, tal y como se indica más adelante. El protocolo *Alert*, envía mensajes de alerta que contienen la severidad (*Warning* o *Fatal*) y una descripción de la alerta. Los mensajes con severidad *Fatal* producen una finalización inmediata de la conexión TLS. Se pueden consultar los mensajes de alerta disponibles en TLS 1.2 en la sección 7.2 de la RFC 5246.

4.1. Handshake Protocol

Toda conexión TLS comienza con un *Handshake,* una negociación entre cliente y servidor en la que se establecen los detalles de cómo se va a realizar la comunicación.

Durante el *Handshake,* entre el cliente y el servidor se realiza, principalmente, lo siguiente:

- Negociar y acordar los algoritmos criptográficos (*cipher suite* y algoritmos de firma), e intercambiar valores aleatorios (*random*).
- Intercambiar parámetros entre cliente y servidor, necesarios para obtener el secreto pre-compartido (*pre_master_secret*).
- Intercambiar certificados para permitir que cliente y servidor se autentiquen mutuamente.
- Generar el *master_secret* a partir del secreto pre-compartido y de los valores aleatorios intercambiados. A partir del *master_secret*, se derivará el valor de claves.
- Proporcionar al *Record Protocol*, los parámetros de seguridad.

Un *TLS Handshake* implica diversos pasos que varían dependiendo del método de *Key Exchange* usado, y de las *cipher suites* seleccionadas por cliente y servidor.

Lo primero que se realiza es el establecimiento de conexión por parte del protocolo de transporte, en este caso TCP. Una vez que se ha establecido la comunicación, comienza el TLS *Handshake,* cuyos pasos se describen a continuación.

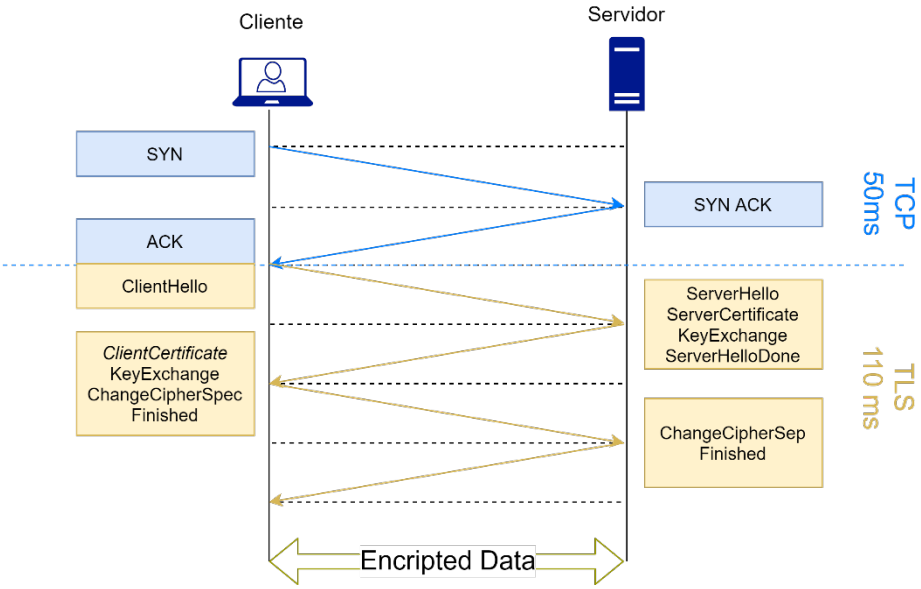

Figura 2. TLS *Handshake Protocol*

Mensaje	Contenido del Mensaje
ClientHello (enviado por cliente)	▪ Versión de TLS que soporta (*client_version*). ▪ Identificador de la sesión (*Session ID*). ▪ Listado de *cipher suites* que soporta. ▪ Métodos de compresión que soporta. ▪ El *client.random*, un número aleatorio de 32 bytes, que va a ser usado para derivar el material de claves. ▪ Extensiones. Son parámetros opcionales entre los que está la extensión de *signature_algorithms* que puede usar el cliente para indicar al servidor los algoritmos que soporta para verificación de la firma del certificado. Si no envía esta extensión, el servidor tomará unos valores por defecto (ver RFC 5246 apartado 7.4.1.4.1).
ServerHello (enviado por servidor)	▪ Selección de la versión de TLS. ▪ Session ID. ▪ La *cipher suite* elegida. ▪ Método de compresión seleccionado. ▪ El *server.random* que junto con el *client.random* va a ser usado para derivar el material de claves.

Mensaje	Contenido del Mensaje
ServerCertificate (enviado por servidor)	▪ Certificado del servidor (cadena de certificados): o La clave pública del certificado debe ser acorde al método de *Key Exchange* y el algoritmo de clave pública negociados en la *cipher suite*. o La firma del certificado debe ser acorde a los algoritmos de firma soportados por el cliente, e indicados en la extensión *signature_algorithms* del *ClientHello*.
Server KeyExchange (enviado por servidor)	▪ Este mensaje se envía inmediatamente después del *ServerCertificate,* únicamente cuando el certificado enviado por el servidor no proporciona suficiente información para que el cliente pueda establecer el secreto pre-compartido. ▪ Esto solo ocurre cuando los métodos de intercambio de clave seleccionados son DHE o ECDHE, con certificados de clave pública RSA, ECDSA o DSS. Es estos casos, el servidor en el *ServerCertificate* ha enviado el certificado con la clave pública RSA, ECDSA o DSS, pero el cliente necesita los parámetros *Diffie-Hellman* del servidor, para generar el secreto. Por ejemplo, en el caso de DHE, el servidor enviará en el mensaje *ServerKeyExchange* los parámetros DH (p y g) y su clave pública DH (ver más información en apartado 6.1). El mensaje, además, irá firmado con la clave privada RSA, ECDSA o DSS del servidor.
CertificateRequest (enviado por servidor)	▪ Este mensaje es la petición de certificado del servidor al cliente cuando hay autenticación mutua. Se indica de qué tipo debe ser el certificado (acorde con el método de *Key Exchange* y el algoritmo de clave pública negociados en la *cipher suite*), qué algoritmos de firma soporta el servidor y un listado de CAs raíz que aceptará el servidor.
Server HelloDone (enviado por servidor)	▪ El servidor finaliza el *ServerHello*.
ClientCertificate (enviado por cliente)	▪ Este mensaje solo se enviará si el servidor ha solicitado un certificado al cliente, a través del mensaje *CertificateRequest*. ▪ Con este mensaje, el cliente envía el certificado (cadena de certificados) que deberá cumplir con los requisitos indicados por el servidor.

Mensaje	Contenido del Mensaje
ClientKeyExchange (enviado por cliente)	▪ En este mensaje el cliente envía al servidor los parámetros que este necesita para generar el secreto pre-compartido: a) En caso de que el método de *Key Exchange* seleccionado sea DHE o ECDHE, se envían los parámetros *Diffie-Hellman*. Por ejemplo, en el caso de DHE se enviaría la clave pública DH del cliente. b) En caso de que el método de *Key Exchange* seleccionado sea RSA, el cliente genera un secreto pre-compartido de 48 bytes y lo envía al servidor, cifrado con la clave pública RSA del servidor obtenida del certificado. ▪ Tras el envío de este mensaje, servidor y cliente disponen del mismo secreto pre-compartido.
CertificateVerify (enviado por cliente)	▪ Este mensaje solo se envía en caso de que el cliente le haya mandado su certificado al servidor y siempre que la clave pública del certificado se pueda utilizar para firma, lo cual se aplica a certificados con claves RSA, DSS, ECDSA, y no a certificados con claves estáticas DH o ECDH. ▪ El mensaje *CertificateVerify* incluye los mensajes *Handshake* enviados hasta entonces por el cliente, firmados con su clave privada. El objeto de este mensaje es que el cliente demuestre explícitamente al servidor que está en posesión de la clave privada.
ChangeCipherSpec (enviado por cliente y servidor)	▪ Este mensaje realmente no pertenece al Protocolo *Handshake*, sino que es el único mensaje que compone el Protocolo *Change Cipher Spec* cuyo objetivo es, únicamente, señalizar un cambio en la estrategia de cifrado. ▪ En este caso, cliente y servidor deben enviar un mensaje de este tipo para indicar que disponen de todos los parámetros de seguridad y están preparados para el cambio a un entorno cifrado.
Finished (enviado por cliente y servidor)	▪ Cliente y servidor deben enviar este mensaje inmediatamente después de Ch*angeCipherSpec.* ▪ Es el primer mensaje que se envía cifrado con el algoritmo negociado y el *master secret* calculado (*). ▪ Cliente y servidor deben recibir el *Finished* del otro lado y verificar que el contenido es correcto.

Tabla 2. Pasos de *Handshake*

(*) Generación del *Master Secret*

Como se ha comentado en los pasos anteriores, tras el intercambio de los mensajes *ClientKeyExchange* y *ServerKeyExchange*, servidor y cliente disponen del secreto pre-compartido. A

partir de este secreto, se genera el secreto maestro (*master secret*) y, a partir de él, en el *Record Protocol* se derivará el material de claves necesario.

El *master secret* tendrá una longitud de 48 bytes. Para calcularlo se utiliza una función PRF (*Pseudorandom Function*) que se basará en la función *hash* de la *cipher suite*. A la función PRF se le pasan como valores de entrada: el secreto pre-compartido y los valores *client.random* y *server.random* intercambiados en el *Handshake*.

master_secret = PRF (pre_master_secret, "master secret", client.random + server.random) [0..47]

El valor "*master secret*" representa una cadena de caracteres que podría ser sustituida por cualquier otra secuencia.

5. RECORD PROTOCOL

TLS *Record Protocol* es el encargado de recoger los mensajes para enviarlos de forma segura. Para ello, utiliza los parámetros de seguridad (*Security Parameters*) proporcionados por el *Handshake*, y deriva el material de claves que necesita para la transmisión.

Estos parámetros de seguridad son los siguientes:

- Algoritmos para el cifrado y HMAC negociados y acordados en los mensajes *Hello* del *Handshake*.
- Método de compresión negociado y acordado en los mensajes *Hello* del *Handshake*.
- Valores *random* intercambiados en los mensajes *Hello* del *Handshake (client.random* y *server.random).*
- *Master_secret* derivado en el *Handshake*.

El proceso de derivación del material de claves se realiza de forma independiente en el cliente y el servidor y consiste en, a partir del *master_secret*, obtener los siguientes valores:

- Dos (2) claves que usará el cliente TLS en el envío de datos al servidor: una clave de cifrado *(client_write_key*) y otra clave para el cálculo del valor MAC (*client_write_MAC_key*), y un vector de inicialización (client_write_IV), en caso de utilizar cifrado con modos de operación AEAD.
- Dos (2) claves que usará el servidor TLS en el envío de datos al cliente: una clave de cifrado (*server_write_key*) y otra clave para el cálculo del valor MAC (*server_write_MAC_key*), y un vector de inicialización (*server_write_IV*), en caso de utilizar cifrado con modos de operación AEAD.

El tamaño de cada clave depende de los algoritmos negociados. Por ejemplo, si la *cipher suite* es TLS_DHE_RSA_WITH_AES_128_CBC_SHA256, significa que las claves de cifrado son de 128 bits (16 *bytes*) y las claves MAC son de 256 bits (32 *bytes*).

Para el cálculo de las claves se utiliza, además del *master_secret,* otros valores aleatorios generados durante el *Handshake*. Se utiliza también la función PRF de la siguiente forma:

key_block = PRF (master_secret, "key expansion", client.random + server.random);

El valor "*key expansion*" representa una cadena de caracteres que podría ser sustituida por cualquier otra secuencia.

Se computa el *key_block* de forma iterativa hasta que su tamaño es suficiente, lo que dependerá del tamaño de las claves necesarias. Entonces, se divide el *key_block* en los cuatro (4) o seis (6) valores siguientes:

client_write_MAC_key [Keyblock.mac_key_length]

server_write_MAC_key [Keyblock.mac_key_length]

client_write_key [Keyblock.enc_key_length]

server_write_key [Keyblock.enc_key_length]

client_write_IV [SecurityParameters.fixed_iv_length]

server_write_IV [SecurityParameters.fixed_iv_length]

Un flujo típico para el envío de datos es el siguiente:

1. El *Record Protocol* recibe el mensaje de la capa de aplicación que se va a transmitir.
2. El mensaje es dividido en bloques (*payload*), de máximo 2^{14} bytes, o 16 KB.
3. Cada bloque se comprime usando el método de compresión negociado en el *Handshake*.
4. Se calcula y se añade el MAC a cada paquete. Se utiliza la función *hash* negociada en el *Handshake* y la clave MAC derivada (*client_write_MAC_key* o server_*write_MAC_key*, en función de si el emisor de la comunicación es el cliente o el servidor).
5. Se cifran los datos de cada paquete mediante el algoritmo de cifrado negociado en el *Handshake* utilizando la clave de cifrado derivada (*client_write_key* o server_*write_key*, en función de si el emisor de la comunicación es el cliente o el servidor).
6. Una vez que se han completado estos pasos, la información cifrada se pasa a la capa de transporte TCP. En el caso contrario, se realizan los pasos a la inversa.

A continuación, se muestra la estructura de un paquete del protocolo *TLS Record*.

Byte	+0	+1	+2	+3
0	Content type			
1..4	Version		Lenght	
5..n	Payload			
n..m	MAC			
m..p	Padding (Solo cuando se usa cifrado por bloques)			

Figura 3. Estructura Paquete TLS *Record*

6. CIPHER SUITES Y OTROS PARÁMETROS CRIPTOGRÁFICOS

En la fase de negociación de TLS (*Handshake*), se acuerdan los parámetros de seguridad de la conexión. Entre ellos, lo que se denomina 'suite criptográfica' o ***cipher suite***. Esta *cipher suite* está compuesta por los algoritmos y funciones criptográficas, junto con sus parámetros (longitud de clave, modo de operación, etc.) que se van a utilizar para proteger la conexión.

De forma general, una *cipher suite* se representa así:

TLS_KeyExchange_Auth_WITH_Cipher_KeyLength_Mode_HashFunction

La primera parte de la *cipher suite* corresponde a los métodos de intercambio de clave (*Key Exchange*) y autenticación (*Auth*). La parte intermedia corresponde al algoritmo de cifrado (*Cipher*), su longitud de clave (*KeyLength*) y su modo de operación (*Mode*). La última parte es la función *hash* (*HashFunction*) que se utilizará para el cálculo del HMAC durante el envío de mensajes, y para la derivación del material de claves en la función PRF.

Existen multitud de *cipher suites* con distintas combinaciones de algoritmos[4]. El presente documento se centra únicamente en aquellas que utilizan algoritmos seguros según las recomendaciones de SOGIS[5] y de la guía CCN-STIC-807, tal y como se indica en los siguiente apartados.

6.1. *Key Exchange* – Autenticación

En TLS 1.2, la primera parte de la *cipher suite* negociada indica el método de ***Key Exchange*** y el **algoritmo de clave pública** para firma y autenticación (*Auth*).

TLS_KeyExchange_Auth_WITH_Cipher_KeyLength_Mode_HashFunction

6.1.1 *Key Exchange*

Los principales métodos que utiliza TLS 1.2 para el intercambio de claves o *Key Exchange* son: claves pre-compartidas (PSK), algoritmos de clave pública (RSA) y métodos *Diffie-Hellman* (DH o ECDH).

Requisito 3:

No se recomienda el uso de **claves pre-compartidas (PSK)**, por dos (2) motivos principales: cualquier parte que conozca la PSK podría autenticarse correctamente y, además, estas claves son vulnerables a ataques de diccionario.

Únicamente deberían utilizarse cuando sea posible asegurar que han sido generadas con la entropía suficiente para aportar la fortaleza deseada (Ej.: para sistemas del ENS categoría Alta se exigen 128 bits) y es posible renovarlas en un período inferior a su "cripto período".

[4] En IANA se puede consultar todas las *cipher suites* registradas para uso en TLS: https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4

[5] Según recomendaciones de *SOGIS Agreed Cryptographic Mechanisms v1.1* https://www.sogis.eu/uk/supporting_doc_en.html

No se recomienda el uso de métodos *Key Exchange* que no proporcionen FS (***Forward Secrecy***). Esta característica impide que se descifre el contenido de la comunicación anterior si se compromete la clave actual de la sesión. Por ello, a pesar de que supone incrementos en coste computacional, es altamente recomendable.

DH o ECDH estáticos no proporcionan FS, ya que las claves públicas/privadas DH son siempre las mismas (las asociadas al certificado). Por lo tanto, el secreto pre-compartido generado por cliente y servidor será siempre el mismo en todas las sesiones. Si un atacante descubre una de las claves privadas, puede obtener el secreto pre-compartido y lograr descifrar con él la comunicación anterior, actual y futura entre cliente y servidor.

RSA como método de *Key Exchange* tampoco proporciona FS. El cliente utiliza la clave pública RSA del servidor obtenida del certificado, para cifrar el secreto pre-compartido. Una vez que un atacante descubra la clave privada RSA del servidor, siempre podrá descifrar el secreto.

Los únicos métodos *Key Exchange* que proporcionan FS son DHE y **ECDHE**, es decir, DH y ECDH *Ephemeral*. En este caso, las claves DH son generadas de forma dinámica por el cliente y servidor en cada sesión.

Requisito 4:

Se recomienda el uso de los métodos de ***Key Exchange* DHE o ECDHE**, ya que proporcionan ***Forward Secrecy (FS).***

En el caso de **ECDHE**, el uso de curvas elípticas (EC) en TLS se describe en la RFC 8422. El protocolo utiliza dos (2) extensiones en los mensajes de *ClientHello* y *ServerHello*:n: *Elliptic Curves* y *Point Formats*, que se utilizan para negociar los tipos de curva elíptica y el formato de puntos (comprimido, no comprimido o ansiX962).

El cliente que propone *cipher suites* EC en el *ClientHello* debe enviar estas extensiones para indicar la curva soportada y sus parámetros. Si no lo hace, el servidor es libre de elegir la curva y los parámetros que decida, con el riesgo de que el cliente no los soporte y aborte la conexión al recibir el *ServerHello*.

Los servidores que implementen EC, deben soportar estas extensiones y escoger la curva y los parámetros de entre los enviados por el cliente.

Las curvas soportadas en TLS 1.2 se detallan en el apartado 5.1.1 de la RFC 8422 y son las que se indican en la siguiente tabla. Como se puede observar, la fortaleza que ofrecen todas estas curvas es de 128 bit o superior.

Curva	Fortaleza (bits)	RFC
secp256r1 (NIST P-256)[6]	128	RFC 8422
secp384r1 (NIST P-384)	192	RFC 8422
secp521r1 (NIST P-521)	256	RFC 8422
x25519 (Montgomery curve)	128	RFC 8422, RFC 7748

[6] Las curvas NIST se encuentran definidas en FIPS PUB 186-4: *http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf*

Curva	Fortaleza (bits)	RFC
x448 (Montgomery curve)	224	RFC 8422, RFC 7748

Tabla 3. Curvas elípticas y fortaleza

Requisito 5:

Se recomienda que la implementación de TLS 1.2 soporte el uso de **curvas elípticas (EC)**.

En el caso de **DHE**, el uso de los Grupos *Diffie-Hellman* en TLS se describe en la RFC 7919.

Cuando el cliente TLS ofrece en el mensaje *ClientHello cipher suites* con el método de *Key Exchange* DHE, el servidor no tiene forma de saber qué grupo DH es soportado y aceptado por el cliente. Por lo tanto, el servidor escoge su propio Grupo DH y envía los parámetros DH acordes con ese grupo al cliente. Si este grupo no es soportado, o no proporciona suficiente fortaleza para el cliente, este aborta la conexión.

La RFC 7919 introduce una modificación a TLS 1.2, 1.1 y 1.0 para solucionar esto. Por un lado, se define un listado de los grupos DH seleccionados para su uso en TLS, con un identificador (*Supported Goups Registry*). Por otro lado, la extensión *Elliptic Curves* pasa a usarse (se renombra) como extensión *Supported_Groups* y en ella el cliente TLS también puede enviar los Grupos DH soportados. Servidor y cliente TLS deben ser compatibles con este nuevo mecanismo definido en la RFC.

Los grupos DH soportados, junto con sus fortalezas aproximadas, se encuentran en el Apéndice A de la RFC 7919 y son:

Grupo	Longitud (bits) de los parámetros (p)	Fortaleza (bits)
ffdhe2048	2048	103
ffdhe3072	3072	125
ffdhe4096	4096	150
ffdhe6144	6144	175
ffdhe8192	8192	192

Tabla 4. Grupos DH y fortaleza

Requisito 6:

En caso de utilizar DHE, se recomienda que la implementación de TLS 1.2 soporte el uso de la extensión ***Supported_Groups***. Para sistemas de categoría MEDIA del ENS deberán utilizarse grupos que proporcionen una fortaleza superior a 112 bits, mientras que para sistemas de categoría ALTA del ENS, deberán utilizarse grupos que presenten una forltaza de 128 bits o superior.

6.1.2 *Autenticación y Firma*

Algunas *cipher suites* no proporcionan autenticación. Por ejemplo: TLS_DHE_**anon**_WITH. **No se recomienda el uso de este tipo de *cipher suites*.**

Tampoco se recomienda el uso de claves pre-compartidas (PSK), por los motivos indicados anteriormente.

Requisito 7:

Se recomienda utilizar **autenticación a través de certificados de clave pública X.509v3 del tipo RSA o ECDSA.**

No se recomienda el uso de DSA ya que presenta más vulnerabilidades que RSA o ECDSA. De hecho, en TLS 1.3 se elimina el uso de DSA.

El algoritmo de clave pública indicado en la *cipher suite* determina el tipo de clave pública del certificado del servidor y para qué se utilizará esta clave.

Por otro lado, el certificado del servidor debe ir firmado con un algoritmo de clave pública / función *hash* soportada por el cliente. Este algoritmo no tiene por qué ser el mismo que el usado para la clave pública del certificado (indicado en la *cipher suite*), aunque lo más habitual es que lo sea. El cliente envía en el *ClientHello* la extensión *signature_algorithms*, con los pares algoritmo – *hash* soportados para verificar la firma del certificado del servidor. Si no envía esta extensión, el servidor seleccionará el algoritmo de clave pública de la *cipher suite* y, por defecto, la función *hash* SHA-1.

Requisito 8:

En TLS 1.2, se recomienda el uso de la extensión ***signature_algorithms*** para indicar los algoritmos de firma soportados. En ella deberán incluirse, además del algoritmo de clave pública (RSA o ECDSA), funciones *hash* iguales o superiores a **SHA-256**.

A continuación, se incluye una tabla en la que se describe, en función del algoritmo de clave pública de la *cipher suite*, cómo debe ser el certificado del servidor y el uso de la clave.

KeyExchange Auth	Clave pública del certificado	Uso de la clave	Funcionamiento general
DHE_RSA ***ECDHE_RSA***	RSA	Firma	El servidor debe disponer de un certificado de clave pública RSA. Esta clave se utilizará para firma. Tras el envío del certificado, en el mensaje *ServerKeyExchage* el servidor le envía al cliente los parámetros *Diffie-Hellman*, firmados con la clave privada RSA del servidor. El cliente podrá verificar la firma con la clave pública RSA del servidor, obtenida del certificado. Con los parámetros *Diffie-Hellman* del servidor, el cliente podrá generar el secreto pre-compartido. Por otro lado, si el cliente no ha enviado otra combinación, el certificado irá firmado con RSA/SHA1, en la extensión *signature_algortithm*.

KeyExchange Auth	Clave pública del certificado	Uso de la clave	Funcionamiento general
ECDHE_ECDSA	ECDSA	Firma	El servidor debe disponer de un certificado de clave pública ECDSA. Esta clave se utilizará para firma. El comportamiento es similar al caso anterior utilizando ECDSA en lugar de RSA.

Tabla 5. Tipo de certificado en función de la *cipher suite*.

Respecto a la longitud de la clave pública RSA y ECDSA, a continuación, se resumen los diferentes tamaños de claves según el nivel de seguridad que proporcionan.

Fortaleza (bits)	Longitud de clave RSA	Longitud de clave ECC
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Tabla 6. Comparación de fortaleza y longitudes de claves.

Requisito 9:

No se deben usar **longitudes de clave con fortaleza inferior a 112 bits**. Para aquellos sistemas de categoría **ALTA** bajo el alcance **del Esquema Nacional de Seguridad (ENS)**, junto con aquellos sistemas que manejan información clasificada, deberán utilizarse claves que proporcionen un nivel de **fortaleza igual o superior a 128 bits**. Esto supone utilizar claves RSA de, al menos, 3072 bits y claves ECC de, al menos, 256 bits.

6.2. Cifrado

En TLS 1.2, la parte central de la *cipher suite* negociada indica el algoritmo de cifrado simétrico (*Cipher*), su longitud de clave (*KeyLength*) y su modo de operación (*mode*).

*TLS_KeyExchange_Auth_WITH_**Cipher_KeyLength_Mode**_HashFunction*

Aunque algunas *cipher suites* de TLS 1.2 utilizan TDEA (3DES). Se recomienda el uso de AES al haberse demostrado que es un algoritmo mucho más robusto y eficiente.

La mínima longitud de clave de AES en las *cipher suites* de TLS es 128 bits, lo cual se considera una fortaleza suficiente incluso para un categoría ALTA del ENS.

Respecto al modo de operación de AES, TLS 1.2 soporta los modos CBC, GCM y CCM.

AES en modo *Cipher Block Chaining (AES_CBC)* es el modo más tradicional donde, antes de ser cifrado, a cada bloque de texto se le aplica una operación XOR con el bloque previo ya cifrado. Además, para hacer cada mensaje único se debe usar un vector de inicialización en el primer bloque.

AES en modo *Galois Counter (AES-GCM)*, AES en modo *Counter* y CBC MAC (AES-CCM) son nuevos

modos de operación **AEAD** (*Authenticated Encryption with Additional Data*)[7] que se introducen en TLS 1.2. Proporcionan, además de la confidencialidad, autenticidad de origen e integridad sin necesidad de hacer uso de algoritmos adicionales.

Requisito 10:

En TLS 1.2, se recomienda el uso de algoritmo de cifrado AES en los modos AEAD, como AES_GCM y AES_CCM.

6.3. Integridad y Autenticidad de Mensajes

En TLS 1.2, la parte final de la *cipher suite* negociada indica la función *hash*:

*TLS_KeyExchange_Auth_WITH_Cipher_KeyLength_Mode_**HashFunction***

Dicha función *hash* se utilizará para:

- El cálculo de los valores MAC que *Record Protocol* añadirá a los paquetes enviados entre el cliente y servidor TLS para verificar la integridad y autenticidad de origen.
- La función PRF (*pseudorandom function*) que se usará para generar el material de clave.

Un código de autenticación de mensaje o MAC (*Message Authentication Code*), es un cálculo del *checksum* de un mensaje, utilizando para ello una función segura y una *Secret Key*. Cuando dicha función es una función *hash*, se denomina HMAC (*keyed-hash message authentication code).* Un MAC proporciona al mensaje: integridad, ya que una variación en el mensaje producirá un MAC distinto y autenticidad de origen, ya que solo el origen auténtico dispondrá de la clave correcta (*client_write_MAC_key o server_write_MAC_key*).

En TLS 1.2 se soportan: *hmac_md5, hmac_sha1, hmac_sha256, hmac_sha384, hmac_sha512.*

MD5 y SHA1 se consideran funciones *hash* inseguras y sujetas a colisiones[8]. La guía CCN-STIC-807 no permite el uso de MD5 y únicamente admite el uso de SHA1 de forma temporal cuando se emplea en HMAC.

Requisito 11:

En TLS 1.2, se recomienda el uso de *cipher suites* con funciones *hash* de la familia **SHA-2 o superior.** Por norma general**,** no se recomienda utilizar **SHA-1**, aunque se permite de forma temporal únicamente con HMAC. MD5 se considera inseguro, por lo que no debe usarse.

6.4. *Cipher suites* recomendadas

Teniendo en cuenta todas las consideraciones anteriores:

[7] Existen varias RFCs que definen el uso de estos algoritmos en TLS 1.2: RFC 5288 (AES *Galois Counter Mode* (GCM) *Cipher Suites for TLS*), RFC 6655 (AES-CCM *Cipher Suites for TLS*), RFC 7251 - AES-CCM *Elliptic Curve Cryptography* (ECC) *Cipher Suites for TLS*).

[8] Una colisión se produce cuando distintos mensajes producen el mismo valor *hash*.

Requisito 12:

Las *cipher-suites* recomendadas en TLS 1.2 son*:*

- *TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256*
- *TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384*
- *TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256*
- *TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384*
- *TLS_DHE_RSA_WITH_AES_128_GCM_SHA256*
- *TLS_DHE_RSA_WITH_AES_256_GCM_SHA384*

6.5. TLS 1.3

TLS 1.3 introduce varios cambios relacionados con la criptografía.

- Se eliminan los algoritmos simétricos considerados obsoletos y únicamente se utilizan algoritmos **AEAD** (*Authenticated Encryption with Associated Data*).
- Se elimina el uso de la función *hash* SHA 1.
- Se elimina el uso de los métodos de *Key Exchange* estáticos como RSA y DH o ECDH. Todos los mecanismos de *Key Exchange* empleados proporcionarán *Forward Secrecy.*
- Los algoritmos de curva elíptica (EC) pasan a formar parte de la especificación base de TLS 1.3. Se añaden también nuevos algoritmos de curva elíptica para firma como EdDSA (*Edwards-Curve Digital Signature Algorithm),* definido en la RFC 8032.
- Se rediseña el cálculo del material de claves, pasando a utilizar primitivas *HKDF (HMAC-based Extract-and-Expand Key Derivation Functions).*

En TLS 1.3, el concepto de *cipher suite* cambia. Se separan los mecanismos de *Key Exchange* y Autenticación del algoritmo de protección AEAD y la función *hash*.

Las ***cipher suites*** se siguen negociando en el *Handshake* (mensajes *ClientHello* y *ServerHello*) y tienen el siguiente formato: **TLS_AEAD_HASH**, donde:

- AEAD es el algoritmo AEAD usado para cifrado e integridad.
- HASH es la función *hash* usada para la derivación de claves con la función HKDF.

Cualquiera de las cinco (5) *cipher suites* soportadas por TLS 1.3 reúne la fortaleza necesaria para sistemas bajo el alcance del ENS, a pesar de que actualmente el algoritmo CHACHA20 no es uno de los contemplados en la CCN-STIC-807, al no estar, por el momento, entre los aprobados formalmente por SOGIS.

Requisito 13:

En TLS 1.3, se recomienda hacer uso de cualquiera de las *cipher suites* que ofrece el protocolo, dado que ofrece la fortaleza requerida para un sistema bajo el alcance del ENS.

Los métodos de Key Exchange soportados en TLS 1.3 son:

- Solo clave precompartida: PSK.

- *Diffie Hellman Ephemeral*, sobre campos finitos (DHE) o sobre curvas elípticas (ECDHE).
- Clave precompartida con Diffie-*Hellman Ephemeral*: PSK_(EC)DHE.

Al igual que en TLS 1.2, no se recomienda el uso de PSKs, y los métodos de Key Exchange recomendados son DHE o ECDHE.

El método de *Key Exchange* se negocia en el *Handshake* (mensajes *ClientHello* y *ServerHello*) utilizando nuevas extensiones:

a) Si el método *Key Exchange* es PSK o PSK_(EC)DHE, se utilizan las extensiones:
   - Extensión *PSK_Key_Exchange_modes*. Esta extensión se utiliza para indicar cuál de los métodos es: PSK o PSK_DHE o PSK_ECDHE.
   - Extensión *Pre-shared_key*. Esta extensión contiene la PSK.

b) Si el método *Key Exchange* es DHE o ECDHE:
   - Extensión *Supported_Groups*. Esta extensión contiene, en el caso de ECDHE, el listado de las curvas EC soportadas. En el caso de DHE, el listado de los Grupos DH soportados.

     TLS 1.3 soporta los siguientes grupos y curvas EC:
     - Curvas: *secp256r1, secp384r1, secp521r1, x25519* y *x448* (ver tabla 3).
     - Grupos: *ffdhe2048, ffdhe3072, ffdhe4096, ffdhe6144* y *ffdhe8192* (ver tabla 4).
   - Extensión *Key_Share.* En función de los *Supported_Groups*, se envían los parámetros DHE o ECDHE calculados. Por ejemplo, en el caso DHE, esta extensión contiene los parámetros (p, g) y la clave pública DH para cada Grupo DH enviado en *Supported_groups*.

El servidor selecciona, de entre todas las opciones enviadas por el cliente en *ClientHello*, los valores finales a utilizar en el *Key Exchange*. En el *ServerHello*, envía los valores correspondientes en las mismas extensiones. De esta forma, tras el intercambio de los mensajes iniciales de *ClientHello* y *ServerHello*, cliente y servidor ya disponen de la información necesaria para generar el secreto pre-compartido.

Requisito 14:

Al igual que para TLS 1.2, en TLS 1.3 no se debe usar el grupo **ffdhe2048**, ya que proporciona una fortaleza inferior a 112 bits. Tampoco se debe utilizar *ffdhe3072* para sistemas de categoría ALTA del ENS, por proporcionar una fortaleza inferior a 128 bits.

Respecto a los **algoritmos de clave pública** para firma, se elimina el uso de DSA y se utiliza la implementación de RSA *Probabilistic Signature Scheme* (RSASSA-PSS).

En los mensajes *ClientHello* y *ServerHello* se envían dos (2) extensiones para negociar los algoritmos de clave pública para firma:

- Extensión *signature_algorithms*. A través de esta extensión el cliente indica los algoritmos de firma que soporta.

- Extensión *signature_algorithms_cert*. A través de esta extensión el cliente indica los algoritmos de firma que soporta, concretamente para la verificación del certificado (firma de la CA).

Los algoritmos de firma soportados por TLS 1.3 están en el apartado 4.2.3 de la RFC 8446.

Respecto a las longitudes de clave, aplica la recomendación 9.

7. CERTIFICADOS

Un **certificado digital** es un documento electrónico que contiene los datos identificativos de la entidad que lo presenta y su clave pública asociada. El certificado es emitido y firmado por una entidad acreditada que es reconocida como Autoridad de Certificación (CA).

TLS hace uso de certificados X.509v3 (RFC 5280) para la autenticación.

El estándar RFC 5280 define la sintaxis de los certificados. Los principales campos en su estructura son los siguientes:

- **Versión.** Especifica cuál de las tres versiones de X.509 se aplica en este certificado.
- **Número de serie.** Es un número que identifica el certificado de forma única, emitido por la CA.
- **Algoritmo de firma.** Algoritmo usado por la CA para firmar el certificado (algoritmo de clave pública – Función *Hash*).
- **Emisor.** Identidad de la CA emisora del certificado.
- **Periodo de validez.** Periodo de tiempo durante el cual el certificado es válido (contiene la fecha de inicio y fin de validez).
- **Sujeto.** Identidad del dueño del certificado.
- **Clave pública.** Clave pública del dueño del certificado y el algoritmo asociado.
- **Firma digital de la CA.** Firma digital realizada con la clave privada de la CA y el algoritmo especificado en "Algoritmo de firma".
- **Extensiones.** Información adicional relacionada con el uso y gestión del certificado. Entre ellas, la huella digital del certificado.

Actualmente, es muy común encontrar sitios que han instalado certificados **autofirmados**. En este tipo de certificados el sujeto y emisor son la misma entidad. Estos no han sido validados por ninguna CA y pueden ser creados por cualquier persona lo que limita el nivel de seguridad frente a un certificado firmado por una CA.

Requisito 15:

Deberán utilizarse certificados emitidos por una **Autoridad de Certificación** (CA), en lugar de certificados **autofirmados**.

7.1. Cadena de Confianza

Un modelo de **cadena de confianza** es necesario para garantizar la procedencia y autenticidad de un certificado digital. Este modelo establece una relación entre certificados, que permite asegurar que el certificado final del servidor o del cliente, ha sido emitido por una CA de confianza.

En el modelo de cadena de confianza jerárquico, existe una **CA raíz (*root CA*)** cuyo certificado es autofirmado. Esta CA raíz emite y firma **Certificados Subordinados** o **Intermedios** (*Sub-CA*). Son estos últimos los que emiten y firman el certificado final del suscriptor (cliente o servidor).

Los **Certificados Intermedios** se emiten para proteger al máximo el certificado Raíz (root CA), ya que, si un tercero tuviese la clave privada del *Root CA*, se verían comprometidos todos los certificados emitidos por dicha CA. Al utilizar los certificados intermedios, se mantiene el certificado raíz detrás de varias capas, para hacer sus claves más inaccesibles.

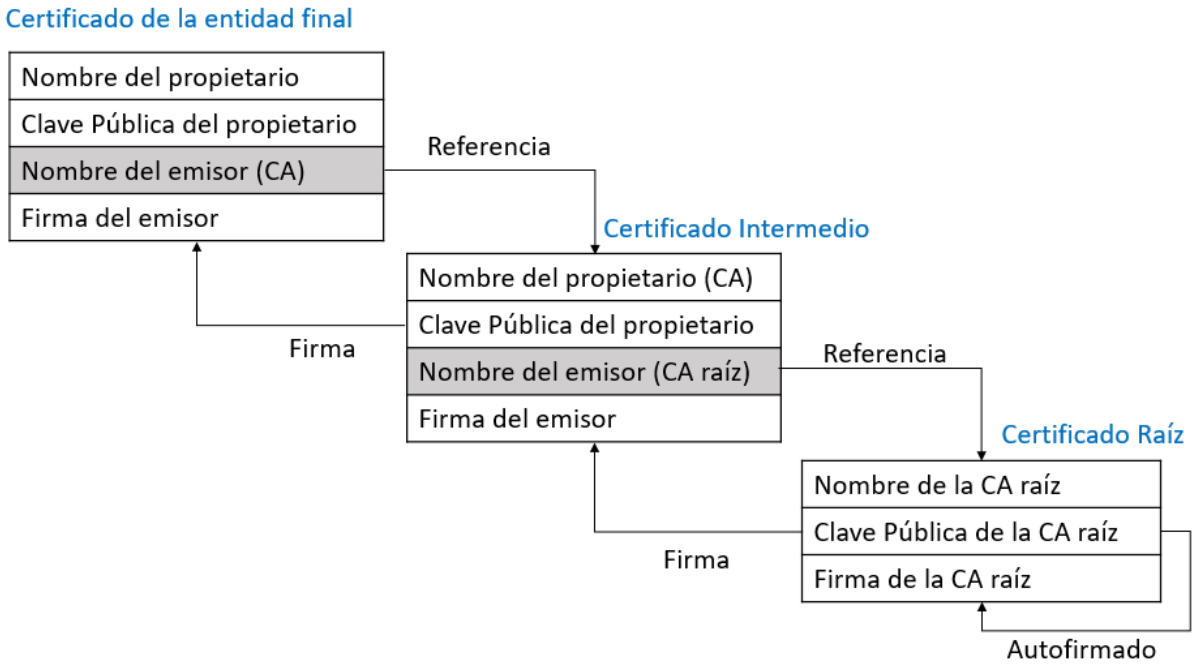

Figura 4. Cadena de Certificados

El servidor TLS envía la cadena de certificados al cliente TLS a través del mensaje *ServerCertificate,* para que este pueda realizar la validación. El primero de la lista es el certificado del servidor (entidad final). A continuación, irán los certificados intermedios que certifican directamente al que les precede (*Sub-CA*) hasta el certificado de la CA raíz (*root CA*).

El cliente también puede enviar un certificado al servidor (autenticación mutua) cuando este lo requiere a través del mensaje *CertificateRequest*. El cliente debe responder con un mensaje *Client Certificate* que incorporará la cadena de certificados.

En caso de que el cliente no disponga de un certificado válido y adaptado a los requisitos solicitados por el servidor, responderá el mensaje *ClientCertificate* con una lista de certificados vacía. El servidor podrá continuar la conexión sin la autenticación de cliente, o rechazarla provocando un error en el *Handshake*.

Requisito 16:

Se recomienda la **autenticación mutua cliente-servidor**, de forma que el servidor también autentique al cliente a través de certificado.

7.2. Validación de Certificados

Una vez que el cliente TLS recibe el certificado del servidor TLS, debe validarlo. Lo mismo sucede cuando es el servidor el que recibe el certificado de cliente.

El objeto de la validación del certificado es confirmar que la clave pública del certificado pertenece, de forma unívoca, al dueño del certificado (*subject*). El uso de esta clave pública, no obstante, estará sujeto a las restricciones especificadas en el certificado a través de variables y extensiones.

La RFC 5280 especifica un algoritmo para la validación del certificado que comprende la validación de toda la cadena de certificados (***Certificate Path Validation***). Este algoritmo se compone de múltiples etapas. Algunas de las validaciones más importantes que se realizan, son las siguientes:

- El primer certificado de la cadena será el de la entidad final (cliente o servidor TLS).
- El último certificado de la cadena será en de la CA raíz (***root CA***). Este certificado es autofirmado.
- El sujeto (*subject*) de un certificado X, será el emisor (*issuer*) del siguiente certificado X+1.
- Todos los certificados de una CA deben tener la extensión *basicContraints* a TRUE. Esto indica que, efectivamente, pertenecen a una CA y su clave pública puede ser usada para verificar la firma de otros certificados.
- Ningún certificado aparecerá más de una vez en la cadena.
- Para cada certificado se realizarán las siguientes comprobaciones:
  - Verificación de la firma usando la clave pública de la CA emisora, y el algoritmo de firma indicado.
  - Verificación de que el certificado está dentro del periodo de validez.
  - Verificación de que el certificado no está revocado.

Requisito 17:

Deberá llevarse a cabo **la validación del certificado** del cliente y servidor TLS, utilizando una implementación que cumpla la RFC 5280, donde se verifique:

- El estado de revocación del certificado a través de listas CRLs (*Certificate Revocation Lists*) según se define en la RFC 5280 apartado 6.3, o a través del protocolo OCSP (*Online Certificate Status Protocol*) especificado en la RFC 6960.
- La cadena de certificados. Deberán ser Al menos tres certificados: el certificado de trust CA raíz (*root CA*), el certificado de una CA subordinada (*Sub-CA*) y el certificado final del cliente o servidor TLS.

Requisito 18:

El servidor y cliente TLS deberán **rechazar la conexión TLS** cuando:

- El certificado no haya superado el proceso de validación.
- No pueda verificarse el estado de revocación del certificado (por ejemplo, debido a un error de red que no permite descargar la CRL o la conexión OCSP). En caso de que el entorno operativo no permita realizar dicha verificación, el establecimiento de la conexión deberá ser aprobado por un administrador.

Requisito 19:

Se recomienda el uso de la extensión ***extendedKeyUsage*** en el certificado del servidor y del cliente TLS. Esta extensión deberá tomar el valor: *Server Authentication purpose (id-kp 1)* o *Client Authentication purpose (id-kp 2*) respectivamente. El proceso de validación del certificado deberá incluir la verificación de este campo.

Finalmente, indicar que ni el certificado ni las CRLs deben mantenerse en secreto. El acceso no autorizado a ambos no supone ningún problema de seguridad. **El factor de seguridad crítico y lo que debe protegerse con las medidas apropiadas, es la clave privada asociada al certificado.**

En el Anexo B se recoge un ejemplo de cómo crear un certificado X.509v3 con OpenSSL.

8. TLS 1.3

La especificación de TLS 1.3 (RFC 8446) fue publicada por IETF en agosto 2018 (RFC 8446). TLS 1.3 representa un cambio significativo y tiene por objetivo abordar las amenazas que han surgido a lo largo del tiempo desde la especificación de TLS 1.2. **A día de hoy, TLS 1.3 es la versión de TLS considerada más segura**. Sin embargo, no es compatible con versiones anteriores, por lo que el uso de TLS 1.3 puede requerir un tiempo de adaptación para los sistemas y aplicaciones.

Los principales cambios de TLS 1.3 son los siguientes:

- Cambios relacionados con los **algoritmos y funciones criptográficas**, ya indicados en el Apartado 6.5.
- **Se rediseña el *Handshake* significativamente** para ser más consistente y eficiente. Se eliminan mensajes superfluos como el *ChangeCipherSpec*.
- **Todos los mensajes posteriores a *ServerHello* van cifrados.** Esto es posible porque, tras el intercambio inicial de los mensajes *ClientHello* y *ServerHello*, cliente y servidor tienen la información suficiente para el cálculo del secreto pre-compartido. En realidad, no solo se cifran los mensajes posteriores al *ServerHello* sino también parte de los parámetros del servidor enviados en *ServerHello*. Esto se hace a través del nuevo mensaje "*EncrytedExtensions*". Solo se envían en claro la versión negociada (extensión *supported_version*) y los parámetros que necesita el cliente para el cálculo del secreto (extensiones *pre-shared key* o *key_share*).
- **No se utiliza compresión**. El campo donde iba la lista de algoritmos de compresión debe ir con valor cero.

- **Negociación de versión:** En TLS 1.2 el cliente envía en *ClientHello* la versión "preferida" de TLS, en el campo *client_version*. El servidor, si no soporta esa versión, rechaza el *ClientHello*. Esto en realidad no es una negociación de versión. En TLS 1.3 el cliente envía una lista de las versiones soportadas en una nueva extensión (*supported_versions*). El servidor puede seleccionar la versión TLS, lo cual sí es una negociación de versiones.
- **Protección frente a ataques *downgrade*.** TLS 1.3 implementa un mecanismo de protección frente ataques *downgrade*. Cuando un servidor TLS 1.3 selecciona una versión TLS 1.2 o TLS 1.1 en respuesta a un ClientHello, inserta un valor fijo en los últimos 8 bytes del *random.server*. Cuando el cliente TLS 1.3 recibe en el *ServerHello* la versión TLS 1.2 o 1.1, debe chequear esos 8 últimos bytes del *random.server* para ver si, efectivamente, corresponden al valor asignado a TLS 1.2 o TLS 1.1. Si no corresponden, significa que no era la intención del servidor negociar esas versiones y se trata de un ataque *downgrade*. Además de este, existen también otros mecanismos que se utilizan en mensajes posteriores del *Handshake*, y que permiten detectar si los mensajes no corresponden a la versión negociada.
- Se añade un **modo "*zero rount-trip time*" (0-RTT)** que permite enviar algunos datos de aplicación tempranos (*early data*) ya en la primera ronda, cuando cliente y servidor comparten una PSK (obtenida externamente o con una negociación previa). El cliente usa la PSK para autenticar el servidor y para cifrar estos datos iniciales. Hay que considerar que las propiedades de seguridad de estos datos enviados en 0-RTT es mucho más débil que los datos enviados posteriormente tras el *Handshake*.

9. RESUMEN

A continuación, se incluye una tabla con las recomendaciones realizadas a lo largo de este documento. **Se ha indicado con un asterisco (*) las que son de obligado cumplimiento para aquellos sistemas bajo el alcance del Esquema Nacional de Seguridad (ENS).** Aunque haya algunas recomendaciones que no son de obligado cumplimiento para el ENS, se resalta que deben ser tenidas en cuenta para hacer uso del protocolo TLS de forma segura.

Nº	Ámbito		Resumen de la Recomendación	TLS 1.2	TLS 1.3
1 (*)	General		Usar la versión del protocolo **TLS 1.2 o superior**. Deshabilitar versiones anteriores a TLS 1.2.	Aplica	Aplica
2 (*)	VPN SSL		Para Sistemas ENS: Utilizar **productos Cualificados** de la familia Redes Privadas Virtuales SSL dentro del Catálogo de Productos de Seguridad TIC (**CPSTIC**). Para Sistemas que manejen información clasificada: Utilizar **productos Aprobados** dentro del Catálogo de Productos de Seguridad TIC (CPSTIC)	Aplica	Aplica
3	Criptografía	*Key Exchange*	**No** utilizar claves pre-compartidas (**PSK**).	Aplica	Aplica
4		*Key Exchange*	Usar **DHE o ECDHE**. No usar RSA ni DH o ECDH estáticos.	Aplica	No Aplica
5		*Key Exchange*	Soporte de **Curvas Elípticas**.	Aplica	No Aplica
6		*Key Exchange*	Soporte de la extensión ***Supported_Groups***.	Aplica	No Aplica
6/14 (*)		*Key Exchange*	**No** utilizar el grupo DH **ffdhe2048**. Para **categoría ALTA del ENS no** utilizar el grupo DH ***ffdhe3072*** o inferior.	Aplica	Aplica
3		Autenticación	**No** utilizar claves pre-compartidas (**PSK**).	Aplica	Aplica
7		Autenticación	Usar **certificados X.509v3 de tipo RSA o ECDSA**.	Aplica	Aplica
8		Firma	Usar la extensión ***signature_algorithms***. Emplear algoritmos de firma **RSA o ECDSA**, con funciones **SHA-2 o superior.**	Aplica	No Aplica
9 (*)		Autenticación y Firma	Usar claves públicas con fortaleza superior a 112 bits. Para **categoría ALTA del ENS** o sistemas clasificados, usar una fortaleza superior a 128 bits (**RSA ≥ 3072 bits, ECDSA ≥ 256 bits**).	Aplica	Aplica
10		Cifrado	Usar AES en **modos GCM** o **CBC**.	Aplica	No Aplica
11 (*)		Hash	Usar funciones *hash* **SHA-2 o superior**.	Aplica	No Aplica

Nº	Ámbito		Resumen de la Recomendación	TLS 1.2	TLS 1.3
12		*Cipher Suites*	Hacer uso de alguna de las siguientes *cipher suites* (TLS 1.2): *TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256* *TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384* *TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256* *TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384* *TLS_DHE_RSA_WITH_AES_128_GCM_SHA256* *TLS_DHE_RSA_WITH_AES_256_GCM_SHA384*	Aplica	No Aplica
13		*Cipher Suites*	Hacer uso de cualquiera de las siguientes *cipher suites que ofrece* (TLS 1.3):	No Aplica	Aplica
15	Certificados		**No** utilizar certificados **auto-firmados**, sino emitidos por una CA de confianza.	Aplica	Aplica
16	Certificados		Usar **autenticación mutua** entre cliente y servidor TLS, de forma que no solo el servidor proporcione un certificado, sino también el cliente.	Aplica	Aplica
17 (*)	Certificados		Implementar **validación de certificados** según RFC 5280.	Aplica	Aplica
17 (*)	Certificados		Implementar **verificación del estado de revocación** del certificado a través de listas CRLs o protocolo OCSP.	Aplica	Aplica
17 (*)	Certificados		Soportar una **cadena** de, al menos, **tres (3) certificados.**	Aplica	Aplica
18 (*)	Certificados		Servidor y cliente TLS deben **rechazar** la conexión TLS cuando el certificado **no haya superado** el proceso de **validación**.	Aplica	Aplica
18 (*)	Certificados		Servidor y cliente TLS deben **rechazar** la conexión TLS cuando **no pueda verificarse** el estado de revocación del certificado. En caso de que el entorno operativo no lo permita, el establecimiento de la conexión deberá ser **aprobado por un administrador**.	Aplica	Aplica
19	Certificados		Usar la extensión ***extendedKeyUsage*** para los propósitos de *Server Authentication purpose* o *Client Authentication purpose.* El proceso de validación del certificado deberá **verificar este campo**.	Aplica	Aplica

Tabla 7. Resumen de recomendaciones TLS

10. ANEXO A. EJEMPLO DE CONFIGURACIÓN TLS EN SERVIDOR WINDOWS

A continuación, se muestra a modo de ejemplo, cómo configurar TLS en un servidor Windows Server 2016.

10.1. Configurar la versión TLS 1.2

Para configurar la versión de TLS, se escribe en el buscador de Windows *regedit* (herramienta que permite editar el registro del sistema operativo), y se pulsa ***Enter*.** Se abrirá el *Editor del Registro*. Dentro de este editor, se busca la siguiente clave de registro:

Equipo\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL

Dentro de esta clave, se ha de entrar en ***Protocols*** para poder añadir la clave de registro de TLS 1.2**.** Inicialmente, la carpeta ***Protocols*** tiene solo la información del protocolo SSL 2.0, tal y como se muestra en la imagen siguiente:

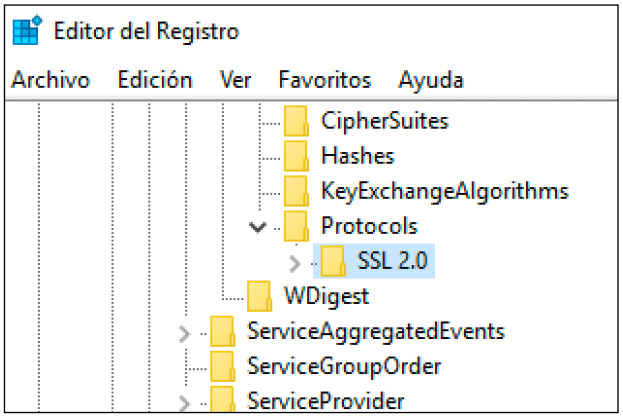

Figura 5. Claves de registro SSL/TLS por defecto

En esta carpeta hay que añadir diferentes claves, según el protocolo que se quiera activar. Para añadir la clave de **TLS 1.2** se ha de pulsar sobre *Protocols* con el botón derecho, escoger la opción *Nuevo > Clave*, y nombrar la carpeta como *TLS 1.2*. Dentro de la nueva carpeta, se deben crear otras dos subcarpetas: *Client* y *Server*. Para hacer esto, simplemente hay que pulsar con el botón derecho sobre *TLS 1.2*, y escoger la opción *Nuevo > Clave.*

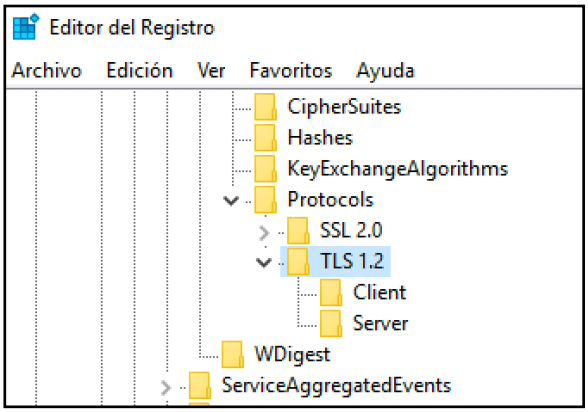

Figura 6. Creación clave de registro para TLS 1.2

Sobre la carpeta *Client*, se pulsa con el botón derecho y se escoge la opción *Nuevo > Valor de DWORD (32 bits)*. Una vez creado, se le da el nombre de *DisabledByDefault*. Se pulsa botón

derecho sobre > Modificar > y se le da el valor 0. Se hace la misma operación para crear otro *Valor de DWORD (32 bits)* al que se le llamará *Enabled* y se le da el valor 1.

Se crean los dos mismos *valores de DWORD* en la carpeta Server.

Como resultado, se dispone de ambas carpetas (*Client* y *Server*) con los mismos dos (2) valores:

DisabledByDefault = 0

Enabled = 1

Esto significa que se ha activado TLS 1.2, cuando el equipo *Windows* funcione como cliente o como servidor TLS.

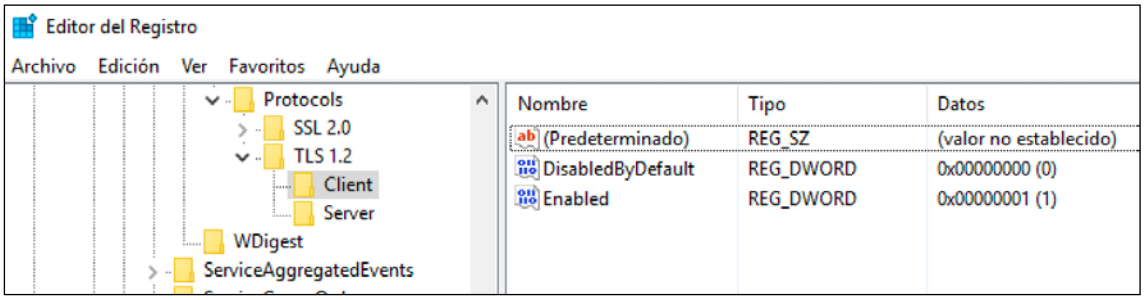

Figura 7. Configuración de clave de registro para TLS 1.2

Una vez activado TLS 1.2, se deberá **desactivar el resto de protocolos**. Para esto se han de crear las claves de dichos protocolos (SSL 3.0, TLS 1.0 y TLS 1.1), de la misma forma que como se ha realizado con TLS 1.2. Por lo tanto, los valores que se han de añadir a los protocolos que se desean desactivar serán los siguientes:

DisabledByDefault = 1

Enabled = 0

Una vez creadas las claves de los anteriores protocoles, para poder actualizar los cambios en el equipo, es necesario **reiniciar**. Para ver si está funcionando correctamente TLS 1.2, accedemos a un sitio web seguro, como puede ser *https://www.google.es* . Una vez dentro del sitio web, se pulsa botón derecho > *Propiedades*, para observar las propiedades de la conexión que se ha realizado:

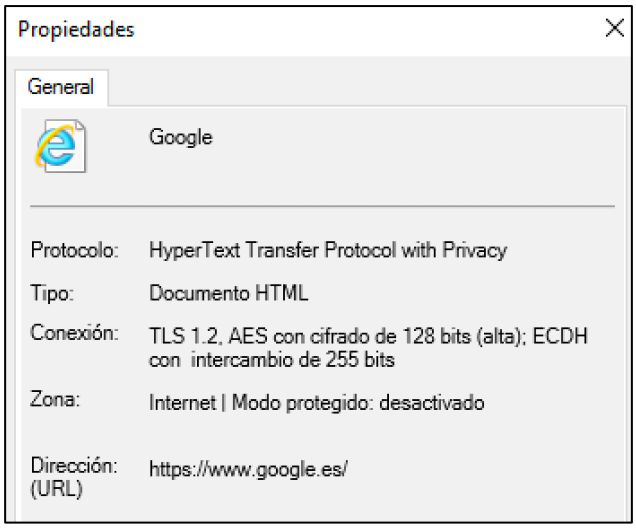

Figura 8. Verificación del funcionamiento de TLS 1.2

10.2. Configurar las Cipher Suites de TLS 1.2

Una vez que la versión de TLS 1.2 ha sido añadida, se procede a seleccionar las ***cipher suites***. Para ello, se debe escribir "*gpedit.msc*" en el buscador de la máquina, y pulsar **Enter**. Se abrirá el editor de directivas de grupo local.

Una vez dentro, acceder a *Configuración del Equipo > Plantillas Administrativas > Red > Opciones de Configuración SSL*. Seguidamente, se ha de hacer doble *click* sobre ***Orden de conjuntos de cifrado SSL***.

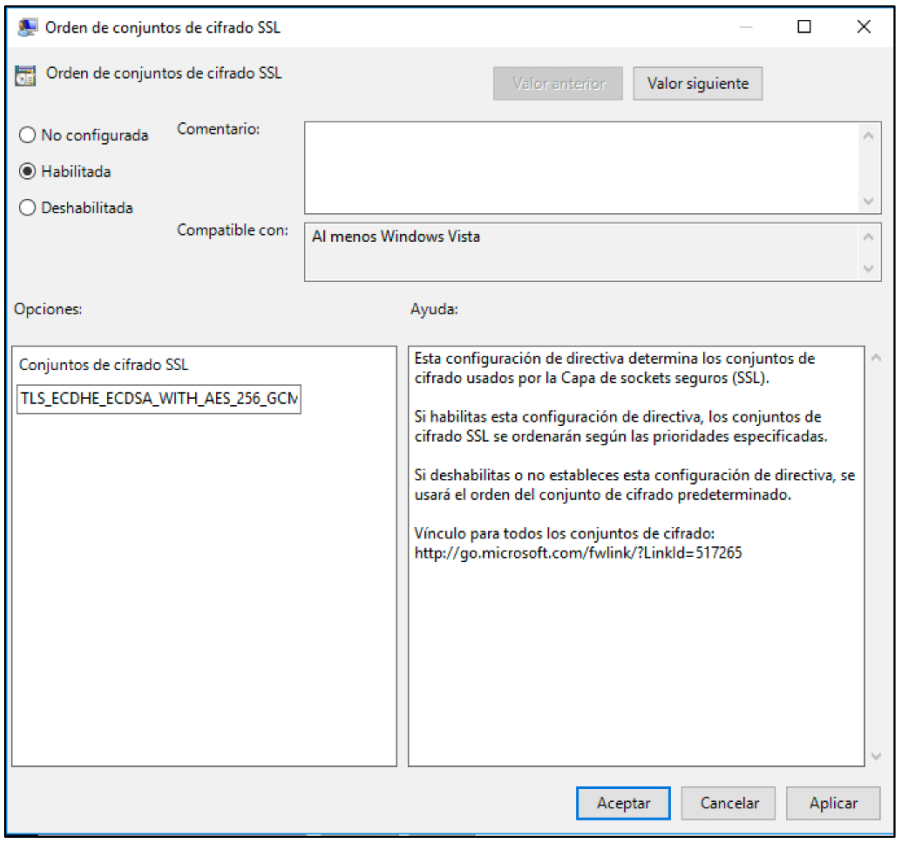

Figura 9. Configuración de *cipher suites*

En esta pantalla se escriben las *cipher suites* y se selecciona la opción *Habilitada*. Se ha de tener en cuenta que deben escribirse correctamente, sin espacios ni saltos de línea. Se debe tener en cuenta también, el orden de prioridad en el que se coloquen las *cipher suites*. Cuando se tenga la lista finalizada, se pulsa **Aceptar** y se **reinicia** el equipo.

11. ANEXO B. EJEMPLO DE CREACIÓN DE UN CERTIFICADO CON OPENSSL

A continuación, a modo de ejemplo, se describe el proceso de creación de una cadena de certificados usando la herramienta gratuita *Openssl.*

Para comenzar, se crea el **Certificado Raíz** (*Root-CA*), de nuestra Autoridad de Certificación simulada, y se obtiene la clave privada correspondiente, la cual será empleada para firmar los diferentes certificados que se generen.

1. Se genera la clave privada RSA de 3072 bits (128 bits de fortaleza).

   *$ sudo **openssl genrsa** -out rootCA.key 3072*

2. Con la clave privada en el fichero *rootCA.key*, se genera el Certificado Raíz de nuestra CA simulada. Es autofirmado, y se firmará con la clave privada generada. El formato de certificado es .pem, con una validez de 365 días.

   *$ sudo **openssl req –x509** –key rootCA.key –days 365 –out rootCA.pem*

   La opción *–x509* indica que se cree un certificado autofirmado y no una petición de certificado (CSR).

3. Se genera la clave privada del equipo final para el que queremos el certificado.

   *$ sudo **openssl genrsa** -out device.key 3072*

4. Se genera una solicitud de certificado CSR a la CA anterior.

   *$ sudo **openssl req –new** –key device.key –out device.csr*

   La opción *–new* indica que se cree una CSR nueva.

5. Se pasa la petición CSR a la CA para obtener el certificado para el equipo. Pero antes, como paso previo se crea un fichero fichero *extensiones.txt*, que pasaremos como argumento a la CA. En este fichero añadimos las **extensiones** que se desean incluir en el certificado. En este caso, queremos incluir la extensión de *ExtendedKeyUsage* para la autenticación de cliente TLS, por lo que el valor de esa extensión debe ser *clientAuth*.

Figura 9. Creación de extensiones a incluir en el certificado

   El certificado se llamará *device.crt.*

   *$ sudo **openssl x509** –req –in device.csr –extensions extensiones –extfile extensiones.txt –CA rootCA.pem –CAkey rootCA.key -out device.ctr –days 365*

6. Una vez obtenido el certificado del dispositivo, podemos comprobar que, efectivamente, ha sido firmado por la *CA*:

   $ sudo openssl x509 –in device.ctr –text -noout

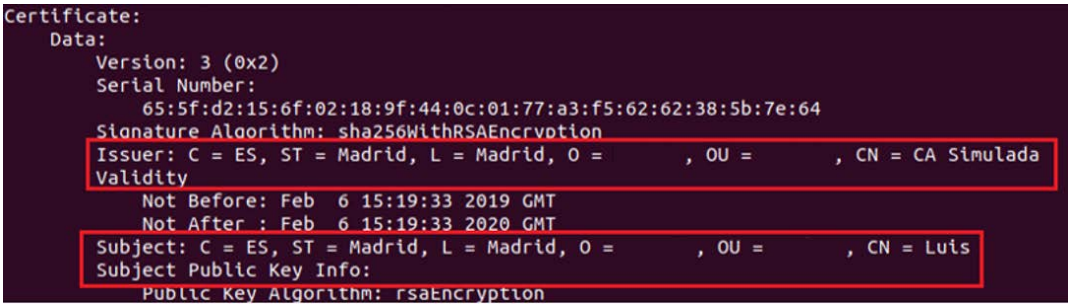

Figura 10. Campos del certificado

7. Asimismo, se puede comprobar que el certificado cuenta con la extensión *ExtendedKeyUsage* con valor ***TLS Client Authentication*.**

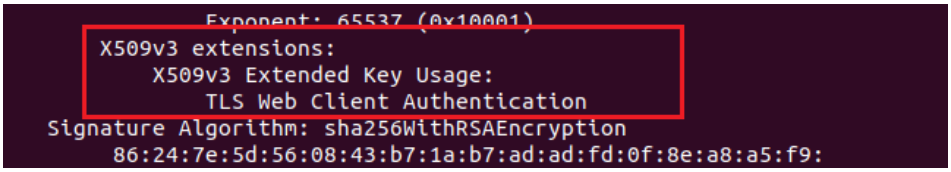

Figura 11. Extensiones del certificado

Contacto

Correo electrónico CCN-PYTEC	**ccn-pytec@cni.es**
Twitter	**@CCNPYTEC**
LinkedIn	**https://www.linkedin.com/company/CCN-PYTEC**
Catálogo CPSTIC	**Enlace web**

El Departamento de Productos y Tecnologías de Seguridad TIC del Centro Criptológico Nacional (CCN-PyTec) promueve el desarrollo, la evaluación, la certificación y el uso de productos para garantizar la seguridad de los sistemas de tecnologías de la información y la comunicación.

Praeventio sit vincere