Universidad Zaragoza
1542

Trabajo Fin de Grado

Cifrado seguro mediante uso protocolo SRTP para proteger comunicaciones de voz de una infraestructura de comunicación de misión crítica (TETRA)

Strong encryption using the SRTP protocol to protect voice communications from a mission-critical communication infrastructure (TETRA)

Autor

Javier García Cantarero

Director

José Ángel Martínez Luengo

Ponente

Álvaro Alesanco Iglesias

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

2023

Índice

1. Lista de acrónimos y siglas.

ACELP	–	Algebraic code-excited linear prediction
AEAD	–	Authenticated Encryption with Associated Data
AES	–	Advanced Encryption Standard
Códec	–	Codificador-Decodificador
DDS	–	Data Distribution Service
DTLS	–	Datagram Transport Layer Security
ECDHE	–	Elliptic Curve Diffie-Hellman Ephemeral
ECDSA	–	Elliptic Curve Digital Signature Algorithm
GCM	–	Galois/Counter Mode
GVOIP	–	Global Voice Over Internet Protocol
ICE	–	Interactive Connectivity Establishment
IETF	–	Internet Engineering Task Force
IKE	–	Internet Key Exchange
IP	–	Internet Protocol
IPSec	–	Internet Protocol Security
ITU-T	–	Unión Internacional de Telecomunicaciones-Telecomunicación
MAC	–	Media Access Control
NAT	–	Network Address Translation
PCMA	–	Pulse Code Modulation A-law
PTT	–	Push To Talk
RSA	–	Rivest, Shamir y Adleman
RTCP	–	Real-time Transport Control Protocol
RTP	–	Real-time Transport Protocol
SDP	–	Session Description Protocol
SHA	–	Secure Hash Algorithm
SIP	–	Session Initiation Protocol
SRTCP	–	Secure Real-time Transport Control Protocol

SRTP	–	Secure Real-time Transport Protocol
SSL	–	Secure Sockets Layer
STUN	–	Session Traversal Utilities for NAT
TETRA	–	TErrestrial Trunked Radio
TFG	–	Trabajo Fin de Grado
TLS	–	Transport Layer Security
TURN	–	Traversal Using Relays around NAT
UDP	–	User Datagram Protocol
VAD	–	Voice Activity Detection

2. Introducción.

En el ámbito de las comunicaciones, la seguridad y la privacidad de la información desempeñan un papel fundamental en el mantenimiento de la integridad y confidencialidad de los datos. En sistemas de radiocomunicación, donde la transmisión de voz y datos es vital para operaciones críticas, como las redes TETRA (*TErrestrial Trunked Radio*) utilizadas por organizaciones de servicios de emergencia y fuerzas de seguridad, la seguridad adquiere aún mayor relevancia. En este contexto, el presente Trabajo de Fin de Grado (TFG) se enfoca en abordar un desafío significativo: implementar un mecanismo de seguridad en la red TETRA que emplea un protocolo privado que funciona a través de ACELP. Para ello, existe un *Gateway* que cambia este protocolo con el RTP y viceversa.

El protocolo RTP es ampliamente utilizado para el transporte de datos en tiempo real, como voz y video, a través de redes IP. Sin embargo, su diseño original no incluye medidas de seguridad robustas, lo que lo hace vulnerable a ataques y amenazas de seguridad. Para mitigar estos riesgos, se propone utilizar el protocolo SRTP (*Secure Real-time Transport Protocol*), que proporciona mecanismos de cifrado, autenticación e integridad de los datos transmitidos, garantizando una comunicación segura y confiable.

La implementación de SRTP en el contexto de la red TETRA requiere un proceso de negociación seguro para establecer los parámetros de seguridad entre los nodos de la red. Para lograrlo, se propone utilizar el protocolo DTLS (*Datagram Transport Layer Security*), que ofrece un mecanismo para establecer una conexión segura antes de iniciar la comunicación mediante SRTP. Esta combinación de SRTP con DTLS proporciona una capa adicional de seguridad, asegurando la privacidad de las comunicaciones en un entorno crítico como el de la red TETRA.

El presente Trabajo de Fin de Grado tiene como objetivo implementar una negociación segura para el protocolo RTP en la red TETRA, utilizando SRTP con DTLS. Se evaluará la viabilidad del enfoque mediante un sistema de prueba de concepto que aborde diversos escenarios para verificar su efectividad y seguridad. Se espera que los resultados de este trabajo contribuyan significativamente a mejorar la seguridad de las comunicaciones en la red TETRA, protegiendo la confidencialidad y la integridad de la información transmitida. Asimismo, sentarán las bases para futuras investigaciones y desarrollos con el objetivo de implantar este protocolo en la red TETRA en su totalidad, partiendo de este prototipo.

En resumen, este TFG aborda la importante tarea de implementar una negociación segura para el protocolo RTP mediante la integración de los protocolos SRTP y DTLS en la red TETRA. Mediante este enfoque, se busca garantizar la protección de las comunicaciones en un entorno crítico donde la seguridad y la privacidad son primordiales para el buen funcionamiento y la confianza de las operaciones.

2.1. Planteamiento del problema

Se parte del escenario mostrado en la *Figura 3.1.a.*, en el que se encuentran el nodo central que actúa como administrador, distintos sistemas de radio dentro de una red TETRA y de un *Line Dispatcher* que se utiliza para comunicarse con estos dispositivos desde fuera de la red, para ello se necesita un *Gateway* de VOIP que actúa como intermediario, ya que su función es modificar el protocolo RTP/RTCP que se emplea en la comunicación del *Line Dispatcher* a un protocolo privado de la empresa que actúa con ACELP y viceversa. Para establecer una llamada, ambos extremos necesitan registrarse en el administrador para que así puedan ser buscados en la red y se encarga también de proporcionarle la información suficiente del otro extremo y de activar o desactivar el PTT si es modo *semi-dúplex*. Una vez establecida la llamada, si un extremo está dentro de la red TETRA y el otro fuera, como puede ser el *Line Dispatcher*, se comunicarán a través del *Gateway* para que este cambie de un protocolo a otro sin dar dificultad a los extremos.

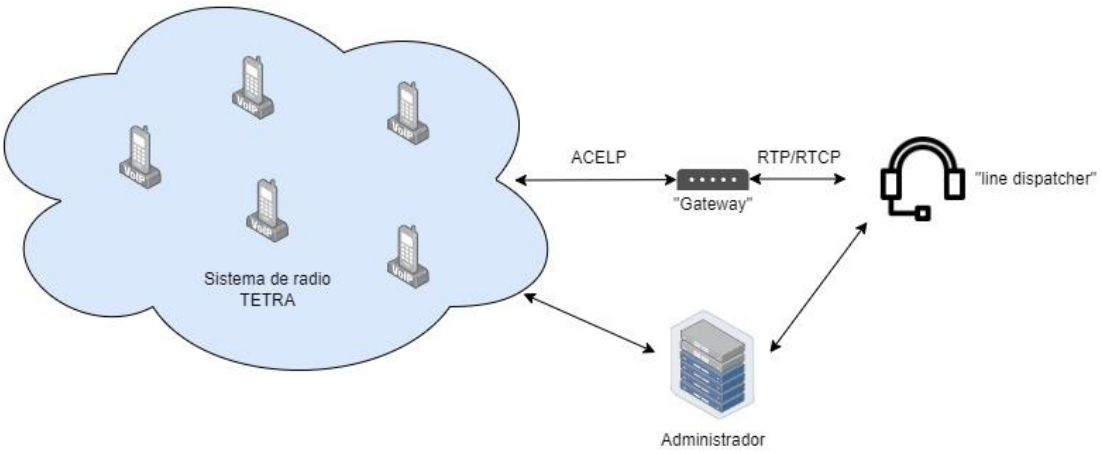


Figura 3.1.a.: Escenario anterior al proyecto.

Este escenario, tal y como se ha explicado, está desarrollado con el protocolo RTP/RTCP, lo cual hace que la conversación no esté cifrada, por lo que intrusos que tengan acceso a cualquier red que trasladen estos paquetes pueden decodificarlo con aplicaciones como *Wireshark* y descubrir la conversación, lo que hace que esta no sea segura tal y como se observa en el capítulo *6. Análisis de las tramas*. Por tanto, se ha planteado cambiar el protocolo a SRTP/SRTCP, lo que nos lleva al siguiente problema, y es que el *Gateway* no está configurado para cambiar de ACELP a SRTP ni viceversa, por lo que primero se quiere verificar si la comunicación con SRTP es viable en la red de la empresa.

En primer lugar, el protocolo SRTP se puede desarrollar a niveles altos como es PJSIP, pero para mantener la compatibilidad en sistemas privados con la empresa, se tiene que desarrollar con PJMedia, para poder así, trabajar al nivel más bajo de interfaz de acceso

a la librería modificando los paquetes a través de la librería presentada en el capítulo *3.2. Librerías utilizadas*.

En segundo lugar, se tiene que comprobar que SRTP sí que funcione al nivel más bajo, yendo paso por paso por los escenarios realizados en el capítulo *5. Escenarios realizados.* en el que se observa que la comunicación es eficaz y segura tal y como se planteaba.

El tercer paso es modificar el *Gateway* para que funcione, aparte de con RTP, con SRTP y así poder utilizar este protocolo en los sistemas de radio de la red TETRA para que la comunicación entre ellos y el *Line Dispatcher* sea segura aunque haya intrusos en la red. Que este paso ya se observará en un futuro, tal y como se explica en el capítulo *8. Conclusión y líneas futuras.*

3. Análisis previo.

3.1. Elementos y lenguajes de programación utilizados.

En este apartado, se muestran los diferentes programas y aplicaciones con sus respectivos lenguajes de programación (si constan de ello):

Microsoft Visual Studio 2012 y Microsoft Visual Studio Code:

Microsoft Visual Studio es un entorno de desarrollo integrado compatible con múltiples lenguajes de programación que permite a los desarrolladores crear sitios y aplicaciones web que se comuniquen entre estaciones de trabajo.

Ambos programas se han utilizado para la edición de código, que ha sido escrito en el lenguaje C++. Este primero, aparte, se ha usado para la ejecución de la aplicación del *Line Dispatcher* que se verá más adelante en varios capítulos. Han sido los dos programas más utilizados a lo largo del proyecto.

Eclipse:

Es un entorno de desarrollo software que soporta varios lenguajes construido alrededor de un *workspace* al que pueden incluirse un gran número de *plugin* que proporcionan funcionalidades concretas relacionadas con lenguajes específicos o con la interacción con otras herramientas implicadas en el desarrollo de una aplicación.

En este proyecto se ha utilizado en el lenguaje de C++ para la visualización y edición del código del *Gateway*. Hemos utilizado este programa y no Microsoft Visual Studio debido a que la aplicación que controla el *Gateway* está en un dispositivo de Linux y no de Windows.

Wireshark:

Es un Software libre que se utiliza como analizador de protocolos utilizado para realizar análisis en redes de comunicaciones añadiendo una interfaz gráfica y múltiples opciones como el filtrado de información.

En este proyecto, su utiliza para poder investigar las distintas tramas empleadas en las comunicaciones planteadas en los distintos capítulos y para representar las señales del audio escuchado durante estas.

3.2. Librerías utilizadas.

PJSIP:

[1] Es una librería SIP (*Session Initiation Protocol*) desarrollada en C y C++, que facilita la creación de aplicaciones de comunicación en tiempo real como llamadas de voz y video, mensajes instantáneos y videoconferencias a través de la red IP. Gracias a su diseño modular y API bien documentada, los desarrolladores pueden seleccionar las características necesarias para sus aplicaciones, lo que la convierte en una opción versátil y fácil de usar. Además, PJSIP es altamente portátil y puede ejecutarse en diversas plataformas, brindando flexibilidad para alcanzar una amplia audiencia.

Esta librería se destaca por su soporte para una amplia variedad de códecs de audio y video, lo que permite la interoperabilidad con diferentes sistemas y dispositivos. Además, PJSIP implementa características avanzadas como la autenticación, la encriptación y el enrutamiento de llamadas, lo que garantiza una comunicación segura y confiable en entornos empresariales y de consumo.

En este proyecto se ha utilizado como apoyo a funciones pertenecientes a la librería de PJMedia que se verá más adelante en este capítulo.

PJNATH:

[2] Esta librería se enfoca en superar los desafíos de las redes NAT (*Network Address Translation*) y los firewalls en las comunicaciones en tiempo real. Los dispositivos detrás de NAT suelen tener direcciones IP privadas y no son directamente accesibles desde Internet. Esto puede causar problemas en el establecimiento de conexiones punto a punto para aplicaciones de VoIP y videoconferencias.

PJNATH proporciona métodos para la resolución de nombres, el descubrimiento de servidores STUN (*Session Traversal Utilities for NAT*) e implementación de ICE (*Interactive Connectivity Establishment*). Gracias a STUN, los dispositivos pueden obtener sus direcciones IP públicas y, con ICE, los dispositivos pueden negociar rutas de comunicación óptimas para atravesar NAT y firewalls. Además, PJNATH es compatible con TURN (*Traversal Using Relays around NAT*), lo que permite el uso de servidores TURN como solución alternativa en situaciones más complejas donde la comunicación punto a punto no es posible.

En este proyecto se ha utilizado como apoyo a funciones pertenecientes a la librería de PJMedia que se explica a continuación.

PJMEDIA:

Es una librería que se utiliza para el procesamiento de medios en aplicaciones de VoIP. Ofrece funcionalidades para manejar códecs, procesar flujos de medios, capturar y

reproducir audio y video, y otras características avanzadas. Con PJMEDIA, los desarrolladores pueden implementar aplicaciones de VoIP con alta calidad de audio y video, así como funciones avanzadas como cancelación de eco y detección de actividad de voz. Esta librería optimiza la experiencia del usuario y asegura una comunicación fluida.

PJMEDIA es altamente personalizable, lo que permite a los desarrolladores adaptar el procesamiento de medios a las necesidades específicas de sus aplicaciones. Además, es compatible con una amplia variedad de códecs y formatos de medios, lo que facilita la interoperabilidad con otras plataformas y sistemas.

En este proyecto se ha utilizado para usar funciones al nivel más bajo de tramas para poder implementar los protocolos RTP y SRTP para el envío y recepción de tramas junto con el proceso de inicialización de las sesiones utilizadas para ello que se pueden observar durante el capítulo *4. Descripción de RTP y SRTP.*

La elección de esta librería como principal de este proyecto es que en el GVOIP ya se utiliza esta, pero a niveles más altos, por lo que simplemente se tendría que cambiar por otros niveles más bajos de interfaz de acceso a la librería.

OpenSSL:

[4] Para garantizar la seguridad en las comunicaciones, OpenSSL es una librería criptográfica fundamental utilizada ampliamente en aplicaciones de red. Ofrece cifrado y descifrado, autenticación y firma digital, y también implementa protocolos de seguridad como SSL/TLS. Con soporte para diversos algoritmos criptográficos y certificados X.509, permite la transmisión segura de datos y la autenticación de extremo a extremo.

Integrando OpenSSL con las librerías PJSIP, PJNATH y PJMEDIA, los desarrolladores pueden proporcionar una solución de VoIP altamente segura y confiable. La combinación de estas librerías ofrece una plataforma completa para el desarrollo de aplicaciones de comunicación en tiempo real, garantizando una experiencia de usuario de alta calidad y protegiendo la privacidad y seguridad de las comunicaciones.

En este proyecto se ha utilizado para ayudar a la negociación de las claves SRTP utilizando el protocolo DTLS, que se explica en el capítulo *4.4. Uso del DTLS*.

Librería interna privada:

Debido a temas de privacidad, no se pueden dar detalles específicos de esta librería. Se ha utilizado para gestionar el *Gateway* y el nodo administrador de la red TETRA para el establecimiento de llamada y dar dirección a cada dispositivo de esta red.

4. Descripción de RTP y SRTP.

4.1. Funcionamiento, Definición y tramas del Protocolo RTP.

El *Real-time Transport Protocol* (RTP) y el *Real-time Control Protocol* (RTCP) son dos protocolos complementarios utilizados en la transmisión de datos multimedia en tiempo real a través de redes IP. Ambos fueron desarrollados por *Internet Engineering Task Force* (IETF) y se rigen por diferentes *RFCs*. La primera versión de RTP fue publicada en 1996 en el documento RFC 1889 y la final en *RFC* 3550 en 2003.

El RTP, divide en tramas los datos multimedia y proporciona servicios de transporte, control de flujo y sincronización. Por otro lado, el RTCP desempeña un papel crítico en el monitoreo y control de la calidad de la transmisión. Aunque utiliza el mismo canal de transporte que el RTP, el RTCP se encarga de enviar periódicamente paquetes de control hacia los participantes de una sesión en tiempo real. Estos paquetes contienen información valiosa sobre la calidad de la conexión, estadísticas de los medios transmitidos y detalles sobre la sincronización entre los participantes. Además, RTCP, facilita la identificación de posibles problemas en la red, como la pérdida de paquetes o la variación en los retardos, lo que permite tomar acciones correctivas para mejorar la calidad de la experiencia de usuario en tiempo real.

La estructura de las tramas RTP consta de una cabecera y una carga útil. La cabecera se divide en varios campos que proporcionan información importante sobre la trama y su contenido. A continuación, se describen los campos más relevantes de la cabecera de una trama RTP:

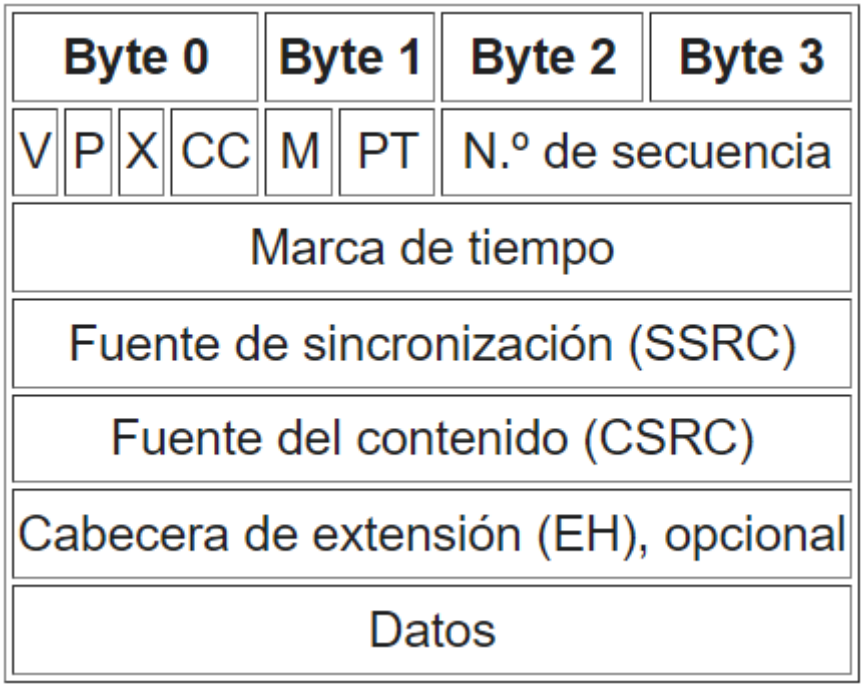


Figura 4.1.a.: Esquema de la estructura de una trama RTP.

- Número de versión (*Version*): Este campo de 2 bits indica la versión del protocolo RTP utilizado.
- Tipo de servicio (*Padding, Extension, CSRC Count*): Estos campos de 1 bit cada uno indican la presencia de información adicional en la trama, como bits de relleno (*Padding*), una sección de extensión (*Extension*) y el número de identificadores de fuentes de contribución (*CSRC Count*).
- Marcador (*Marker*): Este campo de 1 bit se utiliza para marcar tramas importantes dentro de una secuencia de datos.
- Tipo de carga útil (*Payload Type*): Este campo de 7 bits especifica el tipo de datos multimedia incluidos en la trama RTP, como audio, video o metadatos.
- Número de secuencia (*Sequence Number*): Este campo de 16 bits se utiliza para ordenar y detectar pérdidas de tramas durante la transmisión.
- Marca de tiempo (*Timestamp*): Este campo de 32 bits se utiliza para sincronizar los flujos de audio y video en el receptor.
- Número de identificadores de fuentes de contribución (*SSRC Count*): Este campo de 4 bits indica la cantidad de identificadores de fuentes de contribución presentes en la trama.
- Identificadores de fuentes de contribución (*CSRC List*): Estos campos de 32 bits cada uno identifican las fuentes de contribución asociadas con la trama.
- Cabecera de extensión (*EH*): Es un campo opcional que proporciona información de capa de red. Se utilizan para la fragmentación, la seguridad y la movilidad principalmente, aunque tiene varios usos más. Tiene un tamaño de 32 bits.
- La carga útil de la trama RTP contiene los datos multimedia en sí. El formato y la estructura de la carga útil dependen del tipo de datos que se transmiten. Por

ejemplo, si se trata de audio, la carga útil puede contener muestras de audio codificadas.

4.2. Funcionamiento, Definición y tramas del Protocolo SRTP.

El *Secure Real-time Transport Protocol* (SRTP) y el *Secure Real-time Transport Control Protocol* (SRTCP) son extensiones esenciales utilizadas para garantizar la seguridad en la transmisión de datos multimedia en tiempo real a través de redes IP.

SRTP se rige por el RFC 3711. Incorpora funciones críticas de seguridad, como el cifrado y la autenticación, para asegurar la confidencialidad, integridad y autenticidad de las tramas RTP transmitidas.

En el caso del SRTCP, su funcionamiento se encuentra definido en el RFC 5764. SRTCP se encarga de brindar seguridad a los paquetes de control enviados mediante RTCP. Tanto SRTP como SRTCP incorporan técnicas de cifrado, autenticación e integridad para proteger los datos de control contra escuchas no autorizadas y modificaciones malintencionadas. Los parámetros de seguridad son acordados previamente entre el emisor y el receptor a través de protocolos de negociación de claves, como *Internet Key Exchange* (IKE), según lo especificado en el RFC 7296, *Internet Protocol Security* (IPSec) referenciado en RFC 6434 y el *Datagram Transport Layer Security* (DTLS) cuya referencia es RFC 9147. Este último es el protocolo que se utilizará en este proyecto, sin embargo, también existe la posibilidad de tener las claves ya establecidas en ambos extremos, por lo que no se necesitaría la negociación que se realiza en el DTLS.

La utilización de SRTP y SRTCP en aplicaciones como telefonía IP, videoconferencias y transmisiones en vivo por Internet ha demostrado ser crucial en entornos donde la seguridad de la información es prioritaria protegiendo la privacidad de las comunicaciones y la calidad de esta en tiempo real.

4.3. Comparación RTP con SRTP.

Existen varias diferencias significativas entre SRTP y RTP. A continuación, se presentan algunas de las principales comparaciones:

- Seguridad: La principal diferencia entre SRTP y RTP radica en la seguridad. Mientras que RTP no proporciona mecanismos de seguridad por sí mismo, SRTP agrega cifrado y autenticación a las tramas RTP para proteger los datos multimedia.
- Integridad: SRTP asegura la integridad de los datos multimedia mediante la adición de una etiqueta de autenticación de mensaje (MAC) a cada trama, lo que

permite detectar modificaciones malintencionadas. RTP no tiene mecanismos incorporados para verificar la integridad.

- Autenticidad: SRTP proporciona mecanismos para garantizar la autenticidad de las tramas, asegurando que provienen de la fuente esperada y no han sido manipuladas. RTP no tiene estas capacidades.
- Sobrecarga: Debido a los procesos de cifrado y autenticación adicionales, SRTP tiene una sobrecarga computacional y de ancho de banda mayor en comparación con RTP sin seguridad. Esto puede afectar el rendimiento en entornos con recursos limitados.

En este caso, contemplando la codificación con G711, la cabecera tanto de RTP, como SRTP es de 12 bytes y de carga útil de 64 bytes, haciendo un total de 76 bytes, sin embargo, en SRTP se le añaden 10 bytes de autenticación y cifrado del algoritmo *AES_CM* que se podrá observar con más detalle en el capítulo *4.4. Uso del DTLS*. Todo esto, provoca una sobrecarga del 15,79%, sin embargo, a la hora del envío de tramas, se envían 3 paquetes en una trama, por lo que los bytes por trama son 192 a lo que se le añaden a SRTP los 10 bytes mencionados anteriormente dando como resultado 202 bytes, lo que hace una sobrecarga del 4’95%.

Por lo que teniendo únicamente como desventaja para el uso de SRTP esta sobrecarga que es pequeña, compensa su uso frente al RTP sin seguridad.

4.4. Uso del DTLS.

DTLS [5], acrónimo de *Datagram Transport Layer Security*, es un protocolo de seguridad que proporciona una capa de cifrado y autenticación para la transmisión de datos en tiempo real a través de redes de datagramas, como UDP. DTLS es una versión adaptada de TLS (*Transport Layer Security*), diseñado para adaptarse a las características y exigencias de las aplicaciones que requieren una conexión segura y fiable en entornos de red propensos a la pérdida de paquetes, como lo son las comunicaciones en tiempo real.

DTLS se basa en una combinación de algoritmos criptográficos para proporcionar cifrado, autenticación e integridad de los datos transmitidos, asegurando que la información sensible no pueda ser interceptada o modificada por atacantes.

El uso principal de DTLS es la negociación para el intercambio de claves que enviará a SRTP para asegurar su protección. Cuando dos extremos (por ejemplo, un cliente y un servidor) desean establecer una comunicación segura mediante SRTP, primero utilizan DTLS para establecer una conexión segura. Durante este proceso, DTLS negocia los algoritmos criptográficos y las claves de cifrado que se utilizarán para proteger la información en tiempo real. Una vez que la conexión DTLS se ha establecido con éxito, SRTP utiliza las claves acordadas para cifrar y autenticar las tramas de datos multimedia que se transmiten entre los extremos. Esta negociación se realiza a través de un

handshake, que consiste en las siguientes etapas que se ven representadas en la *Figura 4.4.a.*.

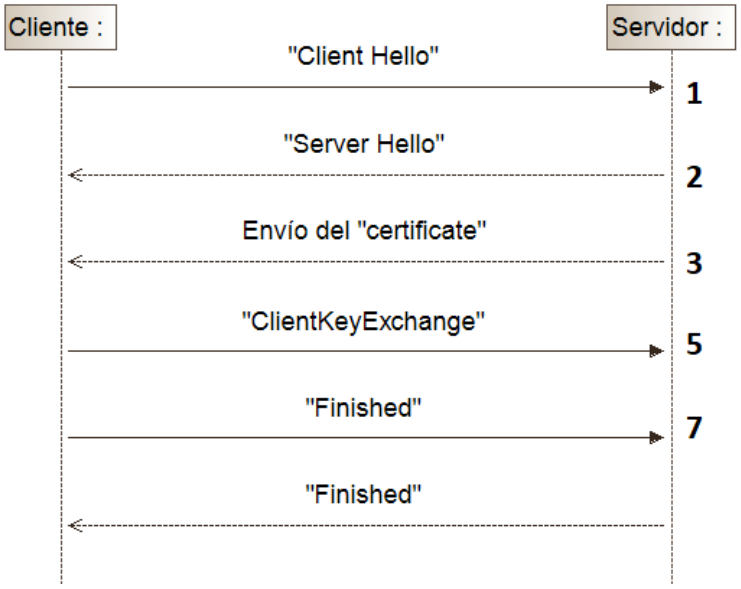


Figura 4.4.a: Esquema de la negociación para obtener las claves de SRTP a través de DTLS.

1. Inicio: El cliente inicia el proceso enviando un mensaje *ClientHello* al servidor. En este mensaje, el cliente incluye su lista de *Cipher Suites* (combinaciones de algoritmos criptográficos) y otras capacidades de seguridad compatibles. También genera un valor de *nonce* (número usado solo una vez) y otros parámetros necesarios para el proceso.
2. Respuesta del servidor: El servidor responde al mensaje del cliente con un mensaje *ServerHello*. En este mensaje, el servidor selecciona una de las *Cipher Suites* propuestas por el cliente que sea compatible y segura para ambos extremos. El servidor también genera su propio valor de *nonce* y otros parámetros necesarios para el proceso.
3. Intercambio de certificados: El servidor envía su certificado digital al cliente en el mensaje *Certificate*. El certificado contiene la clave pública del servidor, que el cliente usará más adelante para cifrar mensajes durante la comunicación.
4. Autenticación y verificación del certificado: El cliente verifica la autenticidad del certificado del servidor utilizando su lista de autoridades de certificación de confianza. Si el certificado es válido y confiable, el cliente continúa con el proceso.
5. Intercambio de claves: El cliente genera un valor de *premaster secret* (secreto previo a la clave) y lo cifra utilizando la clave pública del servidor. El cliente envía el secreto cifrado al servidor en el mensaje *ClientKeyExchange*. El servidor descifra el secreto utilizando su clave privada y obtiene el mismo valor.

6. Generación de claves de sesión: Tanto el cliente como el servidor utilizan el *premaster secret* y los valores de *nonce* intercambiados para generar claves de sesión para la comunicación segura. Estas claves de sesión son utilizadas por los algoritmos criptográficos seleccionados para cifrar y autenticar los datos durante la transmisión.
7. Intercambio de mensajes de finalización: Tanto el cliente como el servidor envían un mensaje *Finished* al otro extremo para confirmar que el *handshake* se ha realizado correctamente y que ambas partes están listas para comenzar la transmisión segura.

4.5. Código de RTP.

Los pasos que se han seguido para poder utilizar el protocolo RTP con la librería PJMedia referenciada en el capítulo *3.2. Librerías utilizadas.* ha sido:

1. Inicialización de la librería a través de la función *pj_init()*.
2. Creación e inicialización de la memoria caché y de la dirección a los codificadores:
    a. *Caching pool*: Grupo de almacenamiento en la memoria caché. Para ello se debe emplear la función *pj_caching_pool_init()*, añadiéndole como parámetros de entrada el propio *Caching pool*, la política por defecto y la capacidad máxima que queremos que almacene nuestra caché. Para poder crearse este espacio, se necesita la función *pj_pool_create()* cuyo parámetro de salida se guarda en una variable y de entrada se mandan los valores del *caching pool*, el nombre y el tamaño inicial y adicional del bloque de la memoria.
    b. Creación de un administrador de eventos a través de la función *pjmedia_event_mgr_create()*, su parámetro de entrada es la variable obtenida con la función anterior.
    c. *Endpoint:* Sirve para administrar posteriormente los codificadores del sistema. Se usa la función *pjmedia_endpt_create()* y como parámetros de entrada: los valores de *caching pool*, el número de hilos necesarios (1 en este caso) y el *endpoint* vacío para poder crearse.
3. Registro del códec en un *endpoint*: Se deben efectuar estas dos funciones: *pjmedia_codec_register_audio_codecs()* y *pjmedia_endpt_get_codec_mgr()*, pasándole como parámetro de entrada a ambas funciones su *endpoint*. Se utilizará *pjmedia_codec_mgr_find_codecs_by_id()* para buscar los códecs que tengan el nombre enviado.
4. Creación del socket RTP y RTCP y vinculación a su puerto de entrada: *pjmedia_transport_udp_create()*, para ello, se necesita el *endpoint* utilizado para registrar los sockets, un nombre (opcional), el número del puerto necesario para RTP (para RTCP será uno mayor) y un transporte *pjmedia_transport* recién iniciado.
5. Se establece el códec: Para ello se deben seguir los siguientes pasos:

a. Se obtiene el administrador del códec a partir del *endpoint* con la función *pjmedia_endpt_get_codec_mgr()*.
b. Se reciben los parámetros por defecto a partir de la función *pjmedia_codec_mgr_get_default_param()*, pero se deshabilita el VAD porque está por defecto.
c. Se solicita al administrador una instancia del códec con una información que se pide por entrada con la función *pjmedia_codec_mgr_alloc_codec()*.
d. Inicializamos y abrimos el códec a partir de *pjmedia_codec_init()* y de *pjmedia_codec_open()*.

6. Establecemos las sesiones RTP:
    a. En primer lugar, se inicializa el socket RTP y el RTCP a partir de la función *pj_sockaddr_in_init()*.
    b. Seguidamente, se activan las sesiones de entrada y salida con el puerto e información correspondiente con *pjmedia_rtp_session_init()*.
    c. Por último se configura la sesión RTCP, para ello, se necesita el puerto RTCP, el reloj del códec y la función *pjmedia_rtcp_init()*.
7. Se activan los *call-backs on_rx_rtp()* y *on_rx_rtcp()*, que se activan al recibir los paquetes RTP y RTCP correspondientemente, para ello, se necesita el transporte que se ha modificado durante los anteriores apartados, la información, los puertos RTP y RTCP y la longitud del socket para enviarlos a la función *pjmedia_transport_attach()*.
8. Se inicializa la sesión de transporte con la configuración en el SDP local y remoto, cuando se trabaja con SRTP, esto activará el cifrado y descifrado de los paquetes. La función que se utiliza para ello es *pjmedia_transport_media_start()*.
9. Para el lanzamiento de paquetes se necesita invocar el hilo *send_RTP_thread*.
10. El *call-back on_rx_rtcp()* se utiliza para recoger los paquetes RTCP que llegan en la comunicación, es decir, los paquetes de control. Una vez recogida cada trama, se actualiza la sesión *pjmedia_rtcp_rx_rtcp()*.
11. El siguiente *call-back* es *on_rx_rtp()*:
    a. Se recogen los paquetes RTP recibidos, estos contienen la información de la señal de audio codificada, en este caso en G711. Lo primero que se hace al recibir la trama es decodificar con *pjmedia_rtp_decode_rtp()*.
    b. Se actualizan las sesiones RTP y RTCP a través de *pjmedia_rtcp_rx_rtp()* y de *pjmedia_rtp_session_update()*.
    c. El códec inspecciona el paquete y lo divide en tramas individuales, para ello, como parámetros de entrada a la función *pjmedia_codec_parse()*, necesitamos el códec, el paquete recibido y su longitud, la marca de tiempo de la primera muestra del paquete (*timestamp*), el puntero en el que se indica el número de *frames* en el vector y la variable que devuelve las tramas que se han detectado en el paquete.
    d. Para decodificar cada *frame* obtenido en la función anterior, se utiliza la siguiente: *pjmedia_codec_decode()* y así se obtiene cada trama que se necesita para obtener la señal de audio.

12. Para el envío de los paquetes, como se ha dicho anteriormente, se utiliza el hilo *send_RTP_thread* se han utilizado para desarrollarlo los siguientes pasos:
    a. Lo primero que hay que hacer, es crear la cabecera que se necesita para el protocolo RTP y se codifica con *pjmedia_rtp_encode_rtp()*, para ello se necesita la sesión de salida, la información del códec y la cabecera.
    b. Se obtienen los bytes que se quieren enviar y se acoplan a la cabecera creada anteriormente.
    c. Por último, se envía directamente el paquete completo a través de la función *pjmedia_transport_send_rtp()*, la que necesita como parámetro de entrada únicamente el transporte (con toda su información almacenada anteriormente) y el paquete. Esta función entregará dicho paquete directamente a la dirección destino especificadas en el *pjmedia_transport_attach()*.

4.6. Código de SRTP.

Para implementar la seguridad con protocolo SRTP, se han tenido que añadir funciones, tanto de PJMedia como de openssl:

1. En primer lugar hay que cambiar el tipo de transporte a *PJMEDIA_TRANSPORT_TYPE_SRTP*.
2. Después de crear las sesiones RTP (apartado 6 del capítulo *4.5. Código de RTP*), agregamos las opciones por defecto de SRTP con la función *pjmedia_srtp_setting_default()*. Y creamos esta sesión gracias a *pjmedia_transport_srtp_create()* añadiéndole el transporte, el *endpoint* del códec y las opciones que se acaban de crear.
3. Una vez empezada la sesión de transporte (apartado 8 del capítulo *4.5. Código de RTP*), existen dos opciones:
    a. La primera es tener las claves anteriormente de cada extremo de la comunicación y crear el transporte SRTP enviando dichas claves a la función *pjmedia_transport_srtp_start()*, por lo que se tendrían que saber antes de empezar la comunicación si solo utilizamos este protocolo.
    b. La segunda consiste en hacer una negociación, se utilizará el protocolo DTLS, para ello hay que seguir los siguientes pasos:
        i. Crear el hilo *on_srtp_nego_complete* para saber el momento en el que acaba este proceso.
        ii. A continuación, inicializar la información que tienen los parámetros de DTLS con *pj_bzero()* y copiamos los sockets con la dirección y los puertos RTP y RTCP con *pj_sockaddr_cp()*.
        iii. Asignamos un cliente y un servidor activando el *flag is_role_active* a este primero. El servidor está en espera hasta que recibe el saludo del cliente que inicia la conversación y envía el *handshake* cada cierto tiempo hasta que recibe respuesta activando así el

hilo *on_srtp_nego_complete*. El inicio de la negociación se efectúa con *pjmedia_transport_srtp_dtls_start_nego()*. A esta función se le añade un *fingerprint* que se usa como certificado remoto, en este proyecto no se hace esta verificación, por lo que no hay una autentificación fiable. Se ha optado a esta opción, debido a que los mecanismos para pasar dicho *fingerprint* entre los dos extremos aún no están perfilados, se ha preferido centrarse en el resto de los aspectos de SRTP.

iv. Una vez se completa la negociación, se activa directamente la comunicación mediante el protocolo SRTP sin tener que llamar ninguna otra función, a parte del hilo de envío.

5. Escenarios realizados.

El escenario general del proyecto está reflejado en la *Figura 5.a.*, en los siguientes apartados se irá viendo el escenario de cada uno de ellos para ser explicados de una manera más clara.

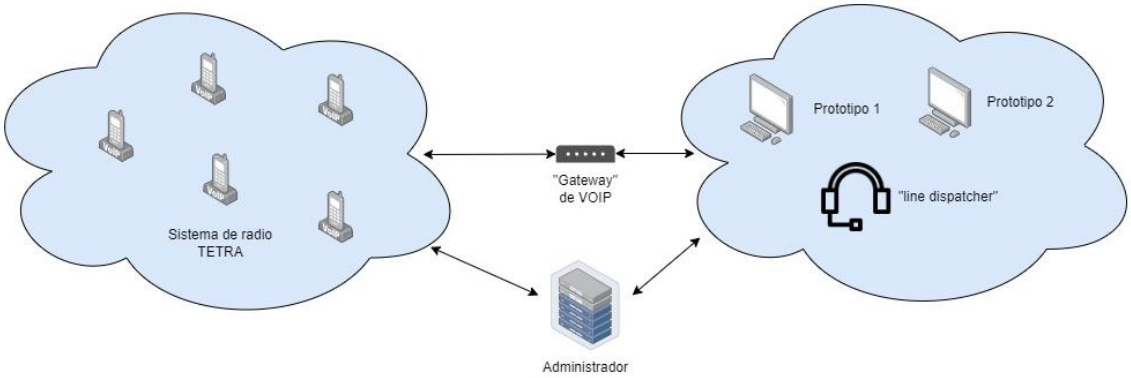


Figura 5.a.: Escenario general del proyecto.

Este escenario general está compuesto por 2 redes, la primera es un sistema de radio TETRA en la que están sus dispositivos, y en la segunda están los dos prototipos que se utilizarán a lo largo de los siguientes apartados y el *Line Dispatcher*. Para establecer una llamada entre dos extremos, tanto cualquier dispositivo de radio como el *Line Dispatcher*, se utiliza el nodo central que se utiliza como administrador y es el encargado, a través de un protocolo privado interno, de proporcionar la dirección, los puertos, el PTT, etc.. Una vez se establece la llamada, si uno de ellos está dentro de la red TETRA y el otro fuera, se comunican traspasando el *Gateway*, cuya función es convertir el tráfico ACELP empleado en la red TETRA a RTP/RTCP utilizado en el exterior y viceversa, por lo que actúa como intermediario entre el dispositivo de radio y el *Line Dispatcher* para que la conversación pueda realizarse.

5.1. Implementación de la librería PJMedia en un prototipo.

Este primer escenario es propuesto para verificar la compatibilidad del protocolo RTP dado a través de PJMedia con las librerías internas de la empresa, para poder así asegurarse de su correcto funcionamiento.

Para ello, se ha realizado un prototipo cuya funcionalidad es enviar y recibir un mismo archivo de audio guardado en el ordenador a través de RTP utilizando un puerto para el envío y otro para la recepción.

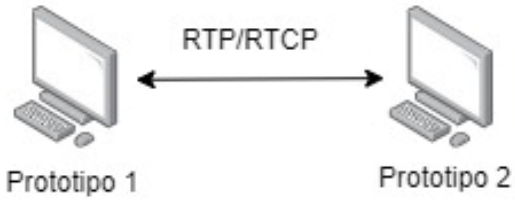


Figura 5.1.a.: Esquema del escenario 1. Envío de archivo de audio entre dos consolas.

Para lograr este objetivo, se requiere la operación simultánea de dos consolas con sistema operativo Windows, ambas envían el fichero por paquetes codificados con RTP, reciben las tramas del otro con un puerto distinto y crean otro archivo de audio para poder reproducirse una vez se haya terminado la conexión.

Ahora, con el conocimiento adquirido en este escenario con el uso de la librería de PJMedia, se dará paso al siguiente, donde se llevará a cabo una demostración práctica en tiempo real para ilustrar mejor el funcionamiento de este sistema.

5.2. Migración del Line Dispatcher a PJMedia y conexión con la radio.

Este segundo escenario es propuesto para verificar que la librería de PJMedia con RTP sin seguridad funcione con un sistema real de radio para posteriormente añadirsela.

Partimos de la aplicación del *Line Dispatcher* ya proporcionada. Los cambios que se han realizado con respecto al original, ha sido la migración a PJMedia, ya que antes, la creación de sesiones y los envíos se realizaban a través de una librería interna que no soporta SRTP, por lo que se necesita esta migración a PJMedia planteada. Los pasos que se han desarrollado para ello han sido explicados en el apartado *4.5. Código RTP* del *Capítulo 4*., tal y como se ha hecho en el apartado *5.1. Implementación de la librería PJMedia en un prototipo.*

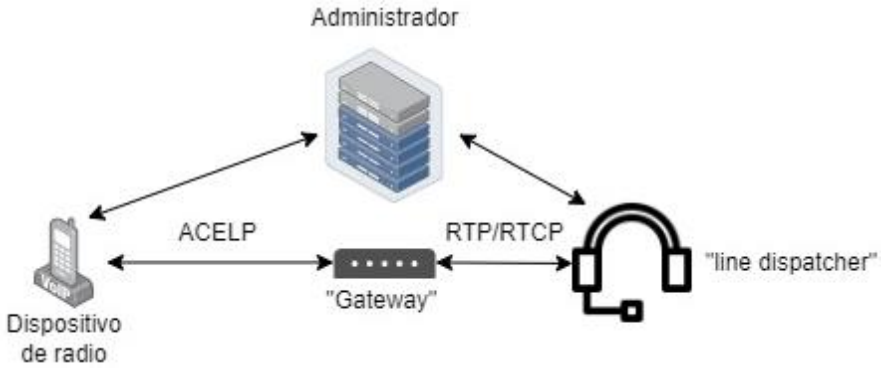


Figura 5.2.a.: Esquema escenario 2. La aplicación de Line Dispatcher se comunica con el sistema de radio atravesando el Gateway.

La radio se encuentra en una red de infraestructura TETRA, por lo que la aplicación del *Line Dispatcher* se debe conectar a esta a través de un *Gateway* que únicamente soporta RTP, por lo que en este escenario seguiría funcionando, ya que se utiliza este protocolo.

En esta conexión se pueden realizar llamadas individuales (*dúplex* y *semi-dúplex*) y de grupo a tiempo real. La señal se envía a través de UDP de los paquetes RTP codificada por G711.

Ambos extremos pueden iniciar la llamada de ambos tipos mencionados anteriormente y el otro extremo es el encargado de aceptarla o rechazarla. Al aceptarla, automáticamente se puede comunicar hablando directamente (modo *dúplex*) o activando y desactivando el PTT de ambos extremos conforme el turno de habla (modo *semi-dúplex*), quien tiene el permiso inicial depende de si la llamada es directa o *hook* (con descuelgue).

La función de este escenario es comprender cómo funciona esta aplicación de *Line Dispatcher* y tener un contacto con lo que sería el producto final que es la comunicación por sistema de radio.

5.3. Implementación del protocolo SRTP en un prototipo.

Este tercer escenario, comprueba el cifrado de SRTP con PJMedia en modo de trabajo compatible con sistemas actuales.

Para ello, se implementan los cambios propuestos en el capítulo *4.6. Código SRTP* en el primer escenario mostrado en *5.1. Implementación de la librería PJMedia en un prototipo*. Con ayuda de los protocolos SRTP y DTLS, se intentará que la señal se cifre con claves después de negociarlas para así no estar expuestas las conversaciones para cualquier intruso que se infiltre en la red o que pertenezca a ella.

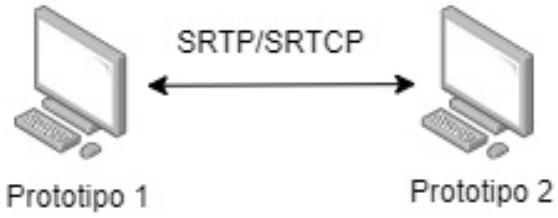


Figura 5.3.a.: Esquema del escenario 3. Envío de archivo de audio entre dos consolas.

Tras comprobar que este cifrado ha funcionado y no puede ser descifrado por *Wireshark*, tal y como se observará en el capítulo *6.4. Demostración de la seguridad de la comunicación de audio.*

En el próximo escenario, se podrá observar el escenario final de este proyecto, donde se apreciarán los resultados y beneficios de la aplicación de estos cambios en términos de seguridad y confidencialidad en la comunicación por sistema de radio. La incorporación de cifrado mejora significativamente la integridad de las transmisiones,

protegiendo la información sensible y asegurando una comunicación más segura y confiable en este entorno.

5.4. Implementación de Comunicación Segura en Red TETRA mediante Simulación SRTP/DTLS.

Este escenario es el más cercano al objetivo final, debido a que el *Gateway* no soporta los protocolos SRTP y DTLS que necesitan convertirse en ACELP para que funcionen en la red interna de TETRA a la que pertenecen los dispositivos de radio. Para ello, se establece la llamada accediendo al administrador por ambos extremos, pero se engaña al sistema para que el *Gateway* utilice un puerto falso. Al final del establecimiento, el tráfico de audio real del dispositivo de radio se sustituye una vez establecida la llamada por el del prototipo, de forma que no es necesario modificar el *Gateway*.

Este escenario se puede observar en la *Figura 5.4.a.* en la que "Prototipo 1" y "line Dispatcher" pertenecen a una red privada, mientras que el "Dispositivo de radio" está en la red TETRA, cuyo tráfico es sustituido por el del prototipo una vez es iniciada la llamada tras establecerse con el "Administrador".

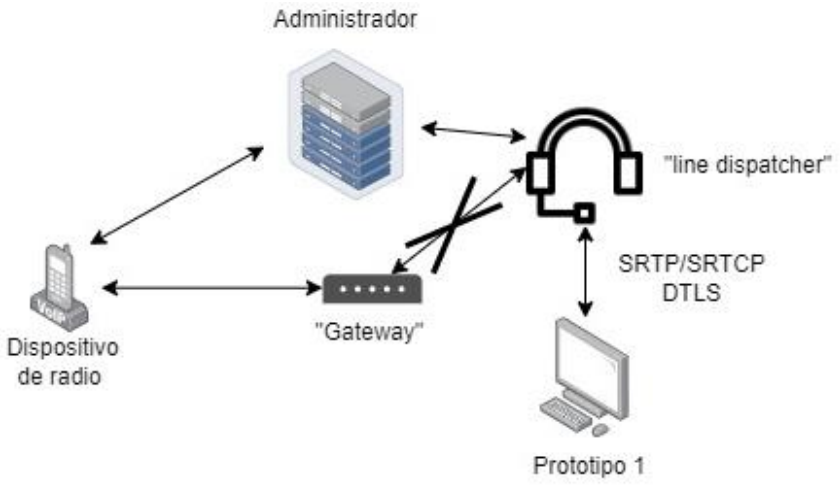


Figura 5.4.a.: Escenario final. El prototipo simula el sistema de radio para enviar el audio SRTP/SRTCP/DTLS.

Para ello, se siguen los pasos de este esquema:

1. En primer lugar, se ejecuta en la ventana de comandos el código del prototipo, que actuará como servidor para la negociación de DTLS para que esté en espera hasta que el *Line Dispatcher* empiece la negociación para obtener las claves de SRTP y comenzar la conversación. Se establece esta disposición de cliente y servidor, debido a que en la negociación DTLS, el cliente es el que empieza la negociación mientras que el servidor espera el mensaje de *ClientHello*.

2. A continuación se llama a través de alguno de los dos extremos, ya sea desde la aplicación como desde la radio, en cualquier modo y desde el otro, se acepta la llamada.
3. Posteriormente, empieza la negociación de DTLS del *Line Dispatcher* con el prototipo.
4. Una vez completada la negociación, si es modo *dúplex* se puede oír a través de la aplicación de *Line Dispatcher* el archivo de audio en bucle, en cambio, si es en modo *semi-dúplex* se necesita desactivar el PTT de la aplicación y activarlo en el dispositivo de radio para que este se escuche.

Este escenario se ha diseñado para que se pueda realizar también por RTP, es decir, sin seguridad. Para ello se tiene que activar esta función desde el código de la aplicación del *Line Dispatcher* y con los comandos desde el prototipo para poder comunicarse ambos con RTP. En este caso, la negociación no se realiza, por lo que el prototipo directamente envía en bucle el dispositivo de audio una vez se inicie con la ventana de comandos, independientemente de si se ha realizado o no el establecimiento de llamada.

Una vez se cierra el prototipo, se puede escuchar también a través de un archivo de audio la parte de la comunicación del *Line Dispatcher* hacia el prototipo, por lo que la conversación funciona en ambos sentidos, es decir, SRTP funciona tanto como para transmisión como para recepción.

Gracias a este escenario, se puede confirmar que esta comunicación es viable y se puede utilizar en el sistema de radio, pero para ello se necesitaría modificar el *Gateway*, ya que actualmente no soporta el SRTP ni el DTLS. Por lo que en unos meses, cuando se modifique el *Gateway* pueda admitir ambos protocolos.

6. Análisis de las tramas.

En este capítulo se comparan las tramas capturadas en cada escenario gracias a *Wireshark*. Como ya hemos visto en el *Capítulo 5. Escenarios realizados*, en los dos primeros escenarios se obtienen tramas TCP con el protocolo RTP y en los dos siguientes, se muestran tanto SRTP para la comunicación como DTLS para la negociación de las claves. Posteriormente se compararán las señales de audio capturadas para así verificar si se han cifrado correctamente y si son vulnerables a ataques.

En un inicio, todos estos paquetes se capturan como UDP tal y como se observa en la *Figura 6.a.*, por lo que se decodifican a través de una función de la propia aplicación de *Wireshark*, *Decode As…*, para poder poner el tipo RTP (tanto para las tramas RTP como para las tramas SRTP).

33	0.540635	172.16.32.243	172.16.32.243	UDP	204 7005 → 4005 Len=172
34	0.540636	172.16.32.243	172.16.32.243	UDP	204 4005 → 7005 Len=172
35	0.572623	172.16.32.243	172.16.32.243	UDP	204 4005 → 7005 Len=172
36	0.572624	172.16.32.243	172.16.32.243	UDP	204 7005 → 4005 Len=172
37	0.604637	172.16.32.243	172.16.32.243	UDP	204 4005 → 7005 Len=172
38	0.604639	172.16.32.243	172.16.32.243	UDP	204 7005 → 4005 Len=172
39	0.636575	172.16.32.243	172.16.32.243	UDP	204 7005 → 4005 Len=172

Figura 6.a.: Captura paquetes UDP sin decodificar a RTP.

Se puede observar únicamente la dirección fuente y destino con sus respectivos puertos y la longitud de los datos y de la cabecera, por lo que así no se obtendría suficiente información, por lo que habría que decodificarlo como se ve en el siguiente apartado.

6.1. Tramas RTP.

Una vez decodificados los paquetes de la *Figura 6.a.*, se observan los de la *Figura 6.1.a.* Se puede observar más información como “P=ITU-T G. 711 PCMA”, esto significa que es un protocolo de la Unión Internacional de Telecomunicaciones-Telecomunicación (ITU-T) que utiliza como códec tanto G711 como PCMA (*Pulse Code Modulation A-law*), en este caso, para la descomprensión se utiliza G711. A continuación aparece el campo de “Fuente de sincronización”, que como se puede observar en el apartado *4.1. Funcionamiento, Definición y tramas del Protocolo RTP.* este campo indica la posición de la fuente de sincronización en la memoria, esto va seguido del número de secuencia del paquete para tenerlo así localizado, y por último, aparece el *Timestamp* en el que se deduce que entre cada paquete hay un valor de 160 de diferencia entre cada una que sirve para sincronizar los flujos de audio y video en el receptor.

```
33 0.540635  172.16.32.243  172.16.32.243   RTP     204 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26518, Time=2880
34 0.540636  172.16.32.243  172.16.32.243   RTP     204 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26503, Time=480
35 0.572623  172.16.32.243  172.16.32.243   RTP     204 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26504, Time=640
36 0.572624  172.16.32.243  172.16.32.243   RTP     204 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26519, Time=3040
37 0.604637  172.16.32.243  172.16.32.243   RTP     204 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26505, Time=800
38 0.604639  172.16.32.243  172.16.32.243   RTP     204 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26520, Time=3200
39 0.636575  172.16.32.243  172.16.32.243   RTP     204 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26521, Time=3360
```

Figura 6.1.a.: Captura de tramas RTP decodificados.

Tras obtener estas tramas, se accede en la misma aplicación de *Wireshark* al menú de *Telephony -> RTP -> RTP Streams*, se selecciona cualquiera de los dos flujos de la conversación o escuchar ambos a la vez que también es posible, y se accede a observar la señal en *Analyze -> Play Streams*. Llegado a este punto, aparece la señal de la *Figura 5.1.a.*, que no está cifrada y puede ser escuchada por cualquier persona que tenga acceso a cualquiera de las dos redes de este escenario o de las redes que separan a ambos extremos en cualquier otro proyecto simplemente accediendo a esté menú y cambiando la opción de *Playback Timing* a *RTP Timestamp* y reproduciendo la señal de audio resultante.

6.2. Tramas SRTP.

Al igual que en el apartado anterior, se decodifica la *Figura6.a.* a RTP, y en un principio, como se puede observar en la *Figura 6.2.a.* en comparación con la *Figura 6.1.a.* no existe ninguna diferencia, el *Timestamp*, la fuente de sincronización y la codificación tienen los mismos valores, por lo que no se va a entrar en detalles de esta captura, sin embargo, la longitud de la trama es de 10 bytes mayor, tal y como se adelantó en el capítulo *4.3. Comparación RTP con SRTP.*, sin embargo, en ese mismo apartado, se comunicó que las tramas tenían un valor de 192 y 202 bytes y en las capturas de *Figura 6.1.a* y *Figura 6.2.a.*, aparecen 12 bytes mayor. Esto es debido a que en *Wireshark*, al capturar los paquetes, añade 12 bytes a cada uno, pero el estudio se ha realizado con la longitud de los paquetes reales que circulan por la red y no por los capturados.

```
 5 0.700622  172.16.32.243   RTP          214 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26501, Time=160
 6 0.721009  172.16.32.243   RTP          214 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26502, Time=320
 7 0.741947  172.16.32.243   RTP          214 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26503, Time=480
 8 0.758081  172.16.32.243   RTP          214 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26501, Time=160
 9 0.762947  172.16.32.243   RTP          214 PT=ITU-T G.711 PCMA, SSRC=0x294823, Seq=26504, Time=640
```

Figura 6.2.a.: Captura de tramas SRTP decodificados.

Reproducimos la señal que aparece en la *Figura 6.3.a.*, y se escucha ruido blanco de fondo, es decir, que se ha conseguido codificar la señal durante el trayecto de la red entre ambos extremos, y debido a que en ambos extremos también se escucha el audio a la perfección, se verifica que este cifrado es válido y funcional para poder seguir adelante en este proyecto.

6.3. Tramas DTLS.

En este apartado, se analiza la negociación de las claves de SRTP a través del protocolo DTLS, la captura de estos paquetes se puede observar en la *Figura 6.3.a.*, se verán en profundidad los 4 paquetes que componen las 7 fases de la negociación vistas en el capítulo *4.4. Uso del DTLS*. Siendo el primer paquete correspondiente a la fase 1, el segundo a las fases 2 y 3, el tercero corresponde a las fases 4, 5, 6 y 7 y el cuarto a las fases 5, 6 y 7.

1	0.000000	172.16.32.243	DTLSv1.2	254	Client Hello
2	0.498082	172.16.32.243	DTLSv1.2	1404	Server Hello, Certificate (Fragment), Certificate (Fragment)
3	0.508169	172.16.32.243	DTLSv1.2	1277	Certificate (Fragment), Certificate (Fragment), Certificate
4	0.510164	172.16.32.243	DTLSv1.2	1077	New Session Ticket (Fragment), New Session Ticket (Fragment)

Figura 6.3.a.: Paquetes capturados de la negociación DTLS.

Los 4 paquetes representados, están divididos por campos que se pueden observar en el *10. Anexo 1: Capturas de los paquetes DTLS.*

El primer paquete corresponde al *ClientHello*. Está compuesto por 242 bytes, ya que hay que restar los bytes añadidos por la captura de Wireshark. En el primer campo se define el tipo *ClientHello* y en el siguiente se añade un número *Random* que corresponde al *nonce*. Seguidamente aparece la lista de 28 *Cipher Suites* que soporta durante la negociación el cliente, que es el *Line Dispatcher*, y define el método de compresión por defecto. Se envían también 3 formatos de *ec_point* (curvas elípticas) que sirven para la elaboración de la clave, acompañado de 5 *supported groups* que se utilizan para acordar que *ec_point* se utiliza en la comunicación y cuales son compatibles. Por último aparece la inclusión del protocolo SRTP con los dos *Cipher Suites* soportados en la comunicación y una lista de 23 firmas *Hash* que aseguran la autentificación del mensaje. Se han enviado listas con todas las opciones posibles que tiene el cliente, para que en la respuesta *ServerHello*, el servidor elija cuál de todas las opciones quiere utilizar.

El segundo paquete, corresponde al *ServerHello*, que en este caso, tiene una longitud de 1392 bytes que se compone de los distintos campos.

En el primer campo se define el tipo *ServerHello* del *Handshake*, a continuación se envía el número *nonce* diferente al del cliente y se establece el *Cipher Suite* que se emplea en la negociación: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030), que suite utiliza el intercambio de claves ECDHE (*Elliptic Curve Diffie-Hellman Ephemeral*) con curvas elípticas, autenticación del servidor mediante RSA (), cifrado simétrico AES con una clave de 256 bits y el modo de operación GCM (*Galois/Counter Mode*) para proporcionar cifrado y autenticación de integridad. La función de hash SHA-384 (*Secure Hash Algorithm 384*) se utiliza para garantizar la integridad de los datos. A continuación se indica el protocolo SRTP, a la vez que se escoge el *Cipher Suite* de la conversación: SRTP_AES128_CM_HMAC_SHA1_80 (0x0001) explicado en el capítulo *4.4. Uso del DTLS.*, posteriormente, indica el id del certificado perteneciente a PJMedia y algunos campos como la fecha de validez, el número de secuencia, la id del algoritmo y su clave

pública. Se envían también 3 tipos de certificados de los tipos RSA, DDS y ECDSA y la misma lista de 23 algoritmos *Hash* que había enviado el cliente.

El tercer paquete se compone de 1265 bytes que envía el cliente al servidor. En el primer campo se define el tipo de *Handshake*, más adelante envía el certificado a nombre de PJSIP y de PJMedia con su firma, id, nombre, número de secuencia, tiempo de validez y clave pública. Envía también la clave pública de *Diffie-Hellman* y posteriormente el *Hash* para la verificación del paquete.

Por último, en el cuarto paquete, el servidor envía 1065 bytes al cliente.

Tras verificar los datos enviados por el cliente, el servidor inicializa una nueva sesión con un tiempo de vida de 2 horas, tal y como se especifica en el campo *TLS Session Ticket*. Completando con este paquete así la negociación de DTLS y obteniendo las claves para el cifrado SRTP.

6.4. Demostración de la seguridad de la comunicación de audio.

En primer lugar, observamos la señal obtenida en el primer escenario planteado en el capítulo *5.1. Implementación de la librería PJMedia en un prototipo.*, a través del uso de la aplicación *Wireshark*, se realiza un análisis mediante una captura de la conversación a través de RTP, lo que permite observar el proceso de envío de los paquetes codificados (pero no cifrados) utilizando el protocolo G711 a través de UDP. Este análisis se visualiza en la *Figura 6.4.a*.

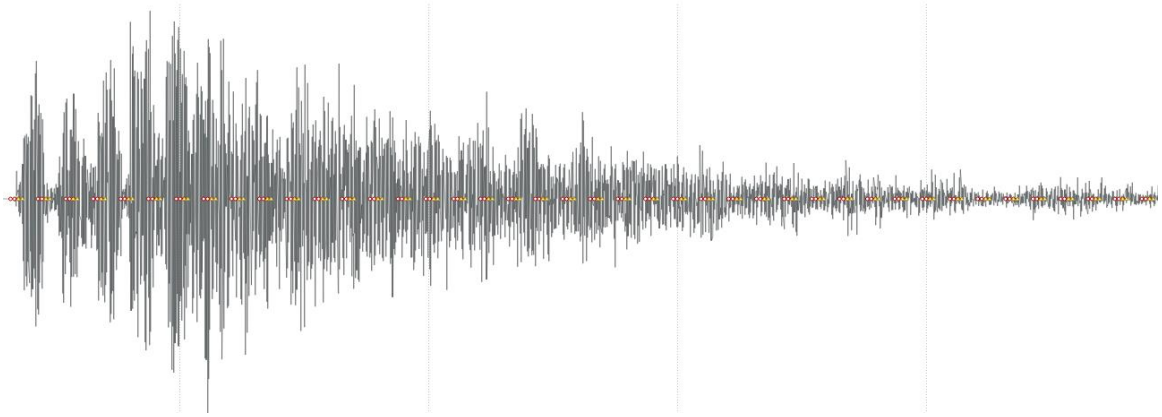


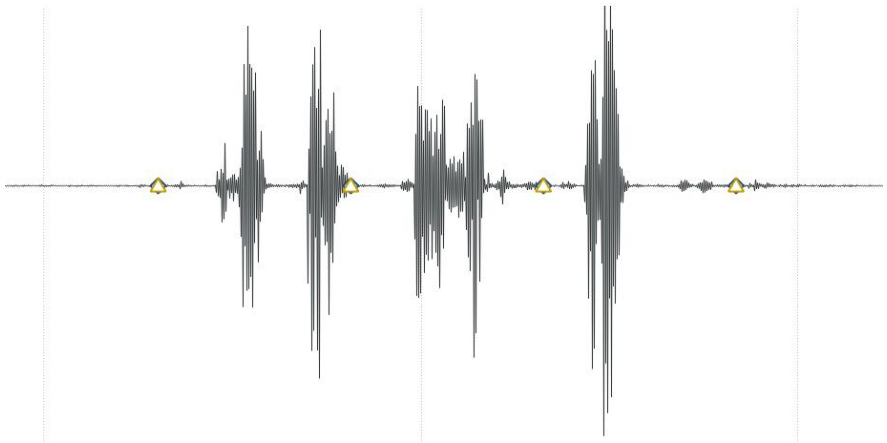
Figura 6.4.a.: Señal de audio codificada por RTP sin cifrado hallada por la aplicación Wireshark de la trama del apartado 6.1. Tramas RTP.

Es importante destacar que, debido a la ausencia de cifrado en la señal de audio transmitida, esta puede ser escuchada al decodificarla. Aunque el protocolo utilizado ha demostrado ser funcional en este caso, su falta de seguridad es una preocupación clave. Debido a que intrusos que tengan acceso a la red en la que existe la comunicación,

pueden escucharla utilizando programas como *Wireshark* de una manera muy sencilla, tal y como se acaba de demostrar.

Se puede averiguar esto, debido a que la señal de la *Figura 6.4.a.* tiene una forma en la que se diferencian varias intensidades, siendo que en la primera parte es más intensa y en la última es menos intensa, pero en ningún momento llega a haber silencio. Al reproducirla, tal y como se explica en el apartado *6.1. Tramas RTP.,* la hipótesis planteada es cierta.

En el siguiente ejemplo mostrado en la *Figura 6.4.b.* se puede observar una señal de conversación más realista, debido a que para ello se ha capturado unos segundos de la conversación realizada en el segundo escenario explicado en el capítulo *5.2. Migración del Line Dispatcher a PJMedia y conexión con la radio.*, en el que es una conversación a tiempo real cuyos extremos son el *Line Dispatcher* y el sistema de radio. Se pueden observar los tiempos de silencio y los tiempos en el que se habla.



6.4.b.: Conversación a tiempo real entre dos usuarios situados en el Line Dispatcher y en el sistema de radio.

Por último, se observa en la *Figura 6.4.c.*, que los tiempos de silencio han desaparecido, y que incluso la señal tiene siempre la misma intensidad. Esto es debido a que esta señal está cifrada debido a que pertenece al escenario del capítulo *5.3. Implementación del protocolo SRTP en el prototipo.*, por lo que simula un ruido blanco que al escucharlo, se puede apreciar únicamente ruido, por lo que la finalidad estaría realizada, ya que nadie que tenga acceso a cualquiera de las redes implicadas en la conversación que no tenga la red privada de cada uno, no podrá descifrar la señal, por lo que evitamos así posibles intrusos que comprometen la seguridad de la llamada.

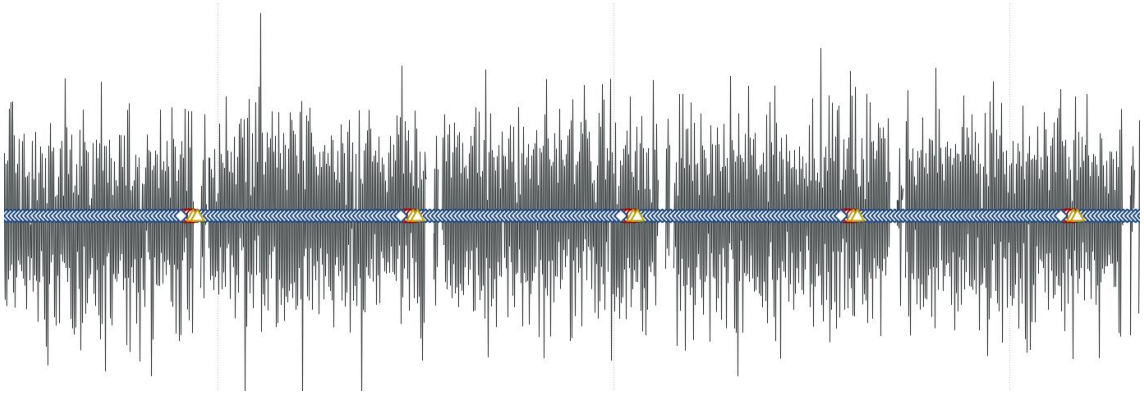


Figura 6.4.c.: Señal de audio codificada por RTP y cifrada por SRTP.

Gracias a este capítulo se ha observado que la comunicación, aparte de ser funcional tal y como se había asegurado en el capítulo *5. Escenarios realizados.*, que los dos últimos apartados funcionan correctamente con SRTP lo que hace que la señal esté cifrada que era el objetivo principal de este proyecto.

7. Problemas resueltos durante el desarrollo.

A lo largo de este capítulo se describirán los problemas que han surgido a lo largo de todo el proceso del proyecto.

En primer lugar, se quería resolver la falta de seguridad en el protocolo RTP cambiándolo a SRTP añadiendo también DTLS para la negociación de las claves, tal y como se ha explicado durante los capítulos anteriores.

Para Implementar estos dos protocolos, se han tenido que buscar distintas librerías que fuesen compatibles con las que ya se utilizan privadamente en la empresa en la que se ha desarrollado el proyecto. La primera librería que se ha pensado es PJSIP, ya que engloba el protocolo SRTP, pero daba problemas de compatibilidad con la red y otras librerías de la empresa, por lo que se tuvo que llevar al nivel más bajo para evitar estos problemas, así que se decidió utilizar la librería PJMedia apoyada con PJSIP y PJNATH para poder trabajar en esos niveles de acceso de la librería.

Se tuvieron que resolver algunos problemas internos también para dar compatibilidad a estas librerías.

Una vez resueltos los temas de compatibilidad con la librería interna, se podía utilizar el protocolo RTP, lo único que solo había un par de ejemplos por internet de cómo utilizar estas librerías a través del tipo *Stream*, pero este tipo es de un nivel demasiado alto y no se podía utilizar, por lo que se ha desarrollado tal y como se explica en el capítulo *4.5. Código de RTP.* en el que se ha investigado lo que hace el tipo *Stream* internamente y se ha modificado para poder adaptarse a lo que se necesita en el proyecto que es del tipo *Transport* (que se usa para el nivel más bajo que es el necesario).

Así se ha podido desarrollar el primer escenario descrito en el capítulo *5.1. Implementación de la librería PJMedia en un prototipo.*

Posteriormente para desarrollar el escenario *5.2. Migración del Line Dispatcher a PJMedia y conexión con la radio.* se ha modificado una aplicación ya existente que maneja el *Line Dispatcher* para comunicarse con el sistema de radio real introducida en la red TETRA. No han aparecido problemas de compatibilidad ya que se resolvieron para el anterior escenario.

Para implementar SRTP para el tercer escenario explicado en el capítulo *5.3. Implementación del protocolo SRTP en el prototipo.* se ha estudiado la utilidad de cada función PJMedia que conlleva este protocolo en la página oficial de la librería, ya que al igual que ocurrió con RTP, tampoco había ejemplos a nivel tan bajo de acceso a la librería. A la hora de implementar las claves, se quería utilizar la negociación, para que así si se averiguase una clave, no se pudiese descifrar las demás conversaciones de esta, y que en cada comunicación hubiera claves distintas. Para ello se utilizó el protocolo DTLS por lo que la librería OpenSSL fue necesaria para ello.

Se volvieron a tener problemas de compatibilidad al implementar esta última librería pero se resolvieron al modificar temas internos de la empresa.

Para desarrollar el último escenario desarrollado en el capítulo *5.4. Implementación de Comunicación Segura en Red TETRA mediante Simulación SRTP/DTLS.*, en un principio, la idea era probarlo directamente con la radio de la red TETRA, pero debido a que el *Gateway* no aceptaba los protocolos DTLS y SRTP, no se pudo hacer, ya que se estimaba que sería más sencillo modificarlo pero llevaba mucho más tiempo del esperado, por lo que se prefirió hacer una simulación y así poder demostrar que las combinaciones creadas fuesen funcionales y seguras antes de modificar la aplicación del *Gateway*.

8. Conclusión y líneas futuras.

En este capítulo, se presenta el resultado final del proyecto y se explora su potencial para futuros desarrollos. A lo largo del desarrollo de este TFG, se ha logrado demostrar la viabilidad de una comunicación en tiempo real funcional. Además, mediante la implementación de los protocolos SRTP y DTLS, se ha demostrado la posibilidad de establecer una conexión segura, íntegra y auténtica en una red TETRA de comunicación de radio, tal como se ilustra en el segundo escenario detallado en el capítulo *5.2. Migración del Line Dispatcher a PJMedia y Conexión con la Radio*.

El enfoque de desarrollo se ha mantenido al nivel más bajo de acceso a la librería para asegurar la compatibilidad con diversas librerías y redes, tanto de la empresa como de los clientes que podrían emplear esta tecnología en sus propios proyectos.

La línea futura de este proyecto se basa en la adaptación del Gateway, como se ha mencionado en distintas secciones de este informe. La modificación del Gateway es esencial para facilitar la efectiva incorporación de esta tecnología en la red. Actualmente, el nodo no es compatible con los protocolos SRTP y DTLS, lo que limita su capacidad para transformar tramas entre los formatos ACELP y los protocolos mencionados. La adaptación del Gateway para que soporte estos protocolos representa un paso esencial para lograr la integración completa y funcionalidad óptima de la solución dentro del sistema de radio en la red TETRA.

9. Bibliografía.

[1]: Librería PJSIP: https://www.pjsip.org/

[2]: Librería PJNATH: https://www.pjsip.org/pjnath/docs/html/

[3]: Librería PJMedia: https://www.pjsip.org/pjmedia/docs/html/index.htm

[4]: Librería de OpenSSL: https://www.openssl.org/

[5]: Protocolo DTLS RFC 9147: https://datatracker.ietf.org/doc/rfc9147/

10. Anexo 1: Capturas de los paquetes DTLS.

10.1. Composición *ClientHello.*

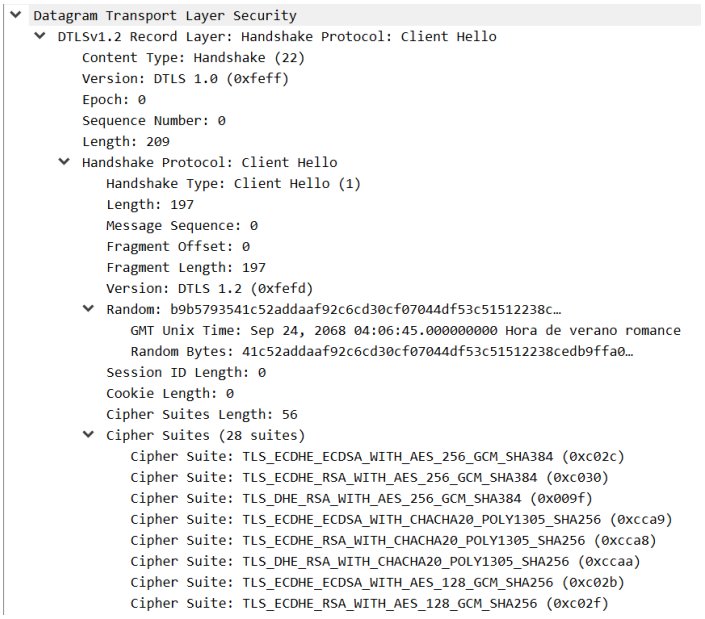


Figura 10.1.a.: Composición ClientHello (1).

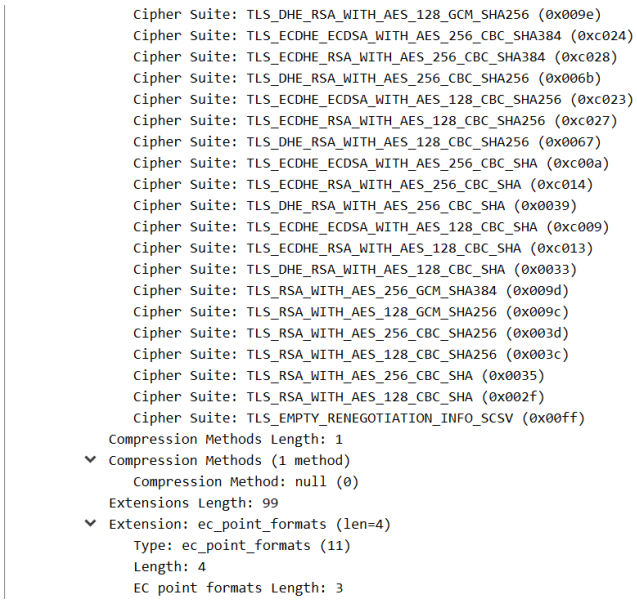


Figura 10.1.b.: Composición ClientHello (2).

```
    Elliptic curves point formats (3)
        EC point format: uncompressed (0)
        EC point format: ansiX962_compressed_prime (1)
        EC point format: ansiX962_compressed_char2 (2)
  Extension: supported_groups (len=12)
      Type: supported_groups (10)
      Length: 12
      Supported Groups List Length: 10
    Supported Groups (5 groups)
        Supported Group: x25519 (0x001d)
        Supported Group: secp256r1 (0x0017)
        Supported Group: x448 (0x001e)
        Supported Group: secp521r1 (0x0019)
        Supported Group: secp384r1 (0x0018)
  Extension: session_ticket (len=0)
      Type: session_ticket (35)
      Length: 0
      Data (0 bytes)
  Extension: use_srtp (len=7)
      Type: use_srtp (14)
      Length: 7
      SRTP Protection Profiles Length: 4
      SRTP Protection Profile: SRTP_AES128_CM_HMAC_SHA1_80 (0x0001)
      SRTP Protection Profile: SRTP_AES128_CM_HMAC_SHA1_32 (0x0002)
      MKI Length: 0
  Extension: encrypt_then_mac (len=0)
      Type: encrypt_then_mac (22)
      Length: 0
```

Figura 10.1.c.: Composición ClientHello (3).

```
    Extension: extended_master_secret (len=0)
        Type: extended_master_secret (23)
        Length: 0
    Extension: signature_algorithms (len=48)
        Type: signature_algorithms (13)
        Length: 48
        Signature Hash Algorithms Length: 46
      Signature Hash Algorithms (23 algorithms)
        Signature Algorithm: ecdsa_secp256r1_sha256 (0x0403)
        Signature Algorithm: ecdsa_secp384r1_sha384 (0x0503)
        Signature Algorithm: ecdsa_secp521r1_sha512 (0x0603)
        Signature Algorithm: ed25519 (0x0807)
        Signature Algorithm: ed448 (0x0808)
        Signature Algorithm: rsa_pss_pss_sha256 (0x0809)
        Signature Algorithm: rsa_pss_pss_sha384 (0x080a)
        Signature Algorithm: rsa_pss_pss_sha512 (0x080b)
        Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
        Signature Algorithm: rsa_pss_rsae_sha384 (0x0805)
        Signature Algorithm: rsa_pss_rsae_sha512 (0x0806)
        Signature Algorithm: rsa_pkcs1_sha256 (0x0401)
        Signature Algorithm: rsa_pkcs1_sha384 (0x0501)
        Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
        Signature Algorithm: SHA224 ECDSA (0x0303)
        Signature Algorithm: ecdsa_sha1 (0x0203)
        Signature Algorithm: SHA224 RSA (0x0301)
        Signature Algorithm: rsa_pkcs1_sha1 (0x0201)
        Signature Algorithm: SHA224 DSA (0x0302)
        Signature Algorithm: SHA1 DSA (0x0202)
        Signature Algorithm: SHA256 DSA (0x0402)
```

Figura 10.1.d: Composición ClientHello (4).

```
        Signature Algorithm: SHA384 DSA (0x0502)
        Signature Algorithm: SHA512 DSA (0x0602)
```

Figura 10.1.e.: Composición ClientHello (5).

10.2. Composición *ServerHello.*

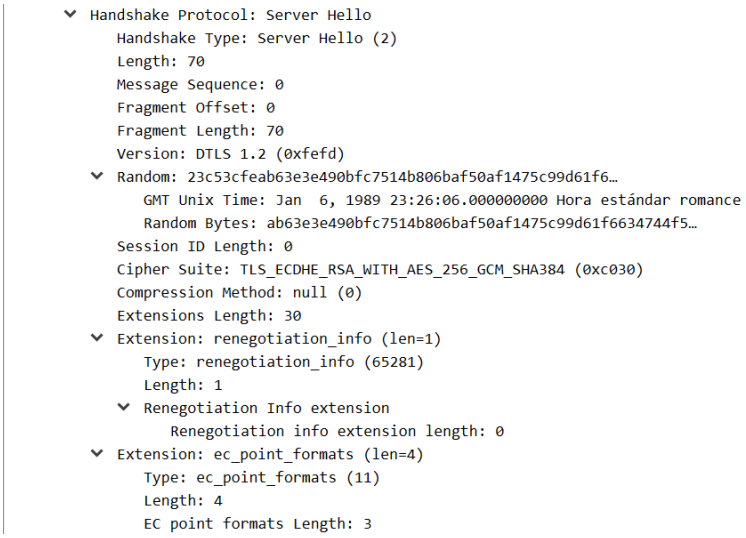


Figura 10.2.a.: Composición ServerHello (1).

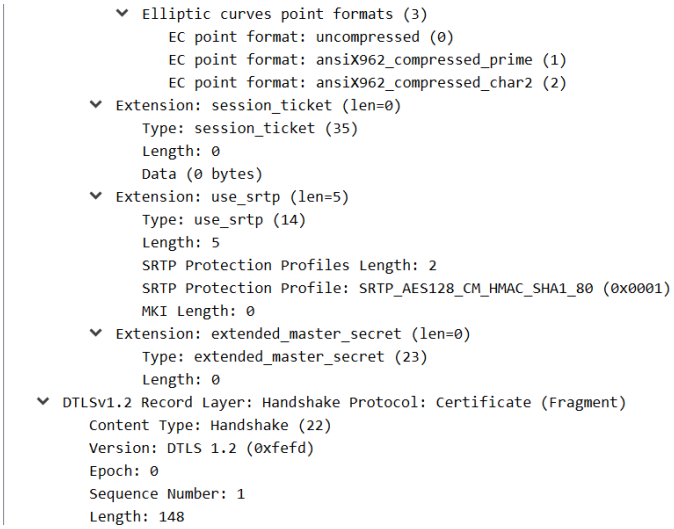


Figura 10.2.b.: Composición ServerHello (2).

```
    Handshake Protocol: Certificate (Fragment)
        Handshake Type: Certificate (11)
        Length: 702
        Message Sequence: 1
        Fragment Offset: 0
        Fragment Length: 136
DTLSv1.2 Record Layer: Handshake Protocol: Certificate (Fragment)
    Content Type: Handshake (22)
    Version: DTLS 1.2 (0xfefd)
    Epoch: 0
    Sequence Number: 2
    Length: 243
    Handshake Protocol: Certificate (Fragment)
        Handshake Type: Certificate (11)
        Length: 702
        Message Sequence: 1
        Fragment Offset: 136
        Fragment Length: 231
DTLSv1.2 Record Layer: Handshake Protocol: Certificate (Fragment)
    Content Type: Handshake (22)
    Version: DTLS 1.2 (0xfefd)
    Epoch: 0
    Sequence Number: 3
    Length: 243
```

Figura 10.2.c.: Composición ServerHello (3).

```
    Handshake Protocol: Certificate (Fragment)
        Handshake Type: Certificate (11)
        Length: 702
        Message Sequence: 1
        Fragment Offset: 367
        Fragment Length: 231
DTLSv1.2 Record Layer: Handshake Protocol: Certificate (Reassembled)
    Content Type: Handshake (22)
    Version: DTLS 1.2 (0xfefd)
    Epoch: 0
    Sequence Number: 4
    Length: 116
    Handshake Protocol: Certificate (Reassembled)
        Handshake Type: Certificate (11)
        Length: 702
        Message Sequence: 1
        Fragment Offset: 598
        Fragment Length: 104
        Certificates Length: 699
        Certificates (699 bytes)
            Certificate Length: 696
            Certificate: 308202b43082019ca003020102020448805113300d06092a… (id-at-commonName=pjmedia.pjsip.org)
                signedCertificate
                    version: v3 (2)
                    serialNumber: 1216368915
```

Figura 10.2.d.: Composición ServerHello (4).

```
                signature (sha1WithRSAEncryption)
                    Algorithm Id: 1.2.840.113549.1.1.5 (sha1WithRSAEncryption)
                issuer: rdnSequence (0)
                    rdnSequence: 1 item (id-at-commonName=pjmedia.pjsip.org)
                        RDNSequence item: 1 item (id-at-commonName=pjmedia.pjsip.org)
                            RelativeDistinguishedName item (id-at-commonName=pjmedia.pjsip.org)
                                Id: 2.5.4.3 (id-at-commonName)
                                DirectoryString: uTF8String (4)
                                    uTF8String: pjmedia.pjsip.org
                validity
                    notBefore: utcTime (0)
                        utcTime: 23-07-20 10:44:58 (UTC)
                    notAfter: utcTime (0)
                        utcTime: 24-07-20 10:44:58 (UTC)
                subject: rdnSequence (0)
                    rdnSequence: 1 item (id-at-commonName=pjmedia.pjsip.org)
                        RDNSequence item: 1 item (id-at-commonName=pjmedia.pjsip.org)
                            RelativeDistinguishedName item (id-at-commonName=pjmedia.pjsip.org)
                                Id: 2.5.4.3 (id-at-commonName)
                                DirectoryString: uTF8String (4)
                                    uTF8String: pjmedia.pjsip.org
                subjectPublicKeyInfo
                    algorithm (rsaEncryption)
                        Algorithm Id: 1.2.840.113549.1.1.1 (rsaEncryption)
```

Figura 10.2.e.: Composición ServerHello (5).

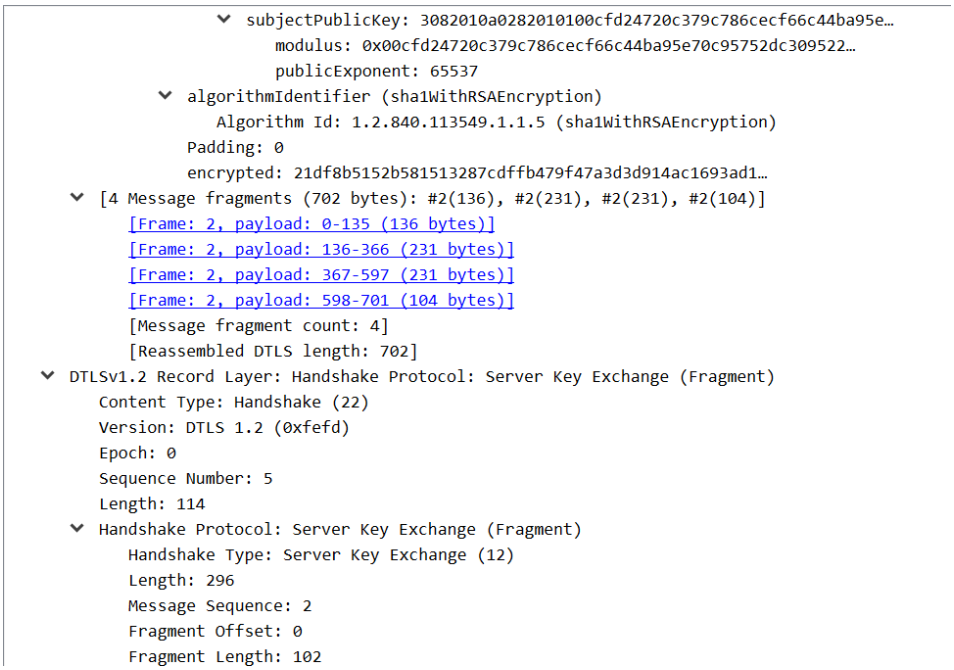


Figura 10.2.f.: Composición ServerHello (6).

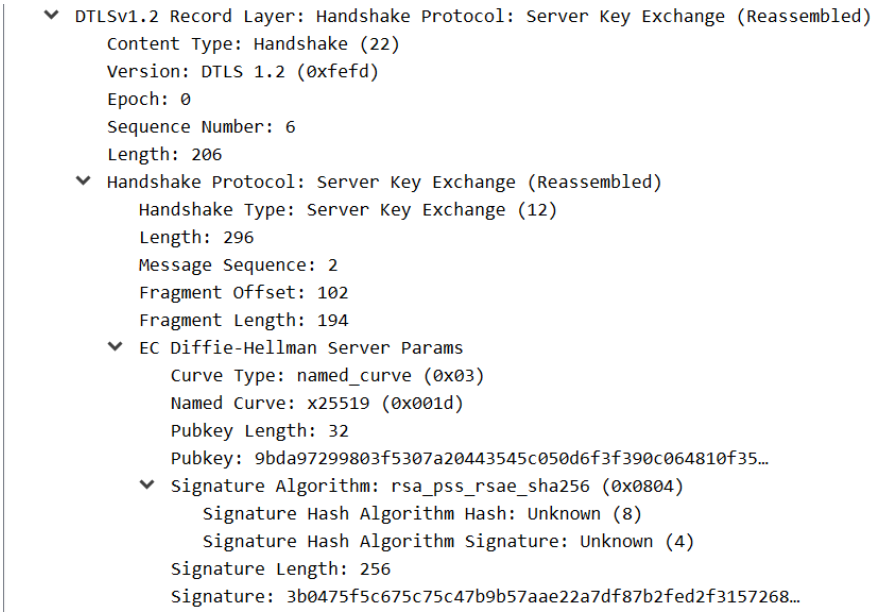


Figura 10.2.g.: Composición ServerHello (7).

```
    [2 Message fragments (296 bytes): #2(102), #2(194)]
        [Frame: 2, payload: 0-101 (102 bytes)]
        [Frame: 2, payload: 102-295 (194 bytes)]
        [Message fragment count: 2]
        [Reassembled DTLS length: 296]
DTLSv1.2 Record Layer: Handshake Protocol: Certificate Request (Fragment)
    Content Type: Handshake (22)
    Version: DTLS 1.2 (0xfefd)
    Epoch: 0
    Sequence Number: 7
    Length: 24
  Handshake Protocol: Certificate Request (Fragment)
      Handshake Type: Certificate Request (13)
      Length: 54
      Message Sequence: 3
      Fragment Offset: 0
      Fragment Length: 12
DTLSv1.2 Record Layer: Handshake Protocol: Certificate Request (Reassembled)
    Content Type: Handshake (22)
    Version: DTLS 1.2 (0xfefd)
    Epoch: 0
    Sequence Number: 8
    Length: 54
```

Figura 10.2.h.: Composición ServerHello (8).

```
  Handshake Protocol: Certificate Request (Reassembled)
      Handshake Type: Certificate Request (13)
      Length: 54
      Message Sequence: 3
      Fragment Offset: 12
      Fragment Length: 42
      Certificate types count: 3
    Certificate types (3 types)
        Certificate type: RSA Sign (1)
        Certificate type: DSS Sign (2)
        Certificate type: ECDSA Sign (64)
      Signature Hash Algorithms Length: 46
    Signature Hash Algorithms (23 algorithms)
```

Figura 10.2.i.: Composición ServerHello (9).

```
  [2 Message fragments (54 bytes): #2(12), #2(42)]
      [Frame: 2, payload: 0-11 (12 bytes)]
      [Frame: 2, payload: 12-53 (42 bytes)]
      [Message fragment count: 2]
      [Reassembled DTLS length: 54]
DTLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
    Content Type: Handshake (22)
    Version: DTLS 1.2 (0xfefd)
    Epoch: 0
    Sequence Number: 9
    Length: 12
  Handshake Protocol: Server Hello Done
      Handshake Type: Server Hello Done (14)
      Length: 0
      Message Sequence: 4
      Fragment Offset: 0
      Fragment Length: 0
```

Figura 10.2.j.: Composición ServerHello (10).

10.3. Composición del tercer paquete.

Figura 10.3.a.: Composición tercer paquete (1).

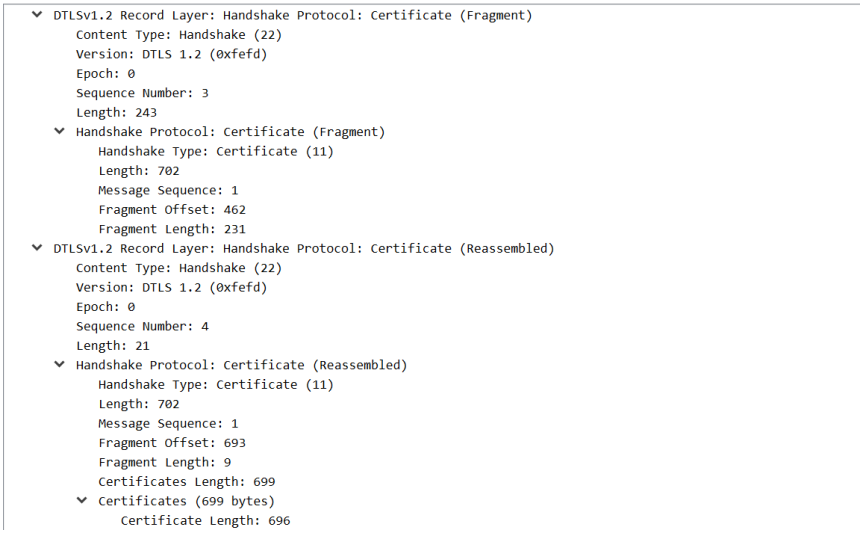


Figura 10.3.b.: Composición tercer paquete (2).

```
Certificate: 308202b43082019ca003020102020448805113300d06092a… (id-at-commonName=pjmedia.pjsip.org)
  signedCertificate
      version: v3 (2)
      serialNumber: 1216368915
    signature (sha1WithRSAEncryption)
       Algorithm Id: 1.2.840.113549.1.1.5 (sha1WithRSAEncryption)
    issuer: rdnSequence (0)
      rdnSequence: 1 item (id-at-commonName=pjmedia.pjsip.org)
        RDNSequence item: 1 item (id-at-commonName=pjmedia.pjsip.org)
          RelativeDistinguishedName item (id-at-commonName=pjmedia.pjsip.org)
              Id: 2.5.4.3 (id-at-commonName)
            DirectoryString: uTF8String (4)
                uTF8String: pjmedia.pjsip.org
    validity
      notBefore: utcTime (0)
          utcTime: 23-07-20 10:44:58 (UTC)
      notAfter: utcTime (0)
          utcTime: 24-07-20 10:44:58 (UTC)
    subject: rdnSequence (0)
      rdnSequence: 1 item (id-at-commonName=pjmedia.pjsip.org)
        RDNSequence item: 1 item (id-at-commonName=pjmedia.pjsip.org)
          RelativeDistinguishedName item (id-at-commonName=pjmedia.pjsip.org)
              Id: 2.5.4.3 (id-at-commonName)
            DirectoryString: uTF8String (4)
                uTF8String: pjmedia.pjsip.org
    subjectPublicKeyInfo
      algorithm (rsaEncryption)
          Algorithm Id: 1.2.840.113549.1.1.1 (rsaEncryption)
```

Figura 10.3.c.: Composición tercer paquete (3).

```
          subjectPublicKey: 3082010a0282010100df21361aaec3e31f58f6d5aed98aa2…
              modulus: 0x00df21361aaec3e31f58f6d5aed98aa28411d6b284e68b08…
              publicExponent: 65537
      algorithmIdentifier (sha1WithRSAEncryption)
          Algorithm Id: 1.2.840.113549.1.1.5 (sha1WithRSAEncryption)
      Padding: 0
      encrypted: cc19f16f9bb101d07ae18770c5d75009ed8ca9213eeef4d2…
[4 Message fragments (702 bytes): #3(231), #3(231), #3(231), #3(9)]
    [Frame: 3, payload: 0-230 (231 bytes)]
    [Frame: 3, payload: 231-461 (231 bytes)]
    [Frame: 3, payload: 462-692 (231 bytes)]
    [Frame: 3, payload: 693-701 (9 bytes)]
    [Message fragment count: 4]
    [Reassembled DTLS length: 702]
DTLSv1.2 Record Layer: Handshake Protocol: Client Key Exchange
  Content Type: Handshake (22)
  Version: DTLS 1.2 (0xfefd)
  Epoch: 0
  Sequence Number: 5
  Length: 45
Handshake Protocol: Client Key Exchange
    Handshake Type: Client Key Exchange (16)
    Length: 33
    Message Sequence: 2
    Fragment Offset: 0
    Fragment Length: 33
```

Figura 10.3.d.: Composición tercer paquete (4).

```
      EC Diffie-Hellman Client Params
         Pubkey Length: 32
         Pubkey: 56a60b5f4be0d6c3f1aca60d1250b027a9fba2cf33c0a448…
DTLSv1.2 Record Layer: Handshake Protocol: Certificate Verify (Fragment)
   Content Type: Handshake (22)
   Version: DTLS 1.2 (0xfefd)
   Epoch: 0
   Sequence Number: 6
   Length: 151
   Handshake Protocol: Certificate Verify (Fragment)
      Handshake Type: Certificate Verify (15)
      Length: 260
      Message Sequence: 3
      Fragment Offset: 0
      Fragment Length: 139
DTLSv1.2 Record Layer: Handshake Protocol: Certificate Verify (Reassembled)
   Content Type: Handshake (22)
   Version: DTLS 1.2 (0xfefd)
   Epoch: 0
   Sequence Number: 7
   Length: 133
   Handshake Protocol: Certificate Verify (Reassembled)
      Handshake Type: Certificate Verify (15)
      Length: 260
      Message Sequence: 3
      Fragment Offset: 139
      Fragment Length: 121
      Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
```

Figura 10.3.e.: Composición tercer paquete (5).

```
      Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
         Signature Hash Algorithm Hash: Unknown (8)
         Signature Hash Algorithm Signature: Unknown (4)
      Signature length: 256
      Signature: 4b1fab12aab91607015408427f68ad3e4db3012084f8ecf0…
   [2 Message fragments (260 bytes): #3(139), #3(121)]
      [Frame: 3, payload: 0-138 (139 bytes)]
      [Frame: 3, payload: 139-259 (121 bytes)]
      [Message fragment count: 2]
      [Reassembled DTLS length: 260]
DTLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
   Content Type: Change Cipher Spec (20)
   Version: DTLS 1.2 (0xfefd)
   Epoch: 0
   Sequence Number: 8
   Length: 1
   Change Cipher Spec Message
Record Layer
   Content Type: Handshake (22)
   Version: DTLS 1.2 (0xfefd)
   Epoch: 1
   Sequence Number: 0
   Length: 48
   Handshake Protocol
```

Figura 10.3.f.: Composición tercer paquete (6).

10.4. Composición del cuarto paquete.

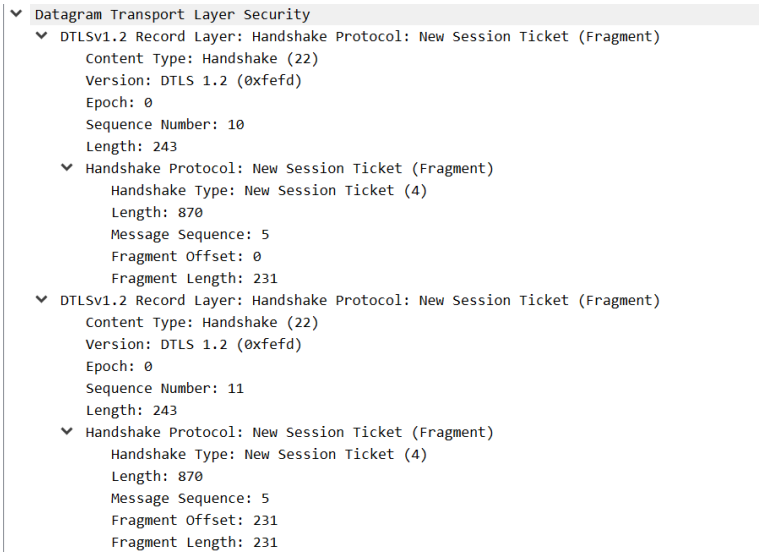


Figura 10.4.a.: Composición cuarto paquete (1).

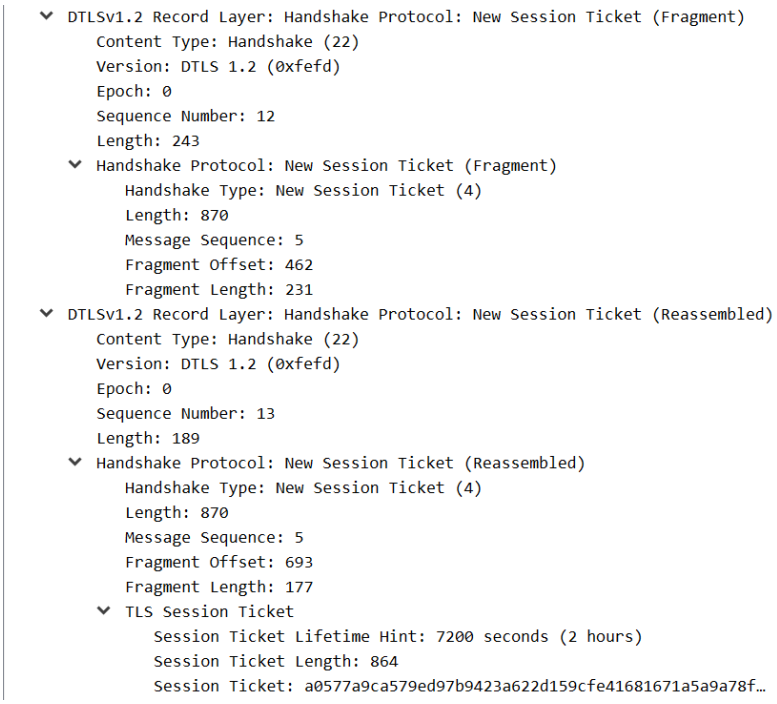


Figura 10.4.b.: Composición cuarto paquete (2).

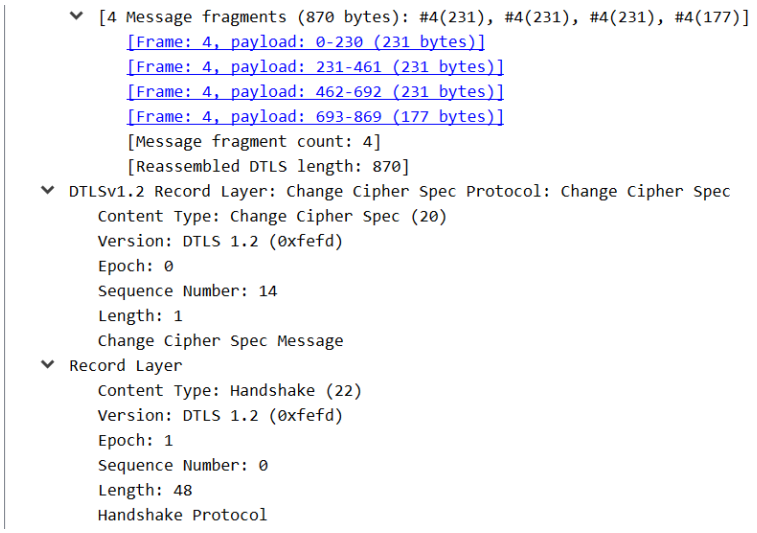


Figura 10.4.c.: Composición cuarto paquete (3).