Universidad de Oviedo

ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN

GRADO EN INGENIERÍA EN TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

ÁREA DE INGENIERÍA TELEMÁTICA

Evaluación e implementación del Internet Draft EAP-based Authentication Service for CoAP

D. Menéndez González, Daniel
TUTOR: D. Dan García Carrillo

FECHA: Enero de 2025

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ECUACIONES

1. Introducción.

Este Trabajo de Fin de Grado titulado "Evaluación e implementación del Internet Draft EAP-based Authentication Service for CoAP" aborda una temática fundamental en el campo del Internet of Things (IoT): el control de acceso a la red [1]. A medida que la conectividad se ha expandido de los dispositivos tradicionales a una amplia gama de dispositivos inteligentes, la seguridad y la autenticación de estos dispositivos se ha vuelto un desafío clave. En el contexto del IoT, el control de acceso no solo garantiza la protección de los dispositivos conectados, sino que también asegura la integridad y privacidad de los datos que estos manejan, protegiéndolos frente a accesos no autorizados.

El IoT abarca una amplia variedad de dispositivos, que van desde sensores y actuadores en entornos industriales hasta dispositivos en hogares inteligentes como cámaras, termostatos y electrodomésticos. Esta diversidad de dispositivos, junto con la variedad de tecnologías de comunicación que emplean, plantea exigencias muy particulares a los métodos de autenticación y control de acceso a la red. A diferencia de los dispositivos tradicionales como ordenadores o smartphones, los dispositivos IoT suelen contar con limitaciones significativas en términos de recursos, como capacidad de procesamiento, memoria y consumo energético. Estos factores hacen que los métodos de autenticación convencionales, diseñados para entornos más robustos, no sean adecuados para el IoT. [2]

En este escenario, surge la necesidad de soluciones de autenticación que sean tanto flexibles como eficientes, capaces de adaptarse a las diversas capacidades de los dispositivos IoT, así como a las restricciones de las redes de baja potencia y ancho de banda limitado. El protocolo CoAP (Constrained Application Protocol) [3], propuesto por el IETF, se ha diseñado específicamente para estos entornos con restricciones, permitiendo la comunicación ligera y eficiente entre dispositivos IoT. Sin embargo, a pesar de que CoAP aborda los problemas de comunicación en redes con restricciones, el desafío de implementar un sistema de autenticación robusto y eficiente sigue siendo una cuestión abierta.

Este trabajo se enfoca en la evaluación e implementación del Internet Draft EAP-based Authentication Service for CoAP, una propuesta del IETF que busca proporcionar un

mecanismo de autenticación adaptado a las necesidades específicas del IoT. El uso del protocolo EAP (Extensible Authentication Protocol) en combinación con CoAP es una de las propuestas más recientes para permitir la autenticación flexible en entornos IoT. EAP es un marco ampliamente utilizado en redes inalámbricas y conexiones punto a punto, lo que lo convierte en una opción interesante para el IoT debido a su capacidad para soportar diversos métodos de autenticación.

1.1.- OBJETIVOS Y ALCANCE

El objetivo principal de este trabajo es implementar en Python la última versión del Internet Draft EAP-based Authentication Service for CoAP, empleando el modelo pass-through de su arquitectura. Este modelo se caracteriza por la separación de roles y la interacción entre tres entidades principales: el EAP Peer, el EAP Authenticator y el EAP Server. Como se puede observar en la figura 1.1 cada una de estas entidades sigue un esquema en capas que proporciona modularidad y facilita la implementación de los diferentes elementos del protocolo.

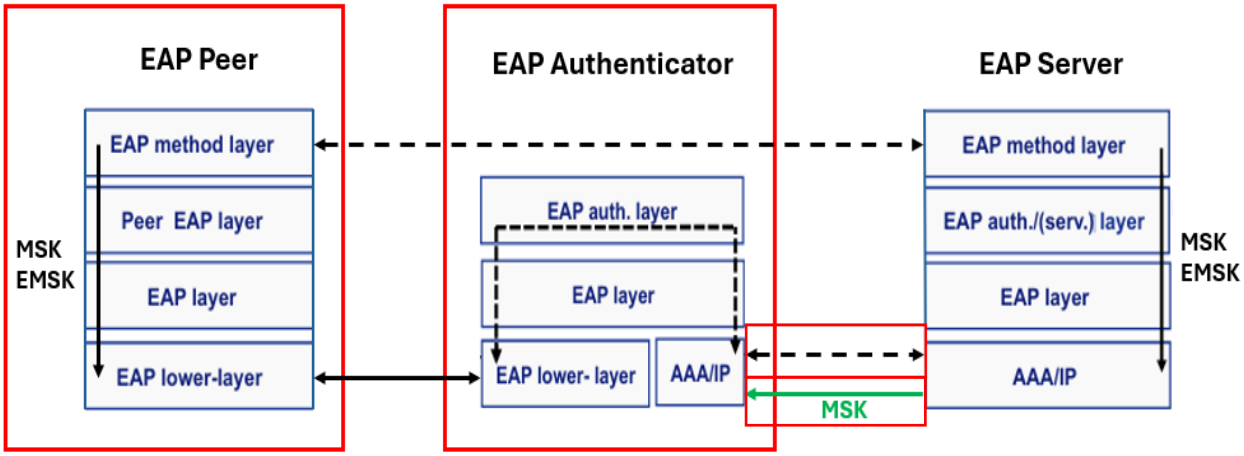


Figura 1.1.- Arquitectura en capas del modelo pass-through de CoAP-EAP

El desarrollo de este objetivo general se puede desglosar en los siguientes objetivos específicos, que abarcan las partes resaltadas en color rojo en la figura:

1. **Implementación del EAP Peer para CoAP-EAP:** Este objetivo aborda la creación de todas las capas necesarias que componen el stack del EAP Peer. Este componente representa al cliente o dispositivo que busca autenticarse. Las capas que se implementarán son:

    - **EAP Method Layer:** Es la capa más alta del EAP Peer y se encarga de implementar la parte del cliente del método EAP-PSK. Se encargará de generar las claves, así como de la construcción y validación de los mensajes específicos del método.
    - **Peer EAP Layer:** Gestiona la interacción entre el EAP Peer y el método EAP. Encapsula el método EAP-PSK para que pueda ser transportado en el protocolo subyacente.
    - **EAP Layer:** Esta capa maneja el flujo genérico de mensajes EAP. Realiza el empaquetado y desempaquetado de los mensajes EAP que serán transmitidos hacia la capa inferior (lower layer).
    - **EAP Lower-Layer:** Es la capa de transporte subyacente que entrega los mensajes EAP al EAP Authenticator. Desarrolla la EAP lower layer conocida como CoAP-EAP empleando la biblioteca aiocoap.

2. **Implementación del EAP Authenticator para CoAP-EAP:** Este objetivo se centra en la creación de todas las capas necesarias que componen el stack del EAP Authenticator. Este componente actúa como como intermediario entre el EAP Peer y el servidor AAA (EAP Server). Las capas a desarrollar son:

    - **EAP Auth. Layer:** Esta capa procesa los mensajes EAP que vienen del EAP Peer. Sin embargo, en el modelo pass-through, el EAP Authenticator no interpreta los detalles del método de autenticación; simplemente pasa los mensajes al servidor AAA.
    - **EAP Layer:** Garantiza el manejo y retransmisión adecuada de los mensajes EAP entre el EAP Peer y el EAP Server.
    - **EAP Lower-Layer:** Es la capa de transporte que entrega los mensajes EAP al EAP Peer. Desarrolla CoAP-EAP empleando la biblioteca aiocoap.

- **AAA/IP:** Esta capa es responsable de comunicar los mensajes EAP hacia el servidor AAA (EAP Server) utilizando el protocolo RADIUS. Aquí es donde se encapsulan los mensajes EAP dentro de paquetes RADIUS para su envío y donde se desencapsulan para su recepción.

3. **Implementación de la interacción entre el cliente AAA y el EAP Server:** Este objetivo incluye la creación y modificación de un cliente RADIUS para gestionar la comunicación entre el EAP Authenticator y el EAP Server. Para ello, se utilizó y adaptó la biblioteca radiustest. Sin embargo, este proceso implicó un gran desafío técnico, ya que fue necesario rehacerla prácticamente desde cero, manteniendo solo las bases iniciales, para conseguir una comunicación efectiva con el servidor RADIUS alojado en la máquina virtual, para incorporar nuevos atributos necesarios como el Message-Authenticator y para mejorar la construcción de los mensajes RADIUS evitando errores en el intercambio.

4. **Descifrado de la MSK:** Este objetivo se basa en la implementación de una función que descifre la MSK enviada por el EAP Server al EAP Authenticator en dos atributos RADIUS (como MS-MPPE-Send-Key y MS-MPPE-Recv-Key). Para ello, se utiliza el secreto compartido RADIUS (Shared Secret) y un proceso basado en MD5 que genera bloques iterativos de descifrado. Una vez descifrada, la MSK sirve como base para derivar el contexto de seguridad OSCORE necesario para proteger las comunicaciones posteriores entre el EAP Peer y el Authenticator de manera segura y eficiente, garantizando la integridad y confidencialidad de los mensajes.

El EAP Server contiene la lógica para autenticar al dispositivo utilizando el método EAP específico. Es el punto final donde ocurre la validación de las credenciales del cliente. Sin embargo, no será implementado en este proyecto, ya que se dispone de una máquina virtual que ya tiene una implementación de FreeRADIUS que soporta EAP-PSK (tiene implementada la parte del servidor del método EAP-PSK).

La combinación de las partes implementadas (destacadas en color rojo) y las preexistentes permite evaluar la propuesta completa del Internet Draft en un entorno controlado, pudiendo realizar pruebas para asegurar que el sistema funciona como un todo.

Además, se abordarán cuestiones relacionadas con la seguridad, la protección de datos y la resiliencia ante ataques. En un contexto donde la conectividad de dispositivos es cada vez más extensa, el aumento de las amenazas de seguridad en IoT hace que la autenticación segura y eficiente sea más crucial que nunca. Los ataques a dispositivos conectados pueden comprometer tanto la privacidad de los usuarios como la integridad de los sistemas en los que operan, lo que convierte la seguridad en una prioridad crítica en el diseño de soluciones IoT.

Este TFG no solo busca desarrollar una implementación funcional del EAP-based Authentication Service for CoAP, sino también contribuir a la discusión más amplia sobre los retos y oportunidades en la autenticación para el IoT. A través de este estudio, se espera aportar al desarrollo de mecanismos de autenticación que puedan garantizar la seguridad en un entorno tan dinámico y limitado como el IoT, ayudando a conformar un futuro más seguro y eficiente para este tipo de redes.

El alcance del trabajo incluye:

- La **contextualización del problema**, donde se proporcionará una revisión exhaustiva de la literatura actual sobre métodos de autenticación en entornos IoT y la importancia del control de acceso. Esto incluirá la identificación de los desafíos específicos que enfrentan los sistemas de autenticación en este contexto, como la diversidad de dispositivos, las limitaciones de recursos y los requisitos de seguridad, así como una discusión sobre las limitaciones y ventajas de los enfoques existentes.

- Una **descripción detallada del Internet Draft EAP-based Authentication Service for CoAP**, explicando en detalle su funcionamiento, incluyendo su arquitectura, la aportación de cada uno de los protocolos involucrados y los mecanismos de seguridad implementados. Se destacarán las características clave de esta solución, de modo que se pueda comprobar el avance que representa en términos de eficiencia, flexibilidad y seguridad en relación con otros enfoques existentes.

- La **implementación** de la última versión del EAP-based Authentication Service for CoAP en un entorno de pruebas utilizando Python, siguiendo las pautas y

recomendaciones de la documentación oficial. Se detallarán los pasos necesarios para la implementación de cada una de las partes de la arquitectura y se describirán los recursos utilizados, con el objetivo de idear una implementación que sea fácil de mantener, renovar y ampliar.

- La **evaluación y el análisis de la solución propuesta**, probando diferentes escenarios para evaluar el correcto funcionamiento de la solución implementada. Además, se comparará la versión actual del Draft con la inicial, se identificarán posibles áreas de mejora y se sugerirán direcciones para investigaciones futuras.

Por otro lado, el alcance de este trabajo no incluye ciertos aspectos:

- E**l desarrollo de nuevos protocolos o estándares de autenticación**, ya que se trata de una tarea compleja que requiere un nivel avanzado de investigación y recursos que exceden el alcance de este TFG. Al centrarse en la implementación y evaluación de un protocolo existente, el trabajo se mantiene manejable y práctico.

- **La implementación en entornos de producción a gran escala**, dado que implicaría coordinar con múltiples partes interesadas y garantizar la compatibilidad con una amplia variedad de sistemas y dispositivos, lo que no es factible dentro del tiempo y los recursos disponibles para un TFG.

- **La evaluación en condiciones del mundo real con una variedad completa de dispositivos y escenarios de uso**, ya que probar la solución en todas las posibles combinaciones de dispositivos y tecnologías de comunicación del mundo real requeriría un esfuerzo logístico considerable. Por ello, la evaluación se realizará en un entorno de prueba controlado para asegurar resultados replicables y consistentes.

- **Consideraciones específicas de implementación para dispositivos de hardware o sistemas operativos particulares**, dado que adaptar la solución para una amplia gama de plataformas requeriría un nivel de especialización y personalización que no es viable dentro de los límites del trabajo.

2. Antecedentes.

En este apartado se presentarán los antecedentes clave que sientan las bases para la correcta comprensión del tema central de este TFG. La importancia del Internet de las Cosas (IoT), su crecimiento exponencial y las complejidades de garantizar la seguridad y el control de acceso en este entorno son temas que, para abordar correctamente el proyecto, requieren de una revisión detallada.

El análisis de los antecedentes que se van a tratar ahora es fundamental para contextualizar los objetivos y las metas de este TFG, y para proporcionar una base sólida desde la cual se pueda entender y justificar la implementación y evaluación del EAP-based Authentication Service for CoAP en el ámbito del IoT.

2.1.- QUÉ ES EL INTERNET DE LAS COSAS (IOT)

El Internet de las Cosas (IoT) se ha convertido en una realidad a partir de un concepto acuñado por Kevin Ashton en la década de 1990. El concepto se refería a un método para mejorar el proceso de obtener información del mundo real utilizando máquinas en lugar de personas. Esto se basa en la suposición de que las máquinas son más eficientes y precisas que los humanos en ciertas actividades. [4]

El concepto de IoT hace referencia a la interconexión de los objetos cotidianos a través de Internet, permitiéndoles recopilar y compartir datos de forma automática y colaborativa. Gracias a IoT, estos dispositivos inteligentes pueden comunicarse entre sí y con otros equipos conectados, como pueden ser smartphones o puertas de enlace. Esto da lugar a una extensa red de dispositivos interconectados capaces de intercambiar información y llevar a cabo diversas tareas de forma autónoma. Esta revolucionaria tecnología tiene el potencial de transformar la manera en que interactuamos con el mundo que nos rodea, brindando una mayor eficiencia, comodidad y seguridad en diversos aspectos de nuestra vida diaria. [5]

Los dispositivos IoT son elementos físicos que están equipados con sensores, actuadores y tecnología de conectividad, lo que les permite recopilar datos del entorno, procesar información y tomar acciones en función de los datos recopilados. Estos dispositivos pueden ser tan simples como sensores de temperatura o tan complejos como sistemas de automatización industrial o dispositivos médicos inteligentes.

2.1.1- Tecnologías IoT

El Internet de las Cosas se basa en una variedad de tecnologías que permiten la conectividad y la comunicación entre dispositivos inteligentes. Como se puede observar en la figura 2.1 [6], algunas de las tecnologías más comunes utilizadas en el ámbito del IoT clasificadas en función del alcance son las siguientes:

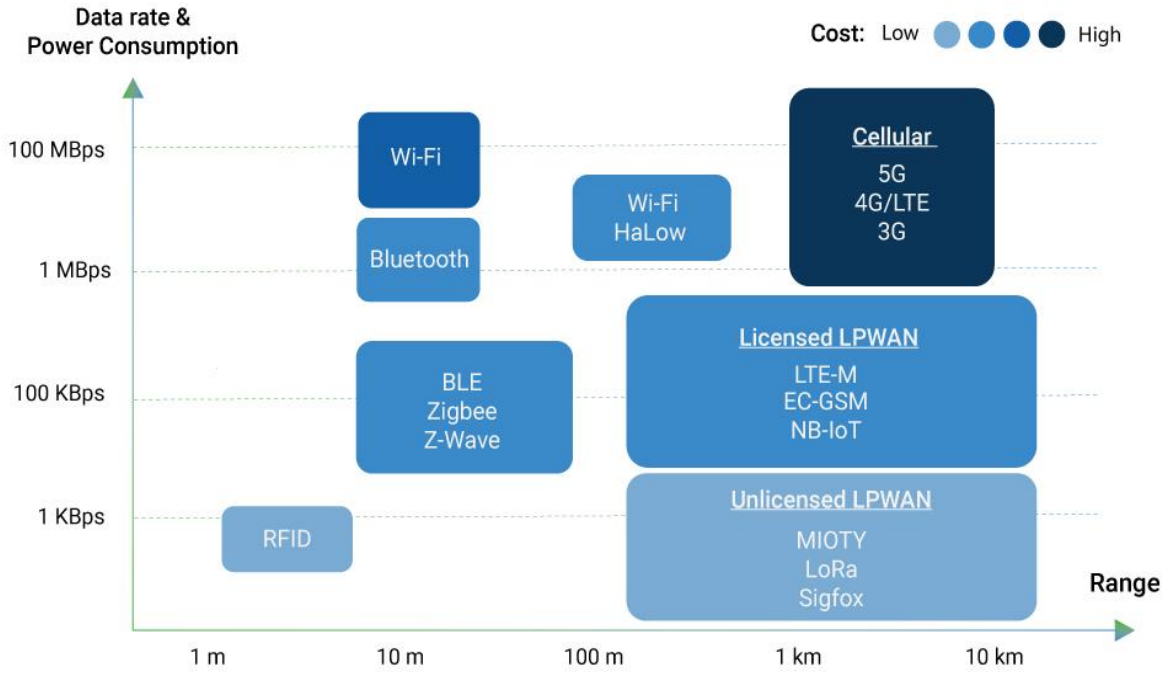


Figura 2.1.- Clasificación de las tecnologías IoT en función del alcance y de la tasa de datos transmitidos

MUY CORTO ALCANCE

- **NFC (Near Field Communication):** NFC es un protocolo de comunicación entre dispositivos que destaca por su capacidad de interactuar con chips carecientes de alimentación, como los utilizados en las tarjetas bancarias, y por operar a distancias

muy cortas, de aproximadamente 20 cm. Actualmente, su uso principal está en el ámbito bancario, ya que sus características lo hacen perfecto para realizar transacciones, lo que explica su presencia en tarjetas de crédito y otros medios similares. Además, NFC permite la transferencia inalámbrica de cualquier tipo de información, funcionando de manera similar a un código QR. [7] Su topología es punto a punto y soporta seguridad basada en certificados ECC.

- **RFID (Radio Frequency Identification):** Esta tecnología emplea ondas de radio para facilitar la identificación y el seguimiento de objetos, personas o animales. RFID no requiere contacto visual para su lectura, a diferencia de los códigos de barras, lo que la convierte en una herramienta altamente eficiente y versátil. Los chips RFID pueden almacenar una cantidad considerable de datos, lo que los hace especialmente valiosos en el ámbito logístico para gestionar cargas, paquetes e inventarios. Asimismo, la tecnología integrada en estos chips dificulta significativamente su falsificación, aumentando la seguridad del sistema. [8] Los pasivos tienen un alcance desde menos de 20 cm, es el caso de LF (RFID de Baja Frecuencia), hasta 5 metros, es el caso de UHF (RFID de Ultra Alta Frecuencia). Sin embargo, los activos pueden tener un alcance superior a los 100 metros. Su topología también es punto a punto y soporta seguridad basada en certificados ECC al igual que NFC.

CORTO ALCANCE

- **BLUETOOTH:** Bluetooth es una tecnología de comunicación inalámbrica de corto alcance, con un rango máximo de hasta 100 metros, que utiliza una topología en estrella o de red mallada. Esta tecnología permite la transmisión de datos y voz entre dispositivos como teléfonos móviles, ordenadores y periféricos sin necesidad de cables. Su objetivo principal es sustituir las conexiones físicas tradicionales, garantizando al mismo tiempo la seguridad en las comunicaciones entre los dispositivos conectados. [9] Un ejemplo común del uso de esta tecnología es la reproducción de música en auriculares inalámbricos, altavoces portátiles o sistemas de audio en coches, donde también se utiliza para realizar y recibir llamadas. Además, Bluetooth permite la transmisión de información entre dispositivos, como ordenadores, smartphones y otros equipos similares, facilitando el intercambio de

mensajes, archivos o datos de sensores, como ocurre con los relojes inteligentes. Esta tecnología también está presente en consolas de videojuegos y en los mandos utilizados para experiencias de realidad virtual. [10]

- **WIFI:** Esta es una de las tecnologías de conectividad más familiares y ampliamente utilizadas en el hogar y en entornos comerciales. Wi-Fi permite una conexión inalámbrica de alta velocidad y es ideal para dispositivos IoT que requieren una comunicación rápida y de alta capacidad. [11]

- **IEE 802.15.4:** El estándar IEEE 802.15.4 opera en las bandas ISM y establece las especificaciones para el nivel físico y el control de acceso al medio (MAC) en redes inalámbricas de área personal con baja tasa de transmisión de datos. Estas redes, conocidas como LR-WPAN (Low-Rate Wireless Personal Area Network), suelen emplear una topología de red mallada [12]. IEEE 802.15.4 ofrece un alcance de entre 10 y 75 metros y es utilizado por diversas tecnologías, algunas de las cuales son estas:

    - **ZigBee:** ZigBee se trata de un protocolo de comunicación inalámbrico diseñado para aplicaciones con una baja tasa de transferencia de datos y de bajo consumo energético. Es ampliamente utilizado en el ámbito del IoT para conectar dispositivos inteligentes como luces, sensores y termostatos. Las redes ZigBee destacan por su alta fiabilidad incluso en entornos con interferencias, tienen un alcance limitado de hasta 100 metros y emplean una topología de red Mesh. Esto permite que los dispositivos se comuniquen entre sí siempre que estén dentro del rango, incluso si no están conectados directamente al coordinador de la red. Una de las ventajas clave de ZigBee es su capacidad para gestionar la red automáticamente, creando y actualizando rutas dinámicamente. Esto garantiza que la red pueda resolver de forma automática problemas como la desconexión de algún nodo, evitando la necesidad de intervenir físicamente para resolver dichas incidencias.

      Debido a su bajo coste y tamaño reducido, este protocolo es especialmente popular en aplicaciones domóticas, donde se integra fácilmente en dispositivos del hogar, como electrodomésticos, alarmas, bombillas y otros equipos típicos de una casa inteligente. [13], [14]

- **ISA100.11a:** Es una red de malla que proporciona comunicación inalámbrica segura para el control de procesos. [12]

- **Wireless Hart:** Se trata de una tecnología de red de sensores inalámbricos basada en el protocolo HART (Highway Addressable Remote Transducer). Fue diseñado como un estándar inalámbrico interoperable, compatible con múltiples proveedores, para satisfacer las necesidades de las redes de dispositivos de campo en procesos industriales. Este protocolo emplea una arquitectura de malla, sincronizada temporalmente, autoorganizada y autorreparable, lo que le permite adaptarse y mantener su funcionalidad de manera eficiente. [15]

- **6LoWPAN:** El sistema 6LoWPAN se utiliza para una variedad de aplicaciones, incluidas las redes de sensores inalámbricos. Esta forma de red de sensores inalámbricos envía datos como paquetes y utiliza IPv6, que proporciona la base para el nombre, IPv6 sobre redes de área personal inalámbricas de baja potencia.

  Surgió inicialmente para superar las metodologías convencionales que se adaptaron para transmitir información. Pero, aun así, no es tan eficiente ya que solo permite que los dispositivos más pequeños con una capacidad de procesamiento muy limitada establezcan comunicación utilizando uno de los Protocolos de Internet, es decir, IPv6. Tiene un costo muy bajo, corto alcance, bajo uso de memoria y baja tasa de bits.

  Comprende un enrutador perimetral y Nodes sensores. Incluso el más pequeño de los dispositivos IoT ahora puede ser parte de la red, y la información también se puede transmitir al mundo exterior. [16]

Para la seguridad de los datos, el estándar IEEE 802.15.4 emplea el Estándar de cifrado avanzado (AES) con una longitud de clave de 128 bits como técnica de cifrado básica.

LARGO ALCANCE

- **LPWAN:** Las redes LPWAN (Low Power Wide Area Networks) son tecnologías inalámbricas de comunicación que permiten la transmisión de datos entre un dispositivo y una estación base o gateway, separados por distancias que pueden abarcar desde cientos de metros hasta varios kilómetros (generalmente superiores a 10 Km), con un consumo energético muy bajo.

  Gracias a sus características, las tecnologías LPWAN se han posicionado como una solución clave en el desarrollo de los grandes proyectos actuales en el ámbito del IoT. Este tipo de redes permiten desplegar decenas o incluso cientos de nodos distribuidos en amplias áreas geográficas, operando con baterías de larga duración que pueden mantenerse activas durante años, sin necesidad de infraestructuras complejas ni de costosos tendidos de cable. Sin embargo, para lograr un gran alcance y un consumo energético extremadamente bajo, estas tecnologías limitan la cantidad de datos que pueden transmitirse, es decir, solo permiten enviar pequeñas cantidades de información (bytes) por unidad de tiempo [17]. Algunas tecnologías LPWAN son estas:

  - **NB-IoT:** NB-IoT es una tecnología LPWAN basada en LTE que opera en un espectro bajo licencia. Está diseñada específicamente para dispositivos fijos que requieren un bajo consumo de energía y que manejan volúmenes reducidos de transferencia de datos. Es la solución que las compañías de telecomunicaciones (Telecom) ofrecen como parte de su propuesta para el ecosistema IoT. Esta tecnología utiliza la infraestructura existente de las redes celulares para proporcionar una cobertura amplia y una larga vida útil de la batería para dispositivos IoT. [18]

  - **Sigfox:** Sigfox es otra tecnología LPWAN con espectro sin licencia que proporciona conectividad de larga distancia y bajo consumo de energía para dispositivos IoT, pensada para no depender de los despliegues de telefonía. Proporciona una red que aporta efectividad para la transmisión de bajos volúmenes de datos, que garantiza una buena calidad de servicio y que es capaz de soportar simultáneamente un gran número de dispositivos, como en

un despliegue masivo de sensores. Los dispositivos que utilizan Sigfox permanecen en estado de espera la mayor parte del tiempo, lo que reduce significativamente su consumo energético. Esta característica representa su mayor ventaja, ya que el bajo consumo hace innecesaria la conexión a la red eléctrica o el uso de baterías de gran tamaño. Sigfox es la primera tecnología LPWAN de gran envergadura desplegada en España y, hasta la fecha, es la que se ha empleado en más proyectos gracias a su a su facilidad de suscripción, a su amplia cobertura y a su accesibilidad. Entre los proyectos más destacados se encuentran la gestión de sistemas de calefacción y ventilación en edificios, el control de vallas publicitarias, la administración de alarmas y la trazabilidad de activos, entre otros. [19]

- **LoRaWAN:** Se trata de una tecnología LPWAN con espectro sin licencia, diseñada para conectar dispositivos IoT a largas distancias y en áreas remotas con bajo consumo de energía. A diferencia de las otras tecnologías LPWAN, permite desplegar redes propias autogestionadas, eliminando muchas de las restricciones habituales impuestas por operadores privados. Estas cualidades lo convierten en una solución ideal para reducir costos, optimizar procesos y mejorar la eficiencia en diversos sectores. Por ejemplo, es ampliamente utilizado para administrar residuos, monitorizar activos en fábricas, gestionar la cadena de frío en la industria alimentaria, manejar la agricultura inteligente, para el control de acceso y la seguridad en entornos industriales, así como para el mantenimiento predictivo de equipos. [20]

2.1.2.- Ámbitos del uso de IoT

Los dispositivos IoT han encontrado aplicaciones en diversas industrias, mejorando la calidad de los servicios y la eficiencia [21]. En el ámbito de la **atención médica**, por ejemplo, se utilizan para la supervisión remota de los pacientes, pudiendo recopilar datos en tiempo real sobre sus signos vitales, tales como son la saturación de oxígeno, la frecuencia cardíaca y la presión arterial. Esta capacidad permite un análisis constante de la salud del paciente, facilitando la detección temprana de patrones que podrían indicar problemas serios. Además, los dispositivos IoT son útiles para gestionar inventarios, rastrear equipos médicos

y asegurar que se cumplan los tratamientos, optimizando así los recursos en el sector de la salud.

En el **sector de la fabricación**, los dispositivos IoT desempeñan un papel vital al permitir la supervisión del rendimiento de las máquinas y la detección de fallos en los equipos. Gracias a los sensores, es posible controlar variables como la humedad y la temperatura en las plantas de producción, asegurando las condiciones ideales para la fabricación de productos sensibles. Además, estos dispositivos ayudan a garantizar la calidad de los productos finales, a gestionar el inventario y a optimizar las cadenas de suministro, contribuyendo a una producción más eficiente y rentable.

El **sector minorista** se beneficia enormemente de los dispositivos IoT que permiten controlar el inventario en sus diferentes niveles y realizar un seguimiento del comportamiento de los clientes. Los sensores, por ejemplo, pueden analizar el flujo de personas en las tiendas, proporcionando información valiosa para mejorar la disposición de los productos y hacer que la experiencia del cliente sea óptima. Estos dispositivos también facilitan la gestión de envíos y la supervisión de las cadenas de suministro, ayudando a los minoristas a mantener un control más efectivo de sus operaciones.

En el ámbito de la **agricultura**, los dispositivos IoT se emplean para monitorear el crecimiento de los cultivos, las condiciones del suelo y los patrones climáticos. Sensores especializados son capaces de medir la humedad del suelo, garantizando que los cultivos reciban el riego adecuado en el momento preciso. Por otra parte, estos dispositivos permiten gestionar las cadenas de suministro, comprobar la salud del ganado y rastrear la ubicación de la maquinaria. Los dispositivos de baja potencia, incluidos los que funcionan con energía solar, son ideales para su uso en ubicaciones remotas donde la supervisión es mínima.

En el sector del **transporte**, los dispositivos IoT son esenciales para llevar a cabo un seguimiento de los envíos, examinar el rendimiento de los vehículos y optimizar rutas. Los sensores que miden la eficiencia del combustible en vehículos conectados no solo reducen costos operativos, sino que también contribuyen a una mayor sostenibilidad. Asimismo, los dispositivos IoT permiten supervisar el estado de la carga, garantizando que los productos sean transportados y entregados en las mejores condiciones.

Los **hogares inteligentes** representan otra área en la que los dispositivos IoT han hecho una gran diferencia. Estos sistemas están interconectados con electrodomésticos para automatizar tareas específicas y suelen ser controlados de manera remota. Ejemplos de dispositivos IoT en este contexto incluyen electrodomésticos de cocina inalámbricos, sistemas de música adaptativos, iluminación inteligente y dispositivos de control del consumo energético.

Finalmente, las **ciudades inteligentes** están adoptando dispositivos IoT, como medidores y sensores conectados, para reunir y estudiar datos que permiten mejorar tanto los servicios públicos como la infraestructura urbana. Esta recopilación de datos no solo optimiza el uso de recursos, sino que también incrementa la calidad de vida de los ciudadanos al permitir una gestión más eficiente de los servicios.

En conjunto, estas aplicaciones demuestran cómo el Internet de las Cosas está transformando múltiples sectores, brindando soluciones innovadoras y mejorando la calidad de vida en diversas áreas.

2.2.- CONTROL DE ACCESO A LA RED EN EL CONTEXTO DE IOT

El Internet de las Cosas (IoT) ha brotado como una tecnología disruptiva con la capacidad de transformar una gran variedad de industrias y aspectos de la vida diaria. Desde dispositivos domésticos inteligentes hasta infraestructuras industriales complejas, el IoT conecta objetos físicos mediante de redes de comunicación, posibilitando recopilar e intercambiar datos de una manera sin precedentes.

El despliegue de dispositivos IoT está alcanzando cifras récord a nivel global. Según estimaciones del IDC (International Data Corporation), para el año 2025 se espera que existan cerca de 75 mil millones de dispositivos conectados, generando aproximadamente 79,4 zettabytes (ZB) de datos, como se ilustra en la figura 2.2 [22, Fig. 1]. Este crecimiento exponencial destaca la creciente importancia de establecer mecanismos eficaces de control de acceso en el ecosistema IoT.

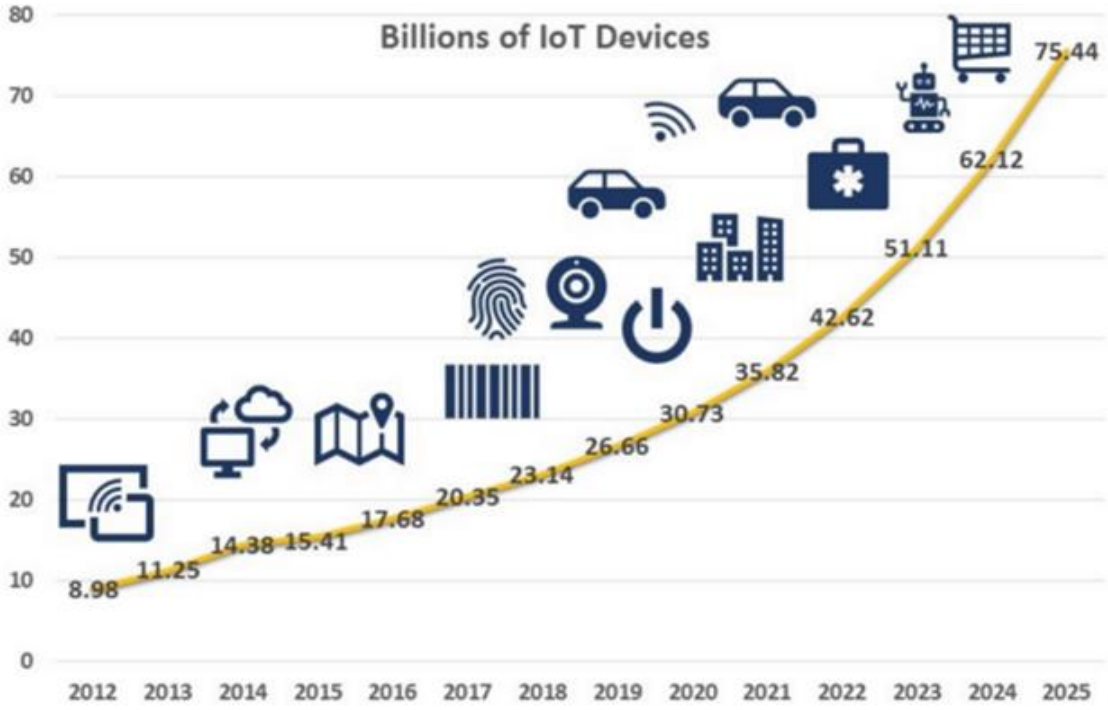


Figura 2.2.- Evolución temporal del número de dispositivos IoT

Dado que un gran número de estos dispositivos están involucrados en la gestión de infraestructuras críticas o en el manejo de información sensible, tanto empresarial como personal, garantizar una autenticación sólida y un control de acceso efectivo se ha convertido en una prioridad esencial en el ámbito del IoT. [2]

La autenticación de dispositivos IoT es fundamental para asegurar que los dispositivos conectados sean legítimos y confiables. De este modo, el control de acceso puede regular qué recursos son accesibles, utilizables y bajo qué circunstancias, reduciendo al mínimo el riesgo de actividades no autorizadas. [23]

Sin embargo, esta conectividad también plantea desafíos significativos en términos de seguridad y control de acceso a la red. En este documento, se va a explorar detalladamente la importancia del control de acceso en el contexto del IoT, describiendo los desafíos específicos que enfrentan los sistemas de autenticación y por qué abordar estos desafíos es crucial para la integridad y la seguridad de los sistemas IoT.

2.2.1.- Importancia del control de acceso

El control de acceso en el IoT es de vital importancia debido a la naturaleza masiva y diversa de los dispositivos conectados en este entorno. Cada dispositivo, desde sensores hasta equipos de cómputo avanzados, interactúa constantemente con otros dispositivos y con sistemas de backend más amplios mediante redes de comunicación, tales como Zigbee, Bluetooth, Wi-Fi y redes celulares, que pueden variar en términos de protocolos y capacidades. Esto significa que, sin un control de acceso adecuado, cualquier dispositivo podría potencialmente acceder a la red, lo que representa un riesgo mayúsculo para la seguridad.

Además, el crecimiento exponencial del número de dispositivos conectados aumenta la superficie de ataque, ya que cada dispositivo no autorizado representa una potencial puerta de entrada a la red para actores maliciosos. Sin un sistema robusto de control de acceso, los dispositivos no confiables podrían interceptar datos sensibles, interferir con las operaciones normales o incluso comprometer la infraestructura crítica de una organización, de modo que la necesidad de asegurar que solo puedan acceder y comunicarse con los recursos de red los dispositivos autorizados se vuelve crítica. [24]

Es especialmente relevante en escenarios industriales o de salud, donde los sistemas de IoT manejan información altamente confidencial, como datos médicos o procesos de producción. La falta de control de acceso no solo pone en peligro la privacidad de los usuarios, sino también la integridad de los servicios que dependen de estos dispositivos. De igual manera, en un contexto doméstico o de ciudades inteligentes, los dispositivos no autorizados podrían acceder a sistemas de seguridad, monitoreo energético, o infraestructura pública, comprometiendo tanto la seguridad personal como la pública. [25]

El control de acceso eficiente en IoT no solo implica verificar la identidad de los dispositivos, sino también gestionar los privilegios que estos tienen dentro de la red. Un control adecuado garantiza que los dispositivos puedan interactuar solo con los recursos específicos para los que están autorizados, minimizando así el impacto de posibles vulnerabilidades. [26]

2.2.2.- Desafíos específicos en sistemas de autenticación IoT

La naturaleza distribuida y heterogénea del ecosistema IoT presenta desafíos únicos en términos de control de acceso a la red.

La escalabilidad representa uno de los principales desafíos en los sistemas de autenticación IoT, ya que se requiere la capacidad de escalar para admitir un gran número de dispositivos. En este sentido, los sistemas tradicionales de autenticación pueden no ser adecuados para entornos IoT masivamente distribuidos, donde miles o incluso millones de dispositivos pueden estar conectados simultáneamente [27]. Por lo tanto, la gestión eficiente de las credenciales y el procesamiento de las solicitudes de autenticación son aspectos críticos que deben abordarse para garantizar la escalabilidad de los sistemas de autenticación IoT.

Además, la heterogeneidad de dispositivos y tecnologías en el IoT presenta un desafío adicional para los sistemas de autenticación. En este contexto, los dispositivos pueden variar en términos de capacidad de procesamiento, capacidad de almacenamiento, protocolos de comunicación y requisitos de seguridad. Esto implica que se requieren enfoques de autenticación flexibles que puedan adaptarse a diferentes contextos y requisitos de dispositivos [28]. A su vez, la interoperabilidad entre diferentes plataformas y protocolos también resulta fundamental para garantizar la compatibilidad entre dispositivos y sistemas de autenticación.

Por otro lado, las limitaciones de recursos son un factor a considerar, dado que muchos dispositivos IoT tienen recursos limitados en términos de energía, capacidad de procesamiento y memoria. Así, los métodos de autenticación tradicionales que requieren un alto consumo de recursos pueden no ser prácticos para estos dispositivos. En consecuencia, es necesario desarrollar métodos de autenticación ligeros y eficientes que puedan funcionar en entornos con recursos limitados, lo que puede implicar el uso de técnicas de criptografía de bajo costo computacional y protocolos de autenticación optimizados para dispositivos de baja potencia. [26], [27]

Finalmente, la seguridad emerge como una preocupación central en los sistemas de autenticación IoT. Los dispositivos IoT pueden ser vulnerables a una variedad de ataques, incluyendo ataques de suplantación de identidad, ataques de denegación de servicio y ataques de retransmisión [29]. Por lo tanto, los sistemas de autenticación deben ser capaces de mitigar estos riesgos y garantizar la integridad y la confidencialidad de las comunicaciones en el IoT. Esto puede implicar el uso de técnicas de autenticación multifactoriales, protocolos de cifrado robustos y mecanismos de detección y respuesta a amenazas en tiempo real.

2.2.3.- Por qué es crucial abordar estos desafíos

Abordar los desafíos en los sistemas de autenticación IoT es crucial por varias razones fundamentales. En primer lugar, la seguridad es esencial para proteger la privacidad de los usuarios, prevenir el acceso no autorizado a los datos y evitar interrupciones en los servicios críticos. Dado que los dispositivos IoT pueden ser utilizados para recopilar datos sensibles, controlar sistemas críticos e interactuar con el entorno físico, se convierten en objetivos atractivos para los ciberdelincuentes.

Además, la integridad de los datos generados por los dispositivos IoT es de suma importancia, ya que pueden ser utilizados para tomar decisiones críticas en áreas como la salud, la seguridad y la producción industrial. Por lo tanto, garantizar la integridad de estos datos es fundamental para la fiabilidad y precisión de los sistemas IoT. La manipulación o corrupción de datos puede tener consecuencias graves, como errores en diagnósticos médicos, fallos en sistemas de control industrial y decisiones erróneas basadas en datos incorrectos.

La confianza del usuario también juega un papel esencial, ya que depende en gran parte de la seguridad y la privacidad de los datos. Los usuarios necesitan la garantía de que sus dispositivos están protegidos contra amenazas de seguridad y que sus datos personales están siendo manejados de manera segura y responsable. La falta de seguridad puede erosionar esta confianza y disuadir la adopción de tecnologías IoT, limitando su potencial de innovación y crecimiento.

Por último, el éxito continuo del IoT depende de su adopción generalizada por parte de consumidores, empresas y organizaciones. Sin embargo, la falta de seguridad puede obstaculizar esta adopción, ya que genera preocupaciones sobre la privacidad y la seguridad de los datos. Las empresas, en particular, pueden ser reacias a implementar soluciones IoT si perciben riesgos significativos para la seguridad de sus datos y operaciones. Por lo tanto, abordar los desafíos en los sistemas de autenticación IoT es fundamental para impulsar tanto la adopción generalizada como el crecimiento sostenible del IoT en todo el mundo.

2.3.- MÉTODOS DE AUTENTICACIÓN EN EL IOT

Los métodos de autenticación juegan un papel esencial en la seguridad del Internet de las Cosas, como se mencionó previamente, ya que garantizan que únicamente los usuarios y los dispositivos autorizados puedan acceder a datos y recursos sensibles.

Cada método de autenticación en el IoT implica diferentes demandas de recursos, lo que requiere un análisis cuidadoso al elegir e implementar mecanismos de seguridad. Aspectos como la capacidad para realizar operaciones criptográficas, junto con la gestión de claves y certificados, afectan a la hora de determinar los recursos requeridos por los dispositivos IoT. Por ello, para seleccionar el método de autenticación que mejor equilibre seguridad, rendimiento y eficiencia energética, es crucial evaluar el almacenamiento seguro, el consumo energético y la capacidad de cómputo de los dispositivos.

Para asegurar que la implementación de la autenticación en entornos IoT se ajuste a las limitaciones y a las capacidades de los dispositivos conectados, es crucial tener en cuenta estos requisitos de recursos, garantizando así que ni la seguridad ni la integridad de la red se vean comprometidas.

Sin embargo, la mayoría de las técnicas de autenticación y autorización tradicionales son excesivamente complicadas para ser ejecutadas en dispositivos IoT con recursos limitados, puesto que en los protocolos de autenticación convencionales existe una sobrecarga de comunicación. Otro inconveniente es que, en ocasiones, el despliegue de los

dispositivos se lleva a cabo en ubicaciones donde garantizar la seguridad física resulta inviable o poco práctico.

Además, existen una gran cantidad de dispositivos que se comunican a través de diversos estándares y protocolos, lo que los diferencia de los entornos informáticos tradicionales, debido a la extensa gama de pilas hardware y software usadas.

Por otro lado, la ausencia de normativas y modelos específicos de control de acceso para el IoT aumenta la complejidad de garantizar la seguridad de estos dispositivos y sus redes.

En esta revisión de literatura, se examinan algunos de los enfoques existentes de autenticación en el IoT, destacando sus fortalezas y limitaciones.

2.3.1.- Métodos tradicionales

- **Contraseñas:** Las contraseñas son uno de los métodos de autenticación más comunes en el IoT. Para acceder a un sistema los usuarios o dispositivos deben introducir una combinación única de nombre de usuario y contraseña [23]. Las contraseñas son fáciles de implementar y comprender, pero son susceptibles a ataques de fuerza bruta, suplantación de identidad y robos de credenciales.

- **Certificados Digitales:** Los certificados digitales utilizan criptografía de clave pública para autenticar dispositivos en el IoT. Cada dispositivo tiene un certificado único que se utiliza para verificar su identidad. Los certificados digitales ofrecen un alto nivel de seguridad, pero pueden ser costosos de integrar y gestionar, especialmente en entornos con grandes cantidades de dispositivos. [23]

- **Token de Acceso:** Los tokens de acceso son cadenas de caracteres generadas dinámicamente que se utilizan para autenticar dispositivos en el IoT. Los tokens de acceso son temporales y pueden ser revocados si se detecta actividad sospechosa. Sin embargo, la gestión de tokens puede ser compleja y requiere una infraestructura de clave pública sólida para garantizar su seguridad. [30]

2.3.2.- Métodos avanzados

- **Biometría:** Para verificar la identidad de un usuario, la autenticación biométrica utiliza características físicas o comportamentales únicas, tales como el iris, las huellas dactilares, o el reconocimiento facial. Esta tecnología proporciona un nivel elevado de comodidad y seguridad, su implementación puede resultar costosa y suscitar inquietudes relacionadas con la privacidad y la protección de los datos personales. [23]

- **Autenticación Multifactorial:** La autenticación multifactorial combina varios métodos de autenticación, como contraseñas, tokens de acceso y biometría, para proporcionar un nivel adicional de seguridad. Los usuarios o dispositivos deben pasar múltiples etapas de autenticación para acceder a un sistema. La autenticación multifactorial es altamente segura, pero puede ser compleja y engorrosa para los usuarios finales. [31]

- **Protocolos de Autenticación Específicos de IoT:** En respuesta a las necesidades únicas de los dispositivos IoT, se han desarrollado varios protocolos de autenticación específicos basados en EAP conocidos como métodos EAP. Algunos ejemplos de métodos EAP son EAP-PSK, EAP-AKA o EAP-TLS entre otros.

  Estos métodos están diseñados para ser ligeros, eficientes y seguros en entornos con recursos limitados. Sin embargo, pueden presentar limitaciones en términos de interoperabilidad, ya que al estar diseñados para casos de uso concretos pueden no ser compatibles con dispositivos, sistemas o infraestructuras que usen protocolos de autenticación más generales. Esto significa que no siempre pueden integrarse fácilmente en redes heterogéneas donde coexistan dispositivos IoT y no IoT. Además, muchos de estos métodos aún son relativamente nuevos (como EAP-NOOB o EAP-EDHOC) y no han alcanzado el nivel de adopción global que tienen otros estándares más establecidos, como WPA2 o TLS en redes Wi-Fi o en conexiones seguras por internet. Esto puede limitar su uso práctico y su soporte por parte de fabricantes y proveedores de tecnología.

2.4.- EVOLUCIÓN TEMPORAL DE LA SEGURIDAD EN EL IOT

A medida que los despliegues de IoT se multiplicaron, los problemas de interoperabilidad, privacidad y, sobre todo, de seguridad, ya que la proliferación de dispositivos conectados incrementa la superficie de ataque, emergieron como áreas críticas. Los despliegues de IoT implican la instalación de miles de dispositivos idénticos, lo que expone a las redes a efectos de "dominó" en caso de vulnerabilidades. Además, la larga vida útil de estos dispositivos supone que las medidas de seguridad pueden quedar obsoletas con el tiempo y la dificultad de actualización agrega otra capa de complejidad. A esto se añade la falta de visibilidad del funcionamiento interno de los dispositivos IoT, lo que dificulta la detección de brechas de seguridad [32, pp. 37-38]. Estos desafíos no son meramente teóricos, sino que ya han sido explotados por atacantes que aprovechan las vulnerabilidades de los dispositivos IoT mal asegurados para realizar ataques de denegación de servicio, robos de datos, y hasta sabotajes a infraestructura crítica.

Otro aspecto relevante es que IoT, en muchos casos, se ha asociado con dispositivos y redes con capacidades limitadas, como limitaciones de batería, baja potencia y velocidad de transmisión reducida. Esto implica una necesidad de soportar protocolos eficientes en cuanto a requerimiento de computación y sobre todo con un intercambio de mensajes reducido, tanto en número de mensajes como en número de bytes intercambiados, para minimizar el consumo de recursos. También hace difícil implementar los mecanismos de seguridad robustos que normalmente se emplean en otras áreas de la informática. [32, p. 53]

La homogeneización de las comunicaciones y la adopción de estándares abiertos ha sido una de las soluciones clave para hacer frente e intentar mitigar estos retos [32, p.39].
Los protocolos de seguridad en IoT están evolucionando hacia el uso de codificaciones eficientes como CBOR [33] (Concise Binary Object Representation), que permite la creación y el procesamiento más ligero de firmas digitales y de códigos de autenticación de mensajes (MAC por sus siglas en inglés), usando el estándar COSE [34] (CBOR Object Signing and Encryption). Este enfoque, junto con el impulso hacia el uso de suites criptográficas que soportan AEAD (Authenticated Encryption with Associated Data), está diseñado para mejorar la eficiencia de los protocolos de seguridad, lo que es vital en entornos de IoT donde el consumo de recursos debe minimizarse. [32, p. 53]

Organizaciones como el IETF y el IEEE han liderado esfuerzos en el diseño de protocolos específicos para IoT basados en EAP (Métodos EAP), que han sido adaptado a dispositivos con capacidades limitadas y redes de baja potencia [32, p. 55]. La adopción de EAP junto con el uso de tecnologías como LPWAN y IEEE 802.15.4, ha permitido que los dispositivos IoT conserven un equilibrio entre eficiencia energética y seguridad robusta. Estos avances han sido esenciales para facilitar el despliegue seguro de dispositivos IoT en sectores críticos como la industria, la sanidad y las ciudades inteligentes.

Generalmente es necesario autenticar , autorizar y monitorizar el uso de los recursos de red de los dispositivos, así como gestionar el material criptográfico necesario para unas comunicaciones seguras. Es por ello por lo que el estado actual de la seguridad en IoT ha alcanzado un punto crítico en el que el ciclo de vida completo de los dispositivos IoT debe ser cubierto, desde su fase inicial de bootstraping (arranque), hasta la fase de operación (post-bootstraping) y el mantenimiento. La federación de identidad, similar al funcionamiento de Eduroam en el entorno académico, está emergiendo como una solución para facilitar el acceso seguro y eficiente en redes IoT, permitiendo que los usuarios y dispositivos se autentiquen en múltiples dominios sin necesidad de credenciales redundantes. [32, p. 56]

Durante las últimas décadas, la seguridad en el Internet de las Cosas (IoT) ha experimentado avances importantes en múltiples áreas. No obstante, dado que el presente trabajo versa sobre una EAP lower layer, a continuación se presenta una cronología de los hitos más relevantes en la evolución de EAP y de las tecnologías relacionadas, destacando su relevancia para la seguridad de los dispositivos IoT.

2.4.1.- Década de 1990

En 1997, se publicó el estándar IEEE 802.1X, que define un marco para la autenticación de puertos en redes cableadas. Este estándar desempeña un papel crucial al proporcionar un mecanismo robusto para la autenticación de dispositivos en redes Ethernet y Wi-Fi. A través del estándar IEEE 802.1X, los dispositivos conectados a un puerto LAN pueden ser autenticados, de tal forma que se establece una conexión punto a punto segura si la autenticación es exitosa, o se deniega el acceso a ese puerto en caso contrario. En esencia, IEEE 802.1X se trata de una norma independiente que actúa en la capa de enlace y verifica

a todos los usuarios presentes en las redes LAN o WLAN, otorgándoles o denegándoles el acceso según corresponda. Su función principal es prevenir accesos no autorizados desde el inicio, asegurando que solo los dispositivos autenticados puedan comunicarse a través de la red. [35, Sec. ¿Qué es 802.1X?]

En IEEE 802.1X, el proceso comienza cuando el solicitante envía sus credenciales al autenticador a través del protocolo EAP. Luego, el autenticador transmite estas credenciales al servidor de autenticación, que las verifica comparándolas con las de autorizaciones anteriores, registradas normalmente en una base de datos o en un archivo de texto. Si las credenciales son válidas, el servidor se lo notifica al autenticador, que permite al solicitante acceder a la red y, si es necesario, asignarle recursos para su uso como ancho de banda. Sin embargo, en caso de que las credenciales sean inválidas, el solicitante es rechazado denegándole el acceso a la red. [35, Sec. ¿Cómo funciona el protocolo IEEE 802.1X?]

Otro avance importante de esa época fue que, en 1998, EAP fue formalmente definido en el RFC 2284 [36], proporcionando un marco extensible para la autenticación en redes. Aunque inicialmente se diseñó para redes como PPP (Point-to-Point Protocol) y Wi-Fi, la arquitectura flexible de EAP le permitió posteriormente adaptarse a una amplia gama de tecnologías, incluyendo IoT, donde la diversidad de dispositivos y sus limitaciones en recursos representan un desafío único. A medida que los dispositivos conectados comenzaron a proliferar, la capacidad de EAP para soportar diversos métodos de autenticación se convirtió en una herramienta clave en el desarrollo de soluciones de seguridad escalables.

2.4.2.- Década de los 2000

Con el avance de las redes inalámbricas y móviles, se desarrollaron métodos específicos de EAP para mejorar la autenticación en entornos más exigentes. En 2004, se publicó el primer documento formal del IETF que describe el Protocolo para Transportar Autenticación para Acceso a la Red (PANA), aunque no fue hasta 2008 cuando se estandarizó con la publicación del RFC 5191 [37]. Este protocolo proporciona un marco para la autenticación de dispositivos en redes IP. Es importante destacar que PANA no define un nuevo método de autenticación, ni un protocolo de distribución de claves o de derivación de

las mismas [38]. En cambio, se diseñó para actuar como una capa inferior de EAP (EAP lower layer), lo que significa que su propósito es transportar los métodos de autenticación que se definen en el protocolo EAP. Esto resultó beneficioso para el uso en IoT, ya que permite la interoperabilidad con diferentes tecnologías de enlace y métodos de autenticación.

DTLS (Datagram Transport Layer Security) fue introducido en 2006 mediante el RFC 4347 [39]. Es un protocolo diseñado para proporcionar seguridad a los datagramas UDP. Permite la autenticación, integridad y confidencialidad de los datos en aplicaciones que requieren comunicaciones seguras sin la sobrecarga de conexión de TCP y es fundamental en entornos IoT donde los dispositivos pueden utilizar UDP para minimizar la latencia y el consumo de energía.

Además, en 2007, se desarrolló y se definió el protocolo EAP-PSK (Extensible Authentication Protocol - Pre-Shared Key) en el RFC 4764 [40]. Se trata de un método diseñado para proporcionar autenticación mutua y derivación de claves de sesión usando una clave precompartida. EAP-PSK proporciona confidencialidad, integridad y autenticidad a las comunicaciones. Es útil en redes inseguras y para dispositivos con recursos limitados, como los sensores de baja potencia en redes IoT, lo que le confiere una aplicación significativa en este entorno.

Posteriormente, en 2008, se publicó el RFC 5216 [41], que describe el protocolo EAP-TLS, diseñado para la autenticación basada en certificados digitales. EAP-TLS es un método EAP que utiliza el protocolo TLS (Transport Layer Security) para proporcionar autenticación segura en las capas superiores del modelo OSI. Este protocolo se utiliza ampliamente en entornos de IoT que requieren un alto nivel de seguridad, como en aplicaciones médicas, industriales y de infraestructura crítica. Gracias a la utilización de certificados digitales para la autenticación mutua entre dispositivos y servidores, EAP-TLS también garantiza la confidencialidad, integridad y autenticidad de las comunicaciones en el IoT.

El funcionamiento de EAP-TLS implica el intercambio de certificados digitales entre el servidor y el usuario mediante SSL/TLS. En este proceso, el servidor envía su certificado digital, que incluye una clave pública y una identificación del servidor. A su vez, el

dispositivo del usuario valida este certificado y, posteriormente, envía su propio certificado, que también consta de una clave pública y una identificación del usuario, al servidor. Si el servidor valida correctamente el certificado del usuario y se cumplen los requisitos, se lleva a cabo la autenticación, permitiendo así el acceso a la red. [42]

En 2009, se publicó el RFC 5448 [43], que describe el protocolo EAP-AKA', una evolución del protocolo EAP-AKA, optimizado para mejorar la autenticación en redes móviles. Aunque está orientado principalmente a redes celulares, EAP-AKA' se ha adaptado en el contexto de IoT, especialmente en dispositivos que dependen de la conectividad móvil, como los que utilizan NB-IoT o LTE-M, brindando un marco robusto que optimiza la seguridad y la eficiencia en la autenticación.

2.4.3.- Década de 2010

En 2014, se publicó el RFC 7296 [44] que describe la versión dos del protocolo Internet Key Exchange (EAP-IKEv2). IKE es un componente de IPsec utilizado para realizar autenticación mutua y establecer y mantener Asociaciones de Seguridad. EAP-IKEv2 se trata de un protocolo de intercambio de claves que se utiliza para establecer sesiones seguras y gestionar la autenticación en redes. Este método es especialmente útil para entornos IoT que requieren una seguridad robusta, así como para la conexión de dispositivos móviles en redes inalámbricas. Este enfoque permite a los dispositivos IoT establecer conexiones seguras y gestionar sesiones de comunicación sin comprometer la seguridad de la información. Además, es eficiente en términos de ancho de banda, lo que es crucial para dispositivos IoT con recursos limitados.

Durante ese mismo año, se lanzó el RFC 7258 [45], el cual destacó las amenazas a la privacidad en entornos de comunicaciones. Este documento puso de relieve la importancia de reforzar la seguridad en IoT, dado el gran volumen de datos personales que estos dispositivos manejan y las vulnerabilidades inherentes a los entornos conectados.

En 2014 también se publicó el RFC 7170 [46], que define EAP-TEAP (EAP Tunneled Transport Layer Security). Este protocolo fue diseñado para proporcionar un túnel seguro mediante TLS dentro del cual se pueden ejecutar otros métodos de autenticación, es

flexible y permite reutilizar credenciales. La capacidad de EAP-TEAP para manejar autenticaciones sucesivas, como la autenticación de la máquina antes de que un usuario inicie sesión, lo hace ideal para aplicaciones IoT en entornos empresariales o académicos. Al ofrecer una capa adicional de seguridad a través del túnel TLS, EAP-TEAP protege las credenciales y la información sensible de ataques de intermediarios y otras amenazas comunes en redes tanto inalámbricas como por cable. Esto permite una gestión más segura y eficiente de la autenticación en dispositivos IoT, asegurando que solo los dispositivos y usuarios autorizados puedan acceder a la red.

Por último, en 2019, se introdujo el protocolo OSCORE (Object Security for Constrained RESTful Environments) a través del RFC 8613 [47], el cual está diseñado para proporcionar seguridad en entornos RESTful restringidos, como los que se encuentran en el IoT. Ofrece un enfoque de seguridad a nivel de objeto, permitiendo la protección de mensajes en las comunicaciones entre dispositivos. Esto es especialmente útil en situaciones donde los dispositivos tienen recursos limitados y requieren un modelo de seguridad eficiente.

2.4.4.- Década de 2020

En 2020 se comenzaron a publicar los primeros drafts IETF sobre el protocolo EDHOC (Ephemeral Diffie-Hellman Over COSE), pero no fue hasta 2024 cuando se estandarizó definitivamente con la publicación del RFC 9528 [48]. Consiste en un protocolo de autenticación diseñado para establecer claves de manera eficiente en entornos restringidos, utilizando el esquema de intercambio de claves de Diffie-Hellman. Se integra con COSE (CBOR Object Signing and Encryption) y permite la autenticación mutua de dispositivos IoT, haciendo hincapié en la simplicidad y el bajo consumo de recursos.

En 2021, se publicó el RFC 9140 [49], que define EAP-NOOB (EAP Method for the Provisioning of Symmetric Keys). Este protocolo es especialmente útil para la autenticación de dispositivos IoT con recursos limitados, ya que permite la provisión de claves simétricas sin necesidad de que los dispositivos cuenten con credenciales preconfiguradas. Está diseñado específicamente para facilitar la incorporación de dispositivos IoT que carecen de interfaces de usuario tradicionales y de capacidades de configuración de red previas. Este protocolo permite que estos dispositivos se autentiquen de manera segura en la red utilizando

canales fuera de banda (OOB) para el intercambio inicial de claves entre el servidor y el dispositivo. EAP-NOOB es un avance importante para entornos IoT donde los dispositivos tienen capacidades limitadas de procesamiento y almacenamiento, pero requieren autenticación segura.

Otro hito que tuvo lugar en 2021 fue la publicación de la última versión del draft IETF del protocolo HIP DEX (HIP Diet EXchange) [50]. HIP DEX extiende el Host Identity Protocol (HIP) para mejorar la autenticación y gestión de identidades de dispositivos IoT. Es un protocolo que separa las identidades de los hosts (dispositivos) de sus direcciones IP, proporcionando una capa de seguridad adicional, es decir, en lugar de basar las comunicaciones en direcciones IP, utiliza identificadores criptográficos conocidos como Host Identifiers (HI) para establecer y mantener sesiones seguras entre dispositivos.

La llegada de TLS 1.3 trajo mejoras significativas en la seguridad, y en 2022, el RFC 9190 [51] actualizó el protocolo EAP-TLS para dar soporte a TLS 1.3, eliminando algoritmos criptográficos obsoletos y reforzando la protección de datos en redes IoT. Esta mejora es crucial para dispositivos IoT que requieren comunicaciones seguras en redes abiertas o compartidas.

Gracias a esto, ese mismo año, también se pudo desarrollar y publicar DTLS 1.3 en el RFC 9147 [52]. Esta nueva versión ofrece una serie de mejoras significativas sobre sus predecesores, como un mejor rendimiento a través de menos rondas de ida y vuelta durante el establecimiento de la conexión. También simplifica la negociación de algoritmos de cifrado y proporciona un marco más seguro para la autenticación y la confidencialidad de los datos.

En 2024 se publicó la última versión del Internet Draft "EAP-based Authentication Service for CoAP" [53], un trabajo que lleva más de 10 años en marcha y que, a pesar de su relevancia, aún no ha alcanzado el estatus de RFC. Este Draft propone una solución para integrar EAP en el protocolo CoAP (Constrained Application Protocol), utilizado en IoT y diseñado para abordar los desafíos de autenticación en este entorno. Se enfoca en garantizar una autenticación eficiente en dispositivos con recursos limitados, utilizando métodos EAP que permiten una mayor flexibilidad y adaptabilidad en redes restringidas. Además, este

proyecto también incluye varias ventajas, como una EAP lower layer altamente eficiente a nivel de aplicación, el soporte de una amplia gama de métodos de autenticación EAP y de gran variedad de tipos de credenciales según los métodos de autenticación, la interacción en redes 5G y en redes IoT limitadas como IEEE 802.15.4 o LPWAN y la posibilidad de federación de identidad [32, p. 56]. El propósito principal de la federación de identidad es facilitar el acceso seguro de los usuarios de un dominio a los datos o sistemas de otro dominio, de manera sencilla y sin necesidad de redundancia en la administración de los usuarios. Para alcanzar este objetivo, es crucial que todos los sistemas involucrados adopten un protocolo común, garantizando así una interoperabilidad óptima. Este enfoque mejora la eficiencia y la interoperabilidad en redes IoT masivas, donde es necesario minimizar la complejidad de gestión de credenciales y autenticación. [54]

3. Planificación.

Una vez explicados los objetivos y los antecedentes, se va a proceder a detallar la planificación organizativa y temporal, utilizando para ello la herramienta Microsoft Project.

3.1.- ESTRUCTURA DEL TRABAJO

El proyecto se inicia con la presentación de la idea por parte de Dan García Carrillo, profesor del área de Ingeniería Telemática, seguido de un proceso de comprensión y análisis del mismo para definir el plan de ejecución, el cual está estructurado en cinco fases:

- **Fase 1. Análisis del contexto y revisión del estado del arte:** Esta fase tiene como objetivo comprender el contexto en el que se desarrolla el proyecto. Aquí se incluyen los siguientes pasos:

  - Repaso de los conceptos básicos de seguridad y de los conceptos relativos a los dispositivos IoT.
  - Revisión exhaustiva de la literatura existente sobre métodos de autenticación en IoT, con foco en la variedad de dispositivos, limitaciones y requisitos de seguridad.
  - Revisión de las tecnologías IoT actuales.
  - Definición de los desafíos principales para la autenticación en entornos IoT, así como las limitaciones de los enfoques tradicionales.

- **Fase 2. Análisis del Internet Draft EAP-based Authentication Service for CoAP:** En esta fase se realiza un análisis detallado de la propuesta del Internet Draft, para entender su funcionamiento y los aspectos clave que aportan eficiencia y seguridad en el control de acceso a la red. Los pasos incluyen:

  - Lectura y comprensión exhaustiva del Draft.

- Estudio de la arquitectura y los mecanismos de seguridad del protocolo CoAP-EAP.
- Análisis del protocolo CoAP y su integración con EAP, incluyendo la revisión de los documentos del IETF (RFCs).
- Análisis de la negociación de cipher suite para establecer los conjuntos de cifrado que serán evaluados.
- Estudio del protocolo OSCORE gracias al análisis de su RFC.
- Definición de los requisitos (funcionales y no funcionales) de implementación de la solución en base a las especificaciones del draft.
- Debate para la elección del método EAP que se va a emplear, así como el estudio del funcionamiento del protocolo elegido (EAP-PSK) analizando su RFC.
- Estudio y análisis del servidor AAA (Servidor RADIUS) ya implementado en C y alojado en una máquina virtual.

- **Fase 3. Implementación paso a paso:** Esta fase desglosa el proceso de implementación progresiva del sistema, siguiendo los documentos del IETF y realizando pruebas incrementales para validar cada paso:

  - Implementación inicial de un “esqueleto” de comunicación CoAP-EAP con mensajes estáticos, basados en una traza de Wireshark de ese protocolo.
  - Incorporación de los arrays CBOR para la negociación de cipher suite.
  - Desarrollo de los mecanismos para la generación de canales protegidos y para la derivación de las claves del protocolo EAP-PSK, necesarios para asegurar la correcta autenticación.
  - Implementación de funciones para la generación y el procesamiento de mensajes EAP-PSK, asegurando que los flujos de mensajes cumplen con las especificaciones del draft.
  - Desarrollo de un cliente RADIUS que permita la comunicación directa con el servidor RADIUS alojado en una máquina virtual, como parte integral del proceso de autenticación.

- **Fase 4. Implementación de la solución completa en un entorno de prueba:** La implementación práctica de la última versión del EAP-based Authentication Service for CoAP es la parte central del trabajo. Esta fase incluye:

  - Traducción de los requisitos funcionales del sistema en una estructura modular de código.
  - Desarrollo del sistema completo, utilizando Python como lenguaje base y haciendo uso de la biblioteca aiocoap. Para ello, se lleva a cabo la integración progresiva de todas las partes diseñadas en la fase anterior, ajustando versiones, añadiendo aleatoriedad, mejorando detalles y optimizando la comunicación.

- **Fase 5. Evaluación y pruebas del sistema:** Una vez implementada la solución, se procede a su ejecución y a su evaluación mediante pruebas en un entorno controlado, así como a la visualización de los resultados por consola y en Wireshark.

3.2.- PLANIFICACIÓN TEMPORAL DEL PROYECTO

Con la finalidad de organizar el trabajo que se va a llevar a cabo, se ha hecho una planificación temporal del proyecto marcando unas fechas límite para cada objetivo, considerando que dichas fechas pueden sufrir una variación en virtud de los problemas que se puedan presentar mientras se realiza el proyecto.

Este proyecto se inicia el 1 de febrero de 2024 y se prevé que finalice a mediados o finales de enero de 2025. El número promedio de horas diarias para llevarlo a cabo será de 4/5 horas semanales en periodo de lunes a viernes.

Nombre de tarea	Duración	Comienzo	Fin
Trabajo Fin de Grado	**212 días**	**jue 01/02/24**	**vie 27/12/24**
Planificación	1 día	jue 01/02/24	jue 01/02/24
Análisis del contexto y Revisión del estado del arte	**11 días**	**vie 02/02/24**	**vie 16/02/24**
Repaso conceptos básicos de seguridad e IoT	3 días	vie 02/02/24	mar 06/02/24
Revisión literatura sobre seguridad, métodos de autenticación y tecnologías IoT	6 días	mié 07/02/24	mié 14/02/24
Definición de desafíos	2 días	jue 15/02/24	vie 16/02/24
Análisis del Internet Draft	**22 días**	**lun 19/02/24**	**mar 19/03/24**
Lectura y comprensión exhaustiva del Draft	7 días	lun 19/02/24	mar 27/02/24
Revisión de los documentos del IETF de todos los protocolos que intervinen (CoAP, EAP, OSCORE, EAP-PSK, RADIUS)	12 días	mié 28/02/24	jue 14/03/24
Definición de los requisitos de implementación en base a las especificaciones	3 días	vie 15/03/24	mar 19/03/24
Implementación paso a paso	**79 días**	**mié 20/03/24**	**lun 05/08/24**
Repaso de Python y estudio de la biblioteca aiocoap	7 días	mié 20/03/24	jue 28/03/24
Esqueleto de comunicación CoAP-EAP con mensajes estáticos	18 días	vie 29/03/24	mar 23/04/24
Generación del cliente RADIUS que se comunique con el servidor RADIUS de la máquina virtual	20 días	mié 24/04/24	mar 21/05/24
Generación de los mecanismos para la derivación de las claves de sesión y larga duración, del canal protegido y de los mensajes EAP-PSK	19 días	mié 22/05/24	mar 02/07/24
Generación de la negociación de cryptosuite	5 días	mar 16/07/24	lun 22/07/24
Desarrollo del contexto de seguridad Oscore y protección y desprotección de los mensajes	10 días	mar 23/07/24	lun 05/08/24
Pruebas unitarias de cada parte	72 días	vie 29/03/24	lun 05/08/24
Implementación de la solución completa	**42 días**	**mar 06/08/24**	**jue 03/10/24**
Integración de todas partes desarrolladas en una solución final completa	25 días	mar 06/08/24	mar 10/09/24
Ajuste de versiones, introducción de aleatoriedad y mejora de detalles de comunicación	7 días	mié 11/09/24	jue 19/09/24
Documentación y optimización del código desarrollado	4 días	vie 20/09/24	mié 25/09/24
Evaluación y pruebas	**6 días**	**jue 26/09/24**	**jue 03/10/24**
Ejecución en un entorno controlado, visualización de resultados y evaluación de diferentes escenarios	6 días	jue 26/09/24	jue 03/10/24
Redacción y revisión de la documentación	46 días	vie 04/10/24	mar 10/12/24
Preparación de la presentación	11 días	mié 11/12/24	vie 27/12/24

Figura 3.1.- Planificación temporal del trabajo usando Microsoft Planner

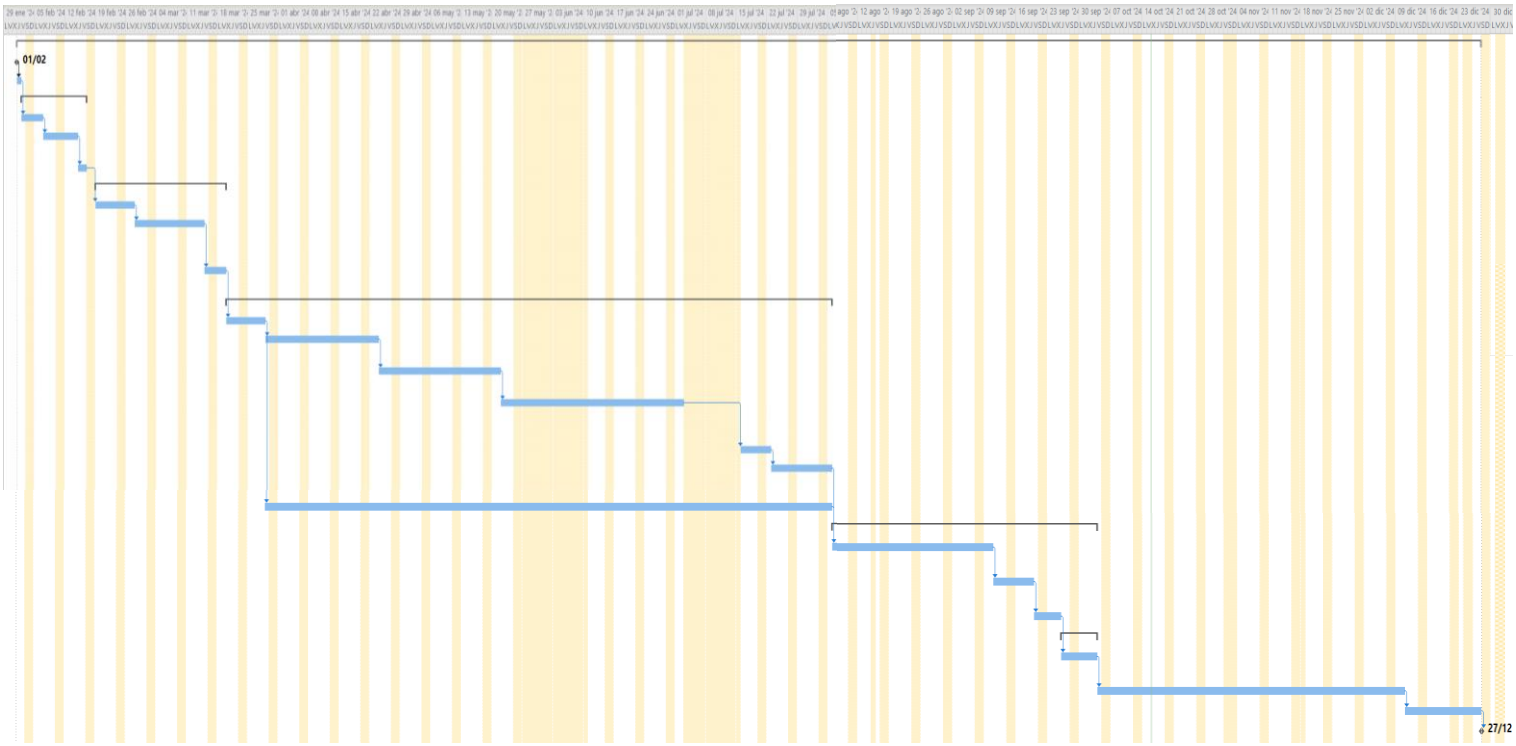


Figura 3.2.- Diagrama de Gantt de la planificación del trabajo

3.3.- ROLES DEL PROYECTO Y DEFINICIÓN

Aunque este proyecto ha sido realizado por una sola persona, en un entorno empresarial o de investigación, proyectos de esta magnitud normalmente involucran a varios profesionales, cada uno con roles específicos. A continuación, se describen los roles más importantes que habrían participado en el desarrollo de un proyecto como este:

- **Director del proyecto:** Es el máximo responsable, encargado de gestionar y supervisar todo el proyecto, asegurando que se cumplan los plazos y los objetivos y asegurando que se ejecute de manera efectiva y correcta. En este contexto, el director del proyecto sería responsable de la planificación y coordinación de todas las fases del desarrollo, desde la investigación inicial hasta la implementación y evaluación del servicio de autenticación. También fijaría los presupuestos, gestionaría los recursos y posibles riesgos técnicos o logísticos que puedan surgir.

- **Analista de sistemas:** Es el encargado de analizar el sistema de información y de definir los requisitos técnicos y funcionales del sistema que se está desarrollando. En este caso, el analista estudiaría la necesidad de un mecanismo de autenticación robusto y flexible en el IoT, evaluando los diferentes métodos de autenticación existentes, y seleccionando la implementación del EAP-based Authentication Service for CoAP como la solución más adecuada. Además, se encargaría de traducir los requerimientos del Internet Draft en especificaciones técnicas que guíen la implementación.

- **Diseñador de arquitectura de red:** La persona que desempeña este rol se enfoca en diseñar la infraestructura y la arquitectura de red sobre la que funcionará el sistema. Para este TFG, el diseñador de la arquitectura aseguraría que el servicio de autenticación basado en EAP se integre de manera eficiente en redes IoT, tomando en cuenta las limitaciones de ancho de banda, los dispositivos involucrados y las tecnologías de comunicación específicas del IoT. Esto incluye la definición de los flujos de autenticación entre los diferentes elementos que componen la arquitectura, es decir, el dispositivo IoT (EAP Peer), el Controlador (EAP Authenticator) y el servidor EAP (Servidor RADIUS).

- **Programador:** El programador se encargará de implementar el código que lleva a cabo la autenticación EAP sobre CoAP. Esto incluye la elección de los lenguajes de programación (en este caso, Python y la biblioteca aiocoap), la implementación de la lógica de autenticación basada en EAP, así como la integración con el protocolo CoAP para asegurar la correcta transmisión de datos en redes de dispositivos con recursos limitados. Además, sería responsable de implementar la derivación de claves, la protección y desprotección de mensajes y las pruebas unitarias.

- **Especialista en seguridad:** Dado que este proyecto implica la implementación de un sistema de autenticación, un especialista en seguridad revisaría las vulnerabilidades potenciales y propondría medidas para asegurar que el sistema de autenticación sea robusto frente a ataques. En este caso, el especialista en seguridad evaluaría la implementación de OSCORE, así como los algoritmos criptográficos

utilizados (como AES-CCM), asegurando que se cumplan los estándares de seguridad en el IoT.

- **Tester o evaluador:** El evaluador es el encargado de probar la implementación en distintos escenarios para asegurarse de que el sistema funciona correctamente y cumple con los requisitos establecidos. Este rol también incluye la elaboración de un plan de pruebas que verifique la interoperabilidad, la eficiencia del proceso de autenticación y la resistencia a posibles ataques o fallos. En este TFG se encargará de elaborar el plan de pruebas para verificar que la implementación cumple con todos los requisitos establecidos en el proyecto.

- **Técnico de despliegue:** Este técnico sería el responsable del despliegue (puesta en marcha) del sistema en un entorno real o simulado, una vez que el Tester comprueba que todo funciona correctamente. Además, el técnico documentaría todo el proceso, generando manuales de usuario y guías de instalación para que otras personas o equipos puedan replicar el entorno de prueba y evaluar el funcionamiento del sistema de autenticación en escenarios similares.

3.4.- PRESUPUESTO

A continuación, se presenta una estimación del presupuesto completo del proyecto. Para la elaboración del mismo se requiere personal y equipamiento hardware y software. Aunque todo el software usado es gratuito, también se puede utilizar la versión extendida de PyCharm (PyCharm Professional) cuyo coste es de 249 €/año.

El presupuesto estimado para el personal, basado en los precios por hora obtenidos a partir de la media salarial de 10 ofertas de trabajo en InfoJobs con perfiles similares a los requeridos para el proyecto, se detalla en la tabla 3.1.

Personal	Horas (5h/día)	Precio/Hora	Total
Director del proyecto	30	55,00 €	1650,00 €
Analista	40	45,00 €	1800,00 €

Diseñador de la arquitectura	70	48,00 €	3360,00 €
Programador	180	38,00 €	6840,00 €
Tester	15	35,00 €	525,00 €
Técnico de despliegue	20	38,00 €	760,00 €
Especialista en seguridad	30	50,00 €	1500,00 €
TOTAL	336	-	**16435,00 €**

Tabla 3.1.- Presupuesto estimado de personal

Se usará un ordenador portátil Gaming HP VICTUS 16-E0015NS con un precio de mercado de 979 € para desarrollar el proyecto. Este es el único hardware necesario tal y como se ve en la tabla 3.2.

Material	Cantidad	Precio/Unidad	Total
HP VICTUS 16-E0015NS	1	979,00 €	979,00 €
		TOTAL	979,00 €

Tabla 3.2.- Presupuesto estimado de hardware

A continuación, se realizarán los presupuestos totales y, seguidamente, se mostrará el presupuesto final que incluye los gastos generales (13 % del total), el beneficio industrial (6 % del total) y el Impuesto sobre el Valor Añadido (IVA del 21 %), como se muestra en la tabla 3.3.

Concepto	Porcentaje	Total
Presupuesto de personal	-	16435,00 €
Presupuesto de hardware	-	979,00 €
Presupuesto de software	-	0,00 €
TOTAL ESTIMADO	-	17414,00 €
Gastos generales	13 %	2263,82 €
Beneficio industrial	6 %	1044,84 €
Total sin IVA	-	20722,66 €
IVA	21 %	4351,76 €
TOTAL CON IVA	-	25074,42 €

Tabla 3.3.- Presupuesto estimado total con IVA

4. Fundamentos teóricos.

En esta sección se va a explicar de la forma más resumida posible en qué consiste el Draft (garantía de orden, URIs, flujo de operación…), las mejoras que ofrece respecto a los enfoques existentes y todos los protocolos que se requieren para implementarlo.

4.1.- QUÉ ES EL INTERNET DRAFT EAP-BASED AUTHENTICATION SERVICE FOR COAP

El "Internet Draft EAP-based Authentication Service for CoAP" es un documento técnico en desarrollo que propone un servicio de autenticación basado en el protocolo EAP (Extensible Authentication Protocol), que es transportado mediante mensajes del protocolo CoAP (Constrained Application Protocol), en el contexto del Internet de las Cosas (IoT). Más concretamente, este documento especifica cómo CoAP puede usarse como una EAP lower layer confiable, independiente de la capa de enlace y para entornos con restricciones. Esta EAP lower layer se llama CoAP-EAP. Uno de los objetivos principales es autenticar un dispositivo IoT habilitado para CoAP (EAP Peer) que pretende unirse a un dominio de seguridad gestionado por un Controlador (EAP Authenticator). El cliente CoAP tiene la opción de contactar con una infraestructura AAA de backend para completar la negociación EAP. Otro aspecto fundamental es que permite derivar el material criptográfico para proteger los mensajes CoAP intercambiados entre el EAP Peer y el EAP Authenticator, basándose en la Seguridad de Objetos para Entornos RESTful Restringidos (OSCORE por sus siglas en inglés), lo que facilita el establecimiento de una asociación de seguridad entre ellos. [53, pp. 1-3]

CoAP es un protocolo de aplicación web creado para dispositivos con recursos limitados, como sensores y actuadores IoT, y diseñado para operar en nodos y redes restringidas con bajo consumo de energía y alta pérdida de paquetes. Proporciona un modelo de comunicación petición/respuesta ligero y eficiente entre puntos finales que permite a los dispositivos intercambiar datos de manera eficiente sobre redes con ancho de banda limitado y alta latencia. Además, admite el descubrimiento incorporado de recursos y servicios e incluye conceptos clave de la Web, como las URIs y los tipos de contenido (tipos MIME) que pueden ser intercambiados a través de Internet. CoAP también facilita la interacción con

HTTP para su integración con la Web, al tiempo que cumple con requisitos especializados como el soporte de multidifusión, muy baja sobrecarga y simplicidad para entornos restringidos. [55, p. 1]

El protocolo EAP, por otro lado, es un marco de autenticación ampliamente utilizado en redes de comunicaciones, que permite la autenticación de usuarios mediante una variedad de métodos como contraseñas, certificados digitales y tokens de seguridad [56, p. 1]. EAP tiene un conjunto de características que lo hacen muy interesante y versátil en comparación con el simple uso de métodos de autenticación por sí solos. El uso de EAP proporciona flexibilidad, es decir, en lugar de utilizar un algoritmo de autenticación específico permite al propietario de los dispositivos IoT decidir qué método de autenticación (EAP-TLS, EAP-PSK, EAP-AKA, EAP-EDHOC…) es más adecuado para las especificaciones del dispositivo y el escenario de implementación. Por ejemplo, si se está tratando con dispositivos muy restringidos que no pueden usar certificados o deben limitar su uso por razones de ahorro de energía, se puede confiar en métodos de EAP como EAP-AKA o EAP-PSK que se basan en claves precompartidas. La decisión de qué método EAP utilizar no tiene que ser negociada y es decidida por el EAP Server, que tiene conocimiento previo sobre el dispositivo y es capaz de elegir el método EAP preferido para realizar la autenticación. [57, p. 12]

EAP también proporciona un marco, llamado Marco de Gestión de Claves de EAP, para derivar material criptográfico que puede ser utilizado para establecer una Asociación de Seguridad (SA por sus siglas en inglés) para proteger la comunicación entre el dispositivo IoT y el Controlador. Además, EAP está estrechamente integrado con las infraestructuras AAA (Authentication, Authorization y Accounting), otorgando a los dispositivos habilitados para EAP los beneficios de AAA. Entre las diferentes características que AAA ofrece, podemos destacar el soporte para Federaciones de Identidad, que facilita la interoperabilidad entre diferentes sistemas o dominios administrativos. Esto resulta especialmente útil en implementaciones multinivel, donde se requiere gestionar el acceso y la autenticación a través de varios niveles de jerarquía o capas en una infraestructura de red. Teniendo AAA, se soporta la autenticación debido a EAP (EAP permite el uso de un servidor de autenticación en la infraestructura de backend, llamado EAP Server, el cual puede implementar algunos o todos los métodos de autenticación, con el EAP Authenticator actuando como

intermediario), pero también se soporta la autorización, lo que dará a la entidad que administra el dominio donde se despliega el nuevo dispositivo información de autorización. La contabilidad o registro de actividades es otra característica que es muy útil, pero a veces pasada por alto. Poder monitorear el uso de los servicios y recursos de la red y proporcionar informes al respecto es útil para diferentes escenarios, como la facturación en caso de que el uso de la red esté sujeto a cargos por utilizarla. [56, p. 3], [57, pp. 12-13]

En resumen, el objetivo principal del "Internet Draft EAP-based Authentication Service for CoAP" es extender el soporte de autenticación de EAP a dispositivos IoT que utilizan CoAP como protocolo de aplicación. Esto proporcionaría una forma estándar y segura de autenticar dispositivos IoT en redes CoAP, lo que facilitaría la interoperabilidad y la seguridad en entornos de IoT.

4.2.- MEJORAS QUE OFRECE

En este apartado se van a comentar las mejoras que ofrece la última versión del EAP-based Authentication Service for CoAP en comparación con los enfoques de autenticación actuales.

En primer lugar, el EAP-based Authentication Service for CoAP introduce una adaptabilidad a entornos restringidos de IoT, siendo ligero en términos de consumo de energía, capacidad de procesamiento y uso de memoria, lo que lo hace ideal para dispositivos con recursos limitados.

Además, el Draft propone una integración nativa directa con el protocolo CoAP, el cual es ampliamente utilizado en IoT para la comunicación entre dispositivos y servidores, lo que asegura una compatibilidad fluida y eficiente con las aplicaciones y dispositivos que emplean este protocolo, lo que simplifica la implementación y mejora la interoperabilidad en el ecosistema IoT. A esto se suma el soporte para múltiples métodos de autenticación, lo que proporciona flexibilidad al permitir a los desarrolladores seleccionar el método más adecuado para sus aplicaciones y escenarios específicos, lo que mejora la adaptabilidad y la seguridad en los entornos heterogéneos de IoT.

Otra ventaja relevante es la seguridad mejorada con EAP, que, al ser un protocolo extensible, añade una capa de seguridad adicional al admitir diferentes métodos de autenticación. Esto, junto con la compatibilidad con estándares y buenas prácticas de seguridad, garantiza que la solución esté alineada con las mejores recomendaciones de seguridad del IETF y otros organismos reconocidos, facilitando su integración con otros estándares y protocolos relacionados con el IoT.

También es importante destacar la facilidad de implementación y despliegue, ya que este servicio se centra en la simplicidad y la eficiencia, lo que reduce la carga de trabajo para los desarrolladores y permite una adopción más rápida y generalizada en el campo del IoT. Además, se ha optimizado el consumo de recursos, logrado mediante el diseño de protocolos y algoritmos que minimizan el consumo de energía, la carga de procesamiento y el uso de memoria, lo que es fundamental en dispositivos con restricciones significativas en estos ámbitos.

El protocolo CoAP, junto con este servicio de autenticación, se presenta como un protocolo eficiente para comunicaciones limitadas, optimizando tanto el tamaño de los mensajes como la sobrecarga de la comunicación, garantizando así un rendimiento óptimo incluso en redes con alta latencia o ancho de banda reducido.

Asimismo, se ha trabajado en la reducción del espacio de almacenamiento requerido, optimizando los algoritmos de cifrado y reduciendo la cantidad de datos necesarios para la autenticación, lo cual es crucial en dispositivos IoT con capacidades de almacenamiento limitadas. De igual forma, se han implementado protocolos de codificación eficiente y optimización de comunicaciones que minimizan la cantidad de datos transmitidos durante la autenticación, reduciendo la carga en la red y el consumo de ancho de banda.

Por último, el servicio introduce el uso de algoritmos ligeros de criptografía, que ofrecen un buen nivel de seguridad con un bajo consumo de recursos, lo que los hace adecuados para dispositivos con limitaciones significativas de CPU y memoria, sin comprometer la seguridad de la autenticación.

4.3.- POR QUÉ SE DECIDE UTILIZAR COAP COMO UNA EAP LOWER LAYER

La estandarización de protocolos para generalizar el proceso conocido como bootstraping (arranque), que tiene lugar cuando el dispositivo, recién puesto en marcha, necesita formar parte del dominio de seguridad de la red de forma segura y que incluye diferentes subprocesos como la autenticación, la autorización, la distribución y la gestión de claves, no fue algo que se considerara específicamente en el IoT hasta hace poco. El proceso de arranque también genera información criptográfica que puede ser utilizada para habilitar tecnologías como OSCORE y DTLS a través de claves precompartidas, lo que se denomina post-bootstraping (fase operacional), que tiene lugar cuando el dispositivo ya es confiable dentro de la red, y necesita establecer nuevas asociaciones de seguridad para otros servicios y/o comunicarse con otros objetos inteligentes. [32, p. 41]

El grupo de trabajo ACE (Authentication and Authorization for Constrained Environments) del IETF ha adoptado el protocolo CoAP-EAP. Se trata de un protocolo que, en jerga EAP, se conoce como una “EAP lower layer” (capa inferior de EAP) y se encarga de transportar el protocolo EAP, entre el EAP Authenticator y el EAP Peer. Además, posibilita la utilización de infraestructuras AAA para externalizar la autenticación, proporcionando así una gran extensibilidad y flexibilidad al proceso de autenticación de los dispositivos IoT. CoAP-EAP es la primera EAP lower layer diseñada específicamente para IoT, teniendo en cuenta las limitaciones de las redes y dispositivos IoT para su diseño.

El IoT, como ya se mencionó en los antecedentes, incluye una amplia gama de dispositivos con capacidades variables (CPU, memoria, vida útil de la batería, etc.), así como diferentes protocolos de comunicación (por ejemplo, Bluetooth, IEEE 802.15.4 o LPWAN) y credenciales (por ejemplo, claves precompartidas o certificados). Esta variedad, que a menudo es restrictiva, requiere repensar enfoques actuales para admitir un dispositivo IoT en una red o dominio de seguridad gestionado por un administrador de dominio, a quien nos referiremos como Controlador (EAP Authenticator) en este documento.

Esta sección discute la idoneidad del protocolo CoAP como una EAP lower layer, y revisa los requisitos impuestos por el protocolo EAP a cualquier protocolo que lo transporte. Lo que EAP espera de sus capas inferiores se cuenta a continuación [53, pp. 18-19]:

- **Transporte no confiable:** EAP no asume que las capas inferiores son confiables, pero puede beneficiarse de una capa inferior confiable. En este sentido, CoAP proporciona un mecanismo de confiabilidad (por ejemplo, a través del uso de mensajes Confirmable).
- **Detección de errores de capa inferior:** EAP se basa en la detección de errores de capa inferior (por ejemplo, CRC, suma de verificación, MIC, etc.). CoAP se basa en UDP/TCP que proporciona un mecanismo de suma de verificación sobre su carga útil.
- **Seguridad de capa inferior:** EAP no requiere servicios de seguridad de las capas inferiores.
- **MTU mínimo:** La unidad de transmisión máxima (MTU) es un parámetro que define el tamaño máximo permitido para un paquete de datos que puede ser transmitido a través de una red sin necesidad de ser dividido o fragmentado. Las capas inferiores deben proporcionar un tamaño de MTU de EAP de 1020 octetos o más. CoAP asume un límite superior de 1024 para su carga útil, lo que cubre los requisitos de EAP.
- **Garantías de orden:** EAP se basa en garantías de orden de capa inferior para su correcto funcionamiento. En este sentido, CoAP proporciona un flujo de paquetes no duplicados y cumple con la "deseable" no duplicación. Además, el RFC3748 indica que cuando EAP se ejecuta sobre una capa inferior confiable, "el temporizador de retransmisión del autenticador debería establecerse en un valor infinito, para que no ocurran retransmisiones en la capa EAP".

Esta comparación muestra cómo CoAP satisface los requisitos de EAP en términos de transporte, detección de errores, seguridad, tamaño de MTU y garantías de orden, lo que lo convierte en una opción adecuada como EAP lower layer. Además, como se comentó anteriormente, el uso de CoAP permite reducir el tamaño de los mensajes y reducir la sobrecarga en la comunicación, algo que es muy beneficioso e importante como se demuestra en el próximo apartado.

4.4.- POR QUÉ ES IMPORTANTE REDUCIR EL TAMAÑO DE LOS MENSAJES

El impedimento en IoT para usar HTTP es, entre otras cosas, su verbosidad. El protocolo HTTP utiliza texto para conformar cada uno de los campos del protocolo, lo que lo hace demasiado pesado para dispositivos y redes restringidos. Para ofrecer un servicio similar para dispositivos IoT, se ideó el protocolo CoAP. CoAP no es una compresión de HTTP, sino un protocolo diferente que implementa un subconjunto de la arquitectura REST (Representational State Transfer), que está optimizado para aplicaciones máquina a máquina y que proporciona métodos como GET, PUT, POST y DELETE [57, p. 11]. Esta es una representación esquemática de un mensaje CoAP:

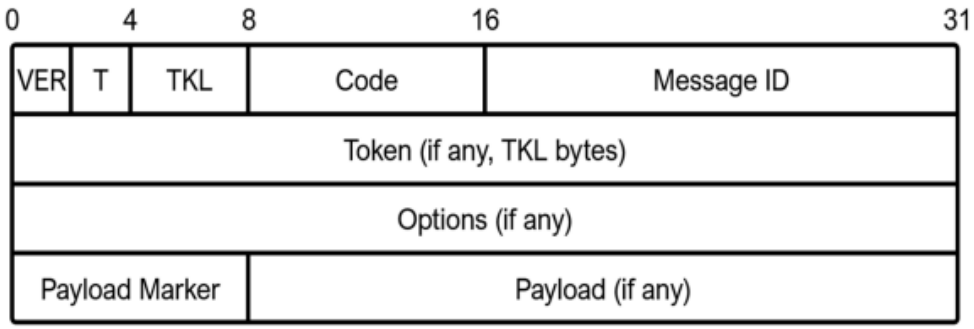


Figura 4.1.- Esquema de un mensaje CoAP

Los mensajes CoAP se codifican en un formato binario sencillo. El formato comienza con una cabecera de 4 bytes de tamaño fijo compuesta por varios campos: la versión (Version) de 2 bits, el tipo (Type) de 2 bits, la longitud del Token (Token Length) de 4 bits, el código (Code) de 8 bits y el ID del mensaje (Message ID) de 16 bits. Le sigue un Token de longitud variable, que puede tener entre 0 y 8 bytes. Tras el Token viene una secuencia de cero o más opciones CoAP (CoAP Options) en formato Type-Length-Value (TLV), opcionalmente seguida de un payload que ocupa el resto del datagrama. Si el payload está presente y tiene una longitud distinta de cero, va precedida de un de un byte 0xFF (Payload Marker), que indica el final de las opciones y el comienzo de la carga útil. [55, pp. 15-17]

Esta reducción en el tamaño de los mensajes, así como la cantidad de bytes intercambiados es claramente interesante y ventajosa para aplicaciones en tecnologías de IoT como 6LoWPAN por varias razones.

En primer lugar, mejora la eficiencia de la red, ya que muchas redes de IoT, como LPWAN o redes celulares, operan con ancho de banda limitado. Al optimizar el tamaño de los mensajes, se facilita un uso más eficiente del ancho de banda, permitiendo comunicaciones más rápidas y confiables entre dispositivos y sistemas de gestión.

Este enfoque también favorece el ahorro de energía, lo cual es esencial para dispositivos IoT alimentados por baterías o fuentes de energía limitadas. La transmisión de grandes cantidades de datos consume una cantidad significativa de energía, lo que puede reducir la vida útil de la batería y requerir recargas frecuentes o reemplazos de baterías. Al reducir el tamaño de los mensajes y la cantidad de bytes intercambiados, el consumo energético se reducirá, prolongando la vida útil de las baterías y disminuyendo los costos de operación y mantenimiento.

Además, la reducción del tamaño de los mensajes tiene un impacto directo en los costos de transmisión de datos, especialmente en redes celulares donde la transmisión de datos puede ser costosa. Menos bytes intercambiados significan menos gastos, lo que hace que las soluciones IoT sean más accesibles y rentables en diferentes aplicaciones y sectores.

También se minimiza la latencia, un aspecto crucial en aplicaciones donde se requiere una respuesta en tiempo real. Al transmitir mensajes más pequeños, los tiempos de viaje entre emisor y receptor se reducen, mejorando la capacidad de respuesta del sistema y brindando una experiencia más fluida para los usuarios.

Por último, esta optimización aligera la carga de procesamiento de los dispositivos IoT, que como ya es sabido, a menudo tienen recursos limitados. Reducir el tamaño de los mensajes alivia el procesamiento necesario para gestionarlos, previniendo congestiones, bloqueos y mejorando el rendimiento general del sistema.

Un estudio llevado a cabo en un artículo de la revista Sensors [58] ha comparado los resultados de someter a la primera versión de CoAP-EAP y a PANATIKI a un conjunto de pruebas para medir varios de estos aspectos. PANATIKI es una implementación optimizada para el bootstrapping en IoT basada en PANA, por lo que es un buen representante de otras alternativas de arranque que utilizan EAP, AAA y tecnologías basadas en claves. Esta

comparación revela importantes mejoras en términos de rendimiento y eficacia en el proceso de arranque en entornos IoT.

CoAP-EAP al tener un tamaño de mensaje más reducido, resulta en un tiempo de arranque más rápido y un notable menor consumo de energía, características cruciales para dispositivos con recursos limitados. En términos de tasa de éxito, CoAP-EAP supera significativamente a PANATIKI, logrando completar aproximadamente el 85% de los procesos de arranque en condiciones de pérdida de paquetes del 0.2, mientras que PANATIKI apenas alcanza el 9% en el mismo escenario. Esto demuestra que, a pesar de que el uso de memoria y el tiempo de procesamiento son áreas donde las mejoras son más limitadas debido a las características del protocolo EAP elegido, CoAP-EAP ofrece una robustez superior en situaciones de alta pérdida de paquetes y un mejor rendimiento general. En conclusión, CoAP-EAP se presenta como una solución más eficiente y confiable para el arranque en IoT, resaltando la necesidad de explorar y diseñar métodos EAP aún más optimizados en el futuro. [58, Sec. 5]

En la sección 6.4 de este Trabajo Fin de Grado se llevará a cabo una comparativa detallada entre la versión original del Internet Draft y la última versión implementada en este documento. Esta comparación analizará tanto el número de mensajes CoAP-EAP intercambiados como el tamaño de los mismos, con el objetivo de evaluar la idoneidad y las mejoras introducidas en la nueva implementación.

4.5.- COAP-EAP

4.5.1.- Arquitectura

La arquitectura de CoAP-EAP [53, p. 4] involucra dos entidades y, opcionalmente, una tercera. Un dispositivo IoT, actuando como el EAP Peer, desea ser autenticado utilizando EAP para unirse a un dominio que es gestionado por un Controlador (EAP Authenticator). El dispositivo IoT actuará como un servidor CoAP para este servicio y el Controlador como un cliente CoAP. La lógica detrás de esta decisión es que las solicitudes

EAP siempre van desde el EAP Server hacia el EAP Peer. En consecuencia, las respuestas EAP van desde el EAP Peer hacia el EAP Server.

CoAP-EAP se ejecuta entre el Controlador y el dispositivo IoT con el propósito de autenticar a este último, de modo que se convierta en una entidad confiable del dominio. Después de CoAP-EAP, se establece una asociación de seguridad entre el Controlador y el dispositivo IoT, lo que implica la integración del dispositivo IoT en el dominio de seguridad.

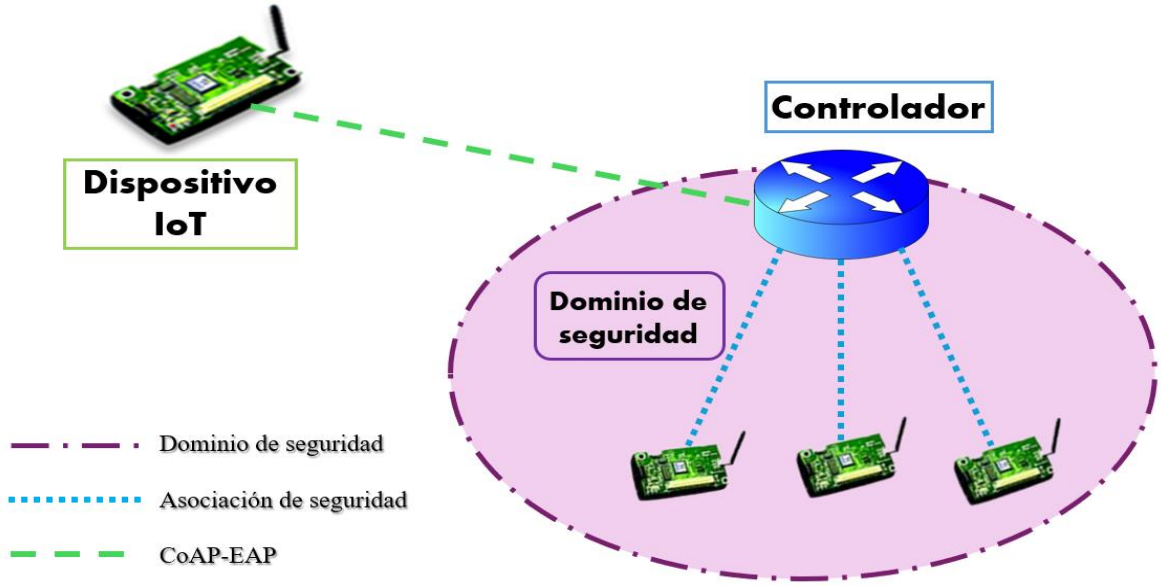


Figura 4.2.- Arquitectura CoAP-EAP

Cabe destacar que el cliente CoAP (EAP Authenticator) puede interactuar con una infraestructura AAA de respaldo cuando se utiliza el modo pass-through de EAP, lo que colocará al EAP Server en el servidor AAA que contiene la información necesaria para autenticar al EAP Peer. Esta entidad es opcional porque el rol del EAP Server también puede ser cumplido por el EAP Authenticator. Esta figura ilustra la arquitectura definida en este documento:

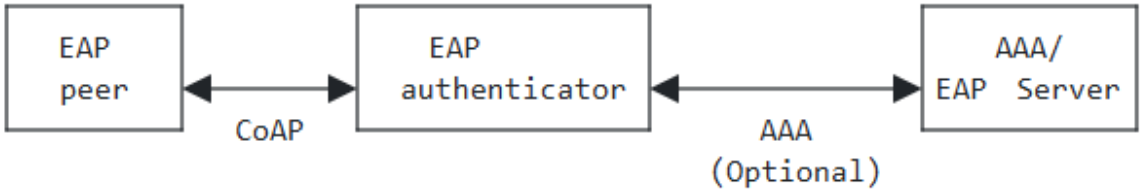


Figura 4.3.- Arquitectura CoAP-EAP con servidor AAA opcional

CoAP-EAP es una aplicación construida sobre CoAP. Sobre la aplicación, hay una máquina de estados EAP que puede ejecutar cualquier método EAP. Para esta especificación, el método EAP debe ser capaz de derivar material criptográfico. CoAP-EAP también depende de los mecanismos de fiabilidad de CoAP para transportar EAP: CoAP sobre UDP con mensajes confirmables o CoAP sobre TCP, TLS y WebSocket.

4.5.2.- Garantía de orden

Inicialmente, para manejar los problemas de mensajes perdidos, se utilizaba la opción Token, que se usa para correlacionar una solicitud CoAP con una respuesta, además de un Message ID (ID de Mensaje), que se utiliza para correlacionar una solicitud CoAP con un Acuse de Recibo (ACK por sus siglas en inglés) CoAP. Dado que CoAP-EAP, por diseño, utiliza una respuesta en el mismo mensaje (piggybacked response) porque se ajusta mejor al modelo de solicitud/respuesta de EAP, ya se tenían las herramientas necesarias, y con redundancia (Token e Message ID) para reconocer mensajes fuera de orden, que serían descartados si llegaban a cualquiera de los puntos finales. Esa opción se demostró que no era la ideal ya que no se puede tener control sobre cómo se desarrolla la funcionalidad específica de la implementación CoAP, por lo que con el tiempo se llegó a la conclusión de que utilizar las URIs para garantizar el orden era la mejor idea, la opción más natural y también una que cualquier implementación de CoAP podría soportar independientemente de la completitud de la implementación.

El formato propuesto para la URI se definió de manera genérica con una raíz común que designa el servicio de autenticación, /b, y completando el location path con un número que indica el paso actual en el proceso de autenticación, /i. El primer valor podría ser seleccionado al azar, y los pasos consecutivos se incrementarían de manera monótona. Si nos encontramos en el primer intercambio del método EAP, la URI donde el cliente CoAP accedería podría ser /b/6, la segunda /b/7. Pero el autor principal de CoAP consideró que el enfoque específico de forzar el esquema utilizando números era demasiado restrictivo. Él sugirió no forzar ningún esquema y dejar que la implementación especifique el identificador para cada paso del proceso de autenticación. En otras palabras, propuso utilizar el principio de Hypermedia as the Engine of Application State (HATEOAS) en CoAP. [57, pp. 27-29]

Entonces, ¿cómo funciona HATEOAS y cómo beneficiaría a CoAP-EAP? HATEOAS se basa en un concepto básico: una respuesta proporcionaría enlaces a recursos adicionales. Estos enlaces serían utilizados por los clientes para comunicarse con el servidor, y estas interacciones podrían resultar en cambios de estado. Como ocurre, CoAP-EAP tiene internamente un estado de arranque, que evoluciona y envía mensajes EAP a la máquina de estados EAP en el orden requerido. Por lo tanto, HATEOAS permitiría materializar los diferentes estados del proceso de autenticación en forma de diferentes URIs. Cada paso durante la autenticación EAP accede a un nuevo recurso en el servidor CoAP (EAP Peer) y el recurso anterior se elimina, una vez que se crea el nuevo recurso, lo que indica el recurso que procesará el siguiente paso de la autenticación EAP.

Pero ¿qué valor se utilizaría en lugar de un número en orden incremental como se sugirió? Este valor se dejaría a la implementación, y se denotaría generalmente como /x, con la única restricción de que el valor elegido debe ser utilizado por el cliente CoAP en el siguiente paso del proceso de autenticación.

4.5.3.- URIS

CoAP utiliza los esquemas de URI "coap" y "coaps" para identificar los recursos de CoAP y proporcionar un medio para localizar dichos recursos. Los recursos se organizan jerárquicamente y están gobernados por un posible servidor de origen CoAP que escucha solicitudes CoAP ("coap") o solicitudes CoAP aseguradas con DTLS ("coaps") en un puerto UDP específico. El servidor CoAP se identifica mediante el componente de autoridad de la sintaxis genérica, que incluye un componente de host y un número de puerto UDP opcional. El resto del URI se considera como la identificación de un recurso que puede ser operado mediante los métodos definidos por el protocolo CoAP. Los esquemas de URI "coap" y "coaps" pueden compararse, respectivamente, con los esquemas de URI "http" y "https".

coap-URI = "coap:" "//" host [":" port] path-abempty ["?" query]

El host (campo Uri-Host) no debe estar vacío; si se recibe una URI con una autoridad ausente o un host vacío, se debe considerar inválido. El subcomponente del puerto (campo Uri-Port) indica el puerto UDP en el que se encuentra el servidor CoAP.

La ruta identifica un recurso (campo Uri-Path) dentro del ámbito del host y del puerto. Consiste en una secuencia de segmentos de ruta separados por un carácter de barra inclinada ("/").

La consulta (campo Uri-Query) sirve para parametrizar aún más el recurso. Consiste en una secuencia de argumentos separados por un carácter de ampersand ("&"). Un argumento suele tener la forma de un par "clave=valor".

El esquema URI "coap" admite el prefijo de ruta "/.well-known/" definido por el RFC5785 para "ubicaciones bien conocidas" en el espacio de nombres de un host. Esto permite el descubrimiento de políticas u otra información sobre un host (metadatos a nivel de sitio), como los recursos alojados. [55, pp. 59-60]

El uso del URI .well-known es algo que surgió en una de las últimas revisiones del draft. Siempre se ha asumido que el dispositivo IoT tendrá acceso al mecanismo de descubrimiento integrado en CoAP a través de /.well-known/core, para obtener información sobre los diferentes servicios disponibles y cómo acceder a ellos, pero realmente no se vio la necesidad de especificar también una URI .well-known específicamente para la autenticación.

Tener una URI .well-known asignado tendría ventajas claras como que los usuarios tendrán acceso directo al servicio de autenticación, pero a su vez tendría algunos inconvenientes de seguridad, pues este acceso directo podría facilitar ataques de denegación de servicio (DoS), ya que un atacante podría saturar el servicio de autenticación con el envío de solicitudes masivas. Aunque se podría argumentar que la exposición del URI incrementa la superficie de ataque, la naturaleza de este riesgo depende en gran parte de las medidas de seguridad implementadas, como la autenticación mutua y las protecciones contra abusos.

En este caso, se entendió que, siendo la autenticación un servicio básico, la exposición se dará independientemente de tener una URI .well-known o no. Así que, tenerlo reconocido como un servicio primario es algo que era inevitable a menos que en futuras revisiones se decida lo contrario. [57, p. 37]

4.5.4.- Flujo de operación

El modelo de interacción de CoAP es similar al modelo cliente/servidor de HTTP. Sin embargo, las interacciones máquina a máquina suelen consistir en una implementación de CoAP que actúa tanto en roles de cliente como de servidor. Una solicitud de CoAP es análoga a una de HTTP y es enviada por un cliente para solicitar una acción, usando un código de método (GET, PUT, POST o DELETE), sobre un recurso identificado por una URI en un servidor. Luego, el servidor envía una respuesta con un Código de Respuesta, que puede incluir una representación del recurso.

A diferencia de HTTP, CoAP maneja estos intercambios de manera asincrónica sobre un transporte orientado a datagramas, como UDP. CoAP define cuatro tipos de mensajes: Confirmable, Non-confirmable, Acknowledgement y Reset. Las solicitudes pueden ser llevadas en mensajes de tipo Confirmable y Non-confirmable, y las respuestas pueden ser llevadas en estos mensajes, así como de manera simultánea en mensajes de tipo Acknowledgement. [55, p. 10]

El modelo de mensajería de CoAP se basa en el intercambio de mensajes sobre UDP entre puntos finales. CoAP, como ya vimos en apartados previos, utiliza un encabezado binario corto que tiene una longitud fija de 4 bytes, al cual pueden seguir unas opciones binarias compactas y una carga útil. Este formato de mensaje lo comparten tanto las solicitudes como las respuestas. Cada uno de los mensajes contiene un Message ID y un Token para garantizar la detección de duplicados, asegurar la confiabilidad y correlacionar una solicitud con una respuesta.

La confiabilidad se proporciona marcando un mensaje como Confirmable (CON). Este tipo de mensaje se retransmite utilizando un tiempo de espera por defecto y un retroceso exponencial entre retransmisiones, hasta que el destinatario envía un mensaje Acknowledgement (ACK) con el mismo Message ID desde el punto final correspondiente. Cuando un destinatario no es capaz de procesar un mensaje Confirmable, es decir, no puede ni siquiera proporcionar una respuesta de error adecuada, lo que hace es responder con un mensaje de Reset (RST) en lugar de con un Acknowledgement (ACK).

Un mensaje que no requiere transmisión confiable, por ejemplo, cada medición individual de una secuencia de datos de sensores puede enviarse como un mensaje Non-confirmable (NON). Estos no son confirmados, pero tienen un Message ID y un Token.

Las semánticas de solicitud y respuesta de CoAP se transmiten en mensajes CoAP, que incluyen un Código de Método o un Código de Respuesta, respectivamente. La información opcional (o por defecto) de las solicitudes y respuestas, como la URI y el tipo de formato de la carga útil, se transmite como opciones de CoAP. [55, pp. 10-12]

En CoAP-EAP, el EAP Peer solo tendrá una sesión de autenticación con un EAP Authenticator específico y no procesará ninguna otra autenticación EAP en paralelo (con el mismo EAP Authenticator). Es decir, se permite una única autenticación EAP en curso para el mismo EAP Peer y el mismo EAP Authenticator. Sin embargo, el EAP Authenticator puede tener sesiones EAP paralelas con varios EAP Peers. [53, p. 5]

Esta información es útil para entender el flujo de operación de COAP-EAP, que es el siguiente [53, pp. 6-9]:

- **Paso 0:** El dispositivo IoT (EAP Peer) debe iniciar el proceso de autenticación enviando una solicitud POST a la URI "/.well-known/coap-eap" (mensaje de activación). El método POST solicita que la representación incluida en la solicitud sea procesada. La función real que realiza el método POST la determina el servidor de origen y depende del recurso de destino. Generalmente, esto resulta en la creación de un nuevo recurso o en la actualización del recurso de destino. Si se ha creado un recurso en el servidor, la respuesta devuelta por el servidor debería tener un Código de Respuesta 2.01 (Created) e incluir la URI del nuevo recurso en una secuencia de una o más opciones Location-Path y/o Location-Query. Si se ha modificado el estado de un recurso existente en el servidor sin crear uno nuevo, la respuesta será 2.04 (Changed). De modo que el método POST no es seguro (puesto que cambia el estado del servidor al poder crear, modificar o eliminar recursos) ni idempotente (porque cada solicitud puede generar un resultado diferente). [55, p. 47]

  Este mensaje lleva la opción CoAP 'No-Response' para evitar esperar por una respuesta que no es necesaria y es la única instancia en la que el EAP Authenticator

actúa como un servidor CoAP y el EAP Peer como un cliente CoAP. El mensaje también incluye una URI en la carga útil del mensaje para indicar a qué recurso (por ejemplo, '/a/eap/1') debe enviar el EAP Authenticator el siguiente mensaje. El nombre del recurso es seleccionado por el servidor CoAP.

Al generar la URI para un recurso de un paso de la autenticación, el recurso podría tener el siguiente formato: "path/eap/counter", donde:

- "path" es una ruta local en el dispositivo para hacer la ruta única. Esto podría omitirse si se desea.
- "eap" es el nombre que indica que la URI es para el EAP Peer. Esto no tiene significado para el protocolo, pero ayuda en la depuración.
- "counter" es un número único que se incrementa con cada nueva solicitud EAP.

Después de esto, el intercambio continúa con el EAP Authenticator como un cliente CoAP y EAP Peer como un servidor CoAP.

- **Paso 1:** El Controlador (EAP Authenticator) envía un mensaje POST al recurso indicado por el dispositivo IoT en el Paso 0 ('/a/eap/1'). La carga útil de este mensaje contiene el primer mensaje EAP (EAP Request/Identity), el Recipient ID del EAP Authenticator (RID-C) para OSCORE y puede contener un array CBOR que incluye una lista con las cipher suites (CS-C) propuestas para OSCORE. Si no se incluye la lista, se utiliza la cipher suite (conjunto de cifrado) predeterminada para OSCORE.

- **Paso 2:** El dispositivo IoT procesa el mensaje POST pasando la solicitud EAP (EAP Req/Id) a la máquina de estados del EAP Peer, que devuelve una respuesta EAP (EAP Response/Identity); asigna un nuevo recurso al proceso de autenticación en curso (por ejemplo, '/a/eap/2') y elimina el anterior ('/a/eap/1'). Finalmente, envía una respuesta '2.01 Created' con el nuevo identificador de recurso en las opciones Location-Path (y/o Location-Query) para el siguiente paso. El objetivo de incluir una combinación de estas opciones en una respuesta Created es indicar la ubicación del recurso creado como resultado de una solicitud POST. La ubicación se resuelve en relación con la URI de la solicitud. Cada Opción Location-Path especifica un

segmento de la ruta absoluta al recurso, y cada Opción Location-Query especifica un argumento que parametriza el recurso. [55, p. 57]

En la respuesta EAP, el Recipient ID del EAP Peer (RID-I) y la cipher suite seleccionado para OSCORE (CS-I) están en la carga útil. Cuando el EAP Authenticator recibe el EAP Resp/Id, lo envía al EAP Server que, en función de la identidad del dispositivo (NAI, por ejemplo, iotdevice@uniovi.es), elige el método EAP que se va a utilizar (en este documento el método elegido por el servidor será EAP-PSK). En este paso, el dispositivo IoT puede crear un contexto de seguridad OSCORE. La clave requerida, conocida como Clave de Sesión Maestra (MSK por sus siglas en inglés), estará disponible una vez que la autenticación EAP sea exitosa en el paso 7.

- **Paso 3-6:** A partir de este momento, el Controlador y el dispositivo IoT intercambiarán paquetes EAP relacionados con el método EAP (se usa EAP-X para indicar que puede usarse cualquier método EAP permitido), transportados en la carga útil del mensaje CoAP. El Controlador utilizará el método POST para enviar solicitudes EAP al dispositivo IoT. El dispositivo IoT utilizará una respuesta para llevar la respuesta EAP en la carga útil. Las solicitudes y respuestas EAP se representan en la Figura 4.4 utilizando la nomenclatura EAP-X-Req y EAP-X-Resp respectivamente. Cuando llega un mensaje POST ('/a/eap/2') que lleva un mensaje de solicitud EAP, si es procesado correctamente por la máquina de estados del EAP Peer, devuelve una respuesta EAP. Junto con cada respuesta EAP, se crea un nuevo recurso (por ejemplo, '/a/eap/3') para procesar la siguiente solicitud EAP y se borra el recurso actual ('/a/eap/2'). De esta manera se logra la garantía de orden. Finalmente, se envía una respuesta EAP en la carga útil de una respuesta CoAP que también indicará el nuevo recurso en las opciones Location-Path (y/o Location-Query). En caso de error al procesar un mensaje legítimo, el servidor devolverá un '4.00 Bad Request'.

- **Paso 7:** Cuando la autenticación EAP termina con éxito, el Controlador obtiene la MSK exportada por el método EAP, un mensaje éxito (EAP Success) e información de autorización (por ejemplo, el tiempo de vida de la sesión). Tras esto, crea el contexto de seguridad OSCORE utilizando la MSK, y el Sender ID (ID de

Remitente) y el Recipient ID (ID de Destinatario) de ambas entidades intercambiados en los pasos 1 y 2. Luego, el Controlador envía el mensaje POST protegido con OSCORE para la confirmación de clave, incluyendo el EAP Success. También puede enviar un tiempo de vida de sesión, en segundos, que se representa con un número entero sin signo en un objeto CBOR. Si este tiempo de vida de sesión no se envía, el dispositivo IoT asume un valor predeterminado de 8 horas. La verificación del mensaje POST protegido por OSCORE recibido, utilizando el RID-I enviado en el paso 2, es considerada por el dispositivo IoT como una indicación alternativa de éxito. La máquina de estados del EAP Peer interpreta la indicación alternativa de éxito de manera similar a la llegada de un EAP Success y devuelve la MSK, que se utiliza para definir el contexto de seguridad OSCORE en el dispositivo IoT y para procesar el mensaje POST protegido recibido del EAP-Authenticator.

- **Paso 8:** Si la autenticación EAP y la verificación del mensaje POST protegido por OSCORE en el paso 7 tienen éxito, entonces el dispositivo IoT responde con un '2.04 Changed' protegido por OSCORE. A partir de este momento, la comunicación con el último recurso (por ejemplo, '/a/eap/(n)') debe estar protegida con OSCORE. Si la política de aplicación lo permite, el mismo contexto de seguridad OSCORE puede usarse para proteger la comunicación con otros recursos entre los mismos puntos finales.

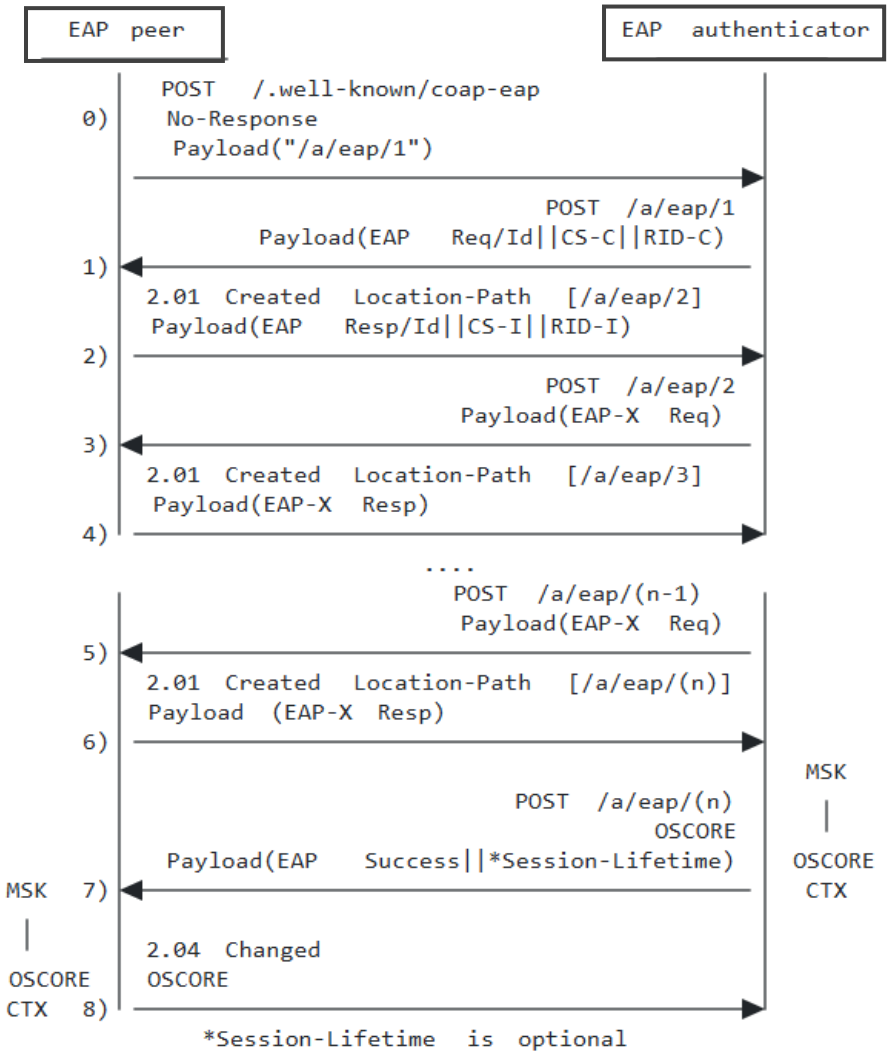


Figura 4.4.- Flujo de operación de CoAP-EAP

4.5.5.- Cipher Suite

La negociación de cipher suite en CoAP-EAP es el proceso mediante el cual el cliente y el servidor acuerdan qué algoritmos criptográficos utilizarán para proteger la comunicación como parte del intercambio de autenticación. Una cipher suite es un conjunto de algoritmos criptográficos que definen cómo se establecerá la asociación de seguridad, y cómo ésta manejará la confidencialidad, la integridad y la autenticación de los mensajes en una comunicación protegida. Una cipher suite incluye:

- Un **algoritmo AEAD** (como AES_128_GCM o ChaCha20/Poly1305).
- Un **algoritmo HASH** (como SHA-256 o SHA-384).

De modo que esta negociación asegura que ambos extremos de la comunicación estén alineados en los algoritmos que utilizarán para garantizar la seguridad de los mensajes intercambiados, ya sea para cifrado, autenticación o ambas.

Se tuvo que reconsiderar la negociación de cipher suite en CoAP-EAP y proporcionar un mecanismo para lograrlo. Para esto, no hay otra forma que enviar la información de las cipher suites soportadas en un mensaje de solicitud y confirmar la elegida en una respuesta. En todos los mensajes, la carga útil ya está llevando mensajes EAP, por lo tanto, se necesita o bien generar una estructura compleja para llevar todo en la carga útil o crear otra opción CoAP para llevar esta información.

Una solución adaptada puede no ser el mejor procedimiento para la estandarización. Se estaría imponiendo que todos los sistemas que busquen beneficiarse de CoAP-EAP tendrían que restringir sus implementaciones de CoAP a aquellas que implementen la opción de Cipher Suite. Por lo tanto, se optó por una solución que no requiera modificaciones en una implementación de CoAP.

Se logró la negociación de cipher suite utilizando la estructura "compleja" en la carga útil de CoAP. La primera idea fue usar un array CBOR que contenga dos elementos: 1) la carga útil de EAP primero y 2) el contenido relacionado con la cipher suite. El problema con este enfoque es que se necesitaría ser consistente y usar una estructura CBOR en todos los mensajes para llevar todos los paquetes EAP, lo que impone una sobrecarga que no se había tenido en cuenta hasta ahora.

Entonces, para evitar la sobrecarga adicional, lo que se hizo es dejar el paquete EAP en la carga útil del paquete CoAP tal como está. El paquete EAP por sí solo proporciona la información necesaria para conocer la longitud, y así saber que ha terminado de leer el paquete EAP y que lo que sigue es otra cosa. Aquí es donde comienza la estructura CBOR, que contiene la información que se quiere transmitir, es decir, la negociación de la cipher suite. [55, pp. 30-33]

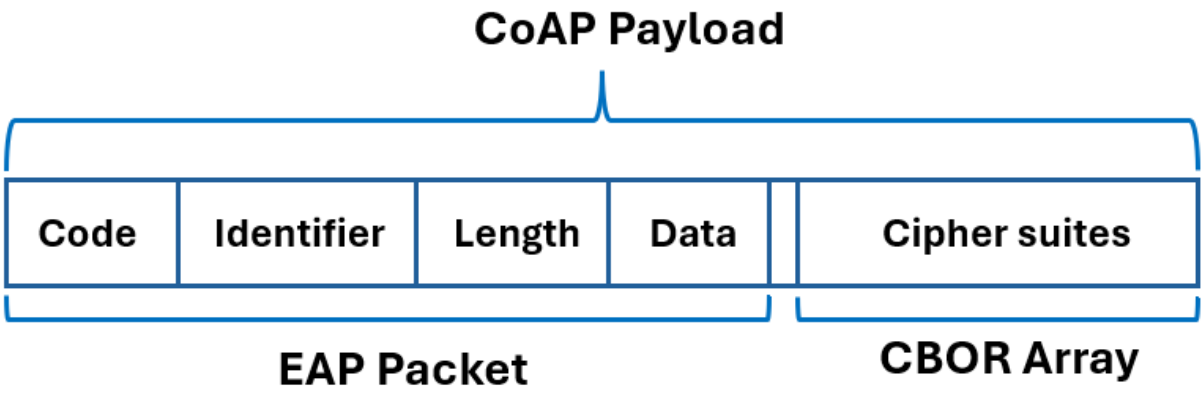


Figura 4.5.- Estructura de un payload del protocolo CoAP con array CBOR

La estructura de datos CBOR utilizada para este propósito se define de la siguiente manera:

CoAP-EAP_Info = {
? 1 : [+ int], ; Cipher Suite (CS-C o CS-I)
? 2 : bstr, ; RID-C
? 3 : bstr, ; RID-I
? 4 : uint ; Tiempo de vida de la sesión
}

Los parámetros en esta estructura de datos CBOR llevan contenida la siguiente información:

1. **Cipher Suite (Índice 1):**

   - Este campo contiene un array que lista los algoritmos COSE propuestos o seleccionados para OSCORE.
   - Si este campo se incluye en una solicitud, indica la cipher suite propuesta. Si se incluye en una respuesta, especifica la cipher suite elegida.

2. **RID-I (Índice 2):**

   - Este campo contiene el Recipient ID del dispositivo IoT.

- El Controlador utiliza este valor como Sender ID para su Sender Context de OSCORE y el dispositivo IoT usa este valor como Recipient ID para su Recipient Context de OSCORE.

3. **RID-C (Índice 3):**

   - Este campo contiene el Recipient ID del Controlador.
   - El dispositivo IoT usa este valor como Sender ID para su Sender Context de OSCORE y el Controlador utiliza este valor como Recipient ID para su Recipient Context de OSCORE.

4. **Tiempo de vida de la sesión (Índice 4):**

   - Este campo indica el tiempo (en segundos) durante el cual la sesión es válida. Ayuda a ambas entidades a gestionar la expiración y renovación de la sesión.

La estructura CBOR permite la adición de nuevos campos en el futuro. Para fines de experimentación y pruebas los índices del 65001 al 65535 están reservados. Esto proporciona flexibilidad a los desarrolladores para introducir nuevos parámetros o elementos de información que puedan ser necesarios para casos de uso futuros sin entrar en conflicto con los índices estándar. [53, pp. 14-15]

Esta estructura de datos CBOR proporciona un medio compacto y extensible para intercambiar información crítica en CoAP-EAP, asegurando tanto flexibilidad como compatibilidad con desarrollos futuros del protocolo.

De esta manera, se ha logrado el objetivo deseado de tener una negociación de cipher suite, sin tener que recurrir a crear una nueva opción de CoAP y sin tener una sobrecarga imprevista en todo el intercambio CoAP-EAP. El requisito en este caso es admitir CBOR.

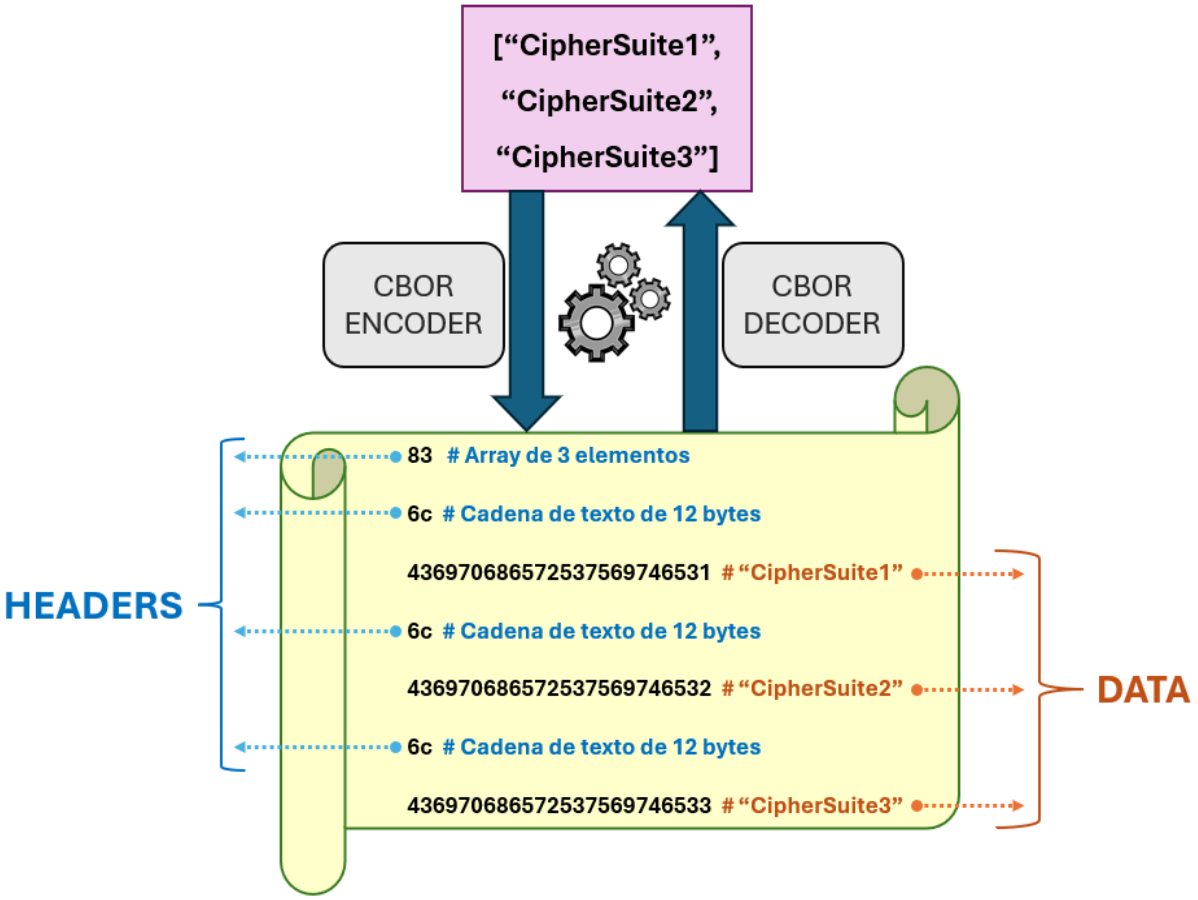


Figura 4.6.- Ejemplo de estructura CBOR para el envío de la negociación de cipher suite

Una consideración final que debe hacerse con respecto a la negociación de cipher suite es proporcionar el intercambio con mecanismos para prevenir un ataque de degradación. Un ataque de degradación se realiza cuando un atacante altera la lista de cipher suites propuestas para establecer los valores a los más débiles posibles, intentando realizar un ataque sobre la información intercambiada. Hay diferentes formas de prevenir el ataque de degradación, pero una de ellas sería utilizando la información recibida y vinculándola con la clave generada. Si la lista enviada por el cliente es diferente a la recibida por el servidor, la clave generada sería diferente, detectando así que algo salió mal. Por ejemplo, si la lista inicial fuera [1,2] y el atacante la cambia a [0,1], la MSK resultante del EAP Peer tendría un componente del proceso de derivación del material de clave con [0,1] y el servidor con [1,2]. Las claves resultantes diferirían y al intentar confirmar las claves el proceso fallaría. [55, pp. 30-31]

4.6.- OSCORE

4.6.1.- Qué es y por qué surgió

En el estándar CoAP se establece que para asegurar las comunicaciones se utilizará el protocolo DTLS. Esta era la norma hasta hace algún tiempo, cuando se añadió otro protocolo a esta lista. Este protocolo es el Object Security for Constrained RESTful Environments (OSCORE). La razón para agregar un nuevo protocolo fue que DTLS no podía proporcionar seguridad de extremo a extremo en todos los diferentes escenarios de comunicación que proporciona CoAP. Cuando se utilizan proxies de CoAP, estos rompen el canal de DTLS ya que la comunicación termina con ellos y la información no llega protegida al punto final de comunicación previsto. Para resolver este problema, OSCORE proporciona seguridad a nivel de aplicación. Por lo tanto, no importa cómo la tecnología subyacente maneje las comunicaciones ya que el contenido de la aplicación se mantendrá inalterado. OSCORE utiliza CBOR Object Signing and Encryption (COSE) para su funcionamiento. [57, p. 16]

4.6.2.- Contexto de seguridad OSCORE

OSCORE requiere que el cliente y el servidor establezcan un contexto de seguridad con material criptográfico compartido que se utilice para procesar los objetos COSE. OSCORE utiliza COSE con un algoritmo AEAD (Authenticated Encryption with Associated Data) para proteger los datos de los mensajes entre un cliente y un servidor [59, p. 8]. Se supone que este material criptográfico compartido está preestablecido o acordado mediante otro protocolo. De hecho, el protocolo EDHOC es una propuesta para este propósito, actualmente en desarrollo por el Lightweight Authenticated Key Exchange (LAKE) Working Group. [57, p. 16]

El contexto de seguridad es el conjunto de elementos de información necesarios para realizar las operaciones criptográficas en OSCORE. Para cada extremo, el contexto de seguridad se compone de un “Common Context” (Common Context), un “Sender Context” (Contexto de Remitente) y un “Recipient Context” (Contexto de Destinatario).

Los extremos protegen los mensajes a enviar utilizando el Sender Context y verifican los mensajes recibidos utilizando el Recipient Context; ambos contextos se derivan del Common Context, el cual contiene los algoritmos y el material clave utilizados para generar el contexto de seguridad OSCORE, y de otros datos. [59, p. 9]

Es decir, una vez que el contexto OSCORE se genera en ambos extremos, el cliente utiliza el Sender Context para proteger el mensaje CoAP. El servidor verifica la solicitud con el Recipient Context, que es el mismo que el Sender Context del cliente. La respuesta se protege con el Sender Context del servidor, que es el mismo que el Recipient Context del cliente [57, p. 16]. Los clientes y servidores deben ser capaces de recuperar el contexto de seguridad correcto.

Un extremo utiliza su Sender ID (SID) para derivar su Sender Context; el otro extremo utiliza el mismo ID, ahora llamado Recipient ID (RID), para derivar su Recipient Context. En la comunicación entre dos extremos, el Sender Context de un extremo coincide con el Recipient Context del otro extremo y viceversa. Por lo tanto, aunque los dos contextos de seguridad identificados por los mismos IDs en los dos extremos no son idénticos, están relacionados de manera complementaria.

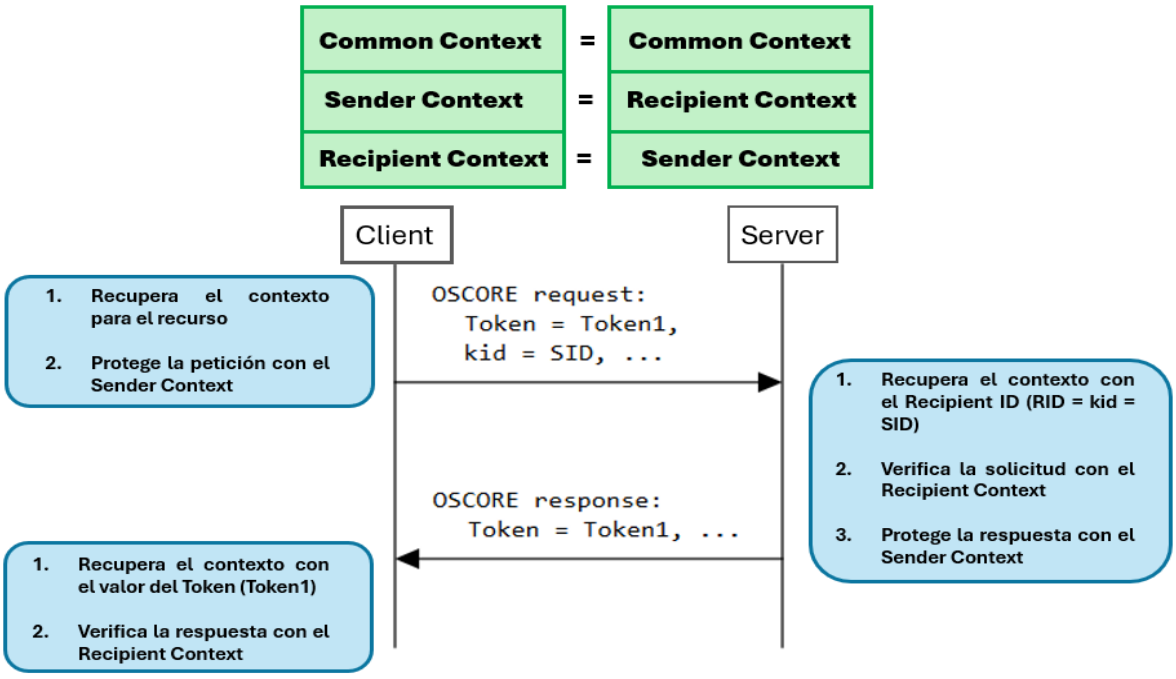


Figura 4.7.- Recuperación y uso del contexto de seguridad OSCORE

El Common Context contiene los siguientes parámetros:

- **Algoritmo AEAD:** El algoritmo AEAD de COSE que se utilizará para el cifrado.
- **Algoritmo HKDF:** Una función de derivación de claves basada en HMAC (HKDF) utilizada para derivar la Sender Key, la Recipient Key y el Common IV.
- **Master Secret:** Cadena de bytes aleatoria de longitud variable utilizada para derivar las claves AEAD y el Common IV.
- **Master Salt:** Cadena de bytes opcional de longitud variable que contiene la sal utilizada para derivar las claves AEAD y el Common IV.
- **Context ID:** Cadena de bytes opcional de longitud variable que proporciona información adicional para identificar el Common Context y derivar las claves AEAD y el Common IV.
- **Common IV:** Cadena de bytes derivada de Master Secret, de la Master Salt y del Context ID. Se utiliza para generar el nonce AEAD. Tiene la misma longitud que el nonce del Algoritmo AEAD.

El Sender Context contiene los siguientes parámetros:

- **Sender ID:** Cadena de bytes utilizada para identificar el Sender Context, derivar las claves AEAD y el Common IV, y contribuir a la unicidad de los nonces AEAD. La longitud máxima la determina el Algoritmo AEAD.
- **Sender Key:** Cadena de bytes que contiene la clave simétrica AEAD para proteger los mensajes a enviar. Se deriva del Common Context y del Sender ID. La longitud está determinada por el Algoritmo AEAD.
- **Sender Sequence Number:** Entero no negativo utilizado por el remitente para enumerar las solicitudes y ciertas respuestas, por ejemplo, notificaciones de Observación. Se utiliza como "Partial IV" para generar nonces AEAD únicos. El valor máximo lo determina el Algoritmo AEAD.

El Recipient Context contiene los siguientes parámetros:

- **Recipient ID:** Cadena de bytes utilizada para identificar el Recipient Context, derivar las claves AEAD y el Common IV, y contribuir a la unicidad de los nonces AEAD. La longitud máxima la determina el Algoritmo AEAD.
- **Recipient Key**: Cadena de bytes que contiene la clave simétrica AEAD para verificar los mensajes recibidos. Se deriva del Common Context y del Recipient ID. La longitud está determinada por el Algoritmo AEAD.
- **Replay Window (sólo para el servidor):** La Replay Window utilizada para verificar las solicitudes recibidas.

Todos los parámetros, excepto el Sender Sequence Number (Número de secuencia del remitente) y la Replay Window (Ventana de Repetición), son inmutables una vez que se establece el contexto de seguridad. Un extremo puede liberar memoria al no almacenar el Common IV (IV Común), la Sender Key (Clave de Remitente) y la Recipient Key (Clave de Destinatario), derivándolos cuando sea necesario. Alternativamente, un extremo puede liberar memoria al no almacenar la Master Secret y la Master Salt después de que se hayan derivado los demás parámetros.

Los extremos pueden actuar tanto de cliente como de servidor, utilizando el mismo contexto de seguridad en ambos casos. No obstante, al cambiar de rol, no deben modificar el Sender ID ni el Recipient ID, lo que significa que el conjunto de claves AEAD a utilizar no se ve afectado por el cambio de rol. [59, pp. 9-11]

4.6.3.- Establecimiento de los parámetros del contexto de seguridad

Cada extremo deriva los parámetros en el contexto de seguridad a partir de un pequeño conjunto de parámetros de entrada. Estos parámetros de entrada deben ser preestablecidos:

- **Master Secret**
- **Sender ID**
- **Recipient ID**

Los siguientes parámetros de entrada pueden ser preestablecidos. En caso de que alguno de estos parámetros no esté preestablecido, se utiliza el valor predeterminado indicado a continuación:

- **Algoritmo AEAD**
  - Predeterminado es AES-CCM-16-64-128.
- **Master Salt**
  - El valor predeterminado es la cadena de bytes vacía.
- **Algoritmo HKDF**
  - El predeterminado es HKDF SHA-256.
- **Replay Window**
  - El mecanismo predeterminado es una ventana deslizante anti-repetición con un tamaño de ventana de 32 mensajes o paquetes.

Todos los parámetros de entrada deben ser conocidos y acordados por ambos extremos aunque la Replay Window puede ser diferente en los dos. La forma en que se preestablecen los parámetros de entrada depende de la aplicación específica. [59, pp. 11-12]

4.6.4.- Derivación de las claves de remitente y de receptor y del Common IV

El HKDF debe ser uno de los algoritmos basados en HMAC definidos para COSE. HKDF SHA-256 es obligatorio de implementar. Los parámetros Sender Key, Recipient Key y Common IV del contexto de seguridad deberán ser derivados de los parámetros de entrada usando el HKDF, que consiste en la composición de los pasos HKDF-Extract y HKDF-Expand:

Parámetro de salida = HKDF (sal, IKM, info, L)

Ecuación 4.1.- Obtención de los parámetros de salida (claves e IV) del contexto OSCORE

Donde:

- **sal** es la Master Salt

- **IKM** es un término utilizado en criptografía para referirse a los datos iniciales utilizados como entrada en un proceso de derivación de claves. En este caso IKM hace referencia a la Master Secret.
- **info** es la serialización de una matriz CBOR que consiste en:

  info = [id : bstr, id_context : bstr / nil, alg_aead : int / tstr, type : tstr, L : uint,]

  Donde:
  - **id** es el Sender ID o el Recipient ID al derivar la Sender Key y la Recipient Key respectivamente y la cadena de bytes vacía al derivar el Common IV.
  - **id_context** es el Context ID, o nil (valor especial que indica la ausencia de un valor) si el Context ID no se proporciona.
  - **alg_aead** es el Algoritmo AEAD, codificado como se define en [RFC8152].
  - **type** es "Key" o "IV". La etiqueta es una cadena ASCII y no incluye un byte NULL al final.
- **L** es el tamaño en bytes de la clave/nonce para el Algoritmo AEAD utilizado.

Por ejemplo, si se utiliza el algoritmo AES-CCM-16-64-128, el valor entero para alg_aead es 10, y el valor de L es 16 para las claves y 13 para el Common IV. Suponiendo el uso de los algoritmos predeterminados HKDF SHA-256 y AES-CCM-16-64-128, la fase de extracción del HKDF (HKDF-Extract) produce una clave pseudorandom (PRK) de la siguiente manera:

PRK = HMAC-SHA-256 (Master Salt, Master Secret)

Ecuación 4.2.- Obtención de la PRK en la fase de extracción del HKDF

Y como L es menor que el tamaño de salida de la función hash, la fase de expansión del HKDF (HKDF-Expand) consiste en una única invocación de HMAC; por lo tanto, la Sender Key, la Recipient Key y el Common IV son los primeros 16 o 13 bytes de:

Parámetro de salida = HMAC-SHA-256 (PRK, info || 0x01)

Ecuación 4.3.- Obtención del parámetro de salida en la fase de expansión del HKDF

Donde se usan diferentes valores de info para cada parámetro derivado y donde || denota la concatenación de cadenas de bytes. El valor 0x01 se utiliza para indicar que es la primera (y en este caso, única) invocación de HMAC en la fase de expansión y permite mantener la integridad y la consistencia en la derivación de las claves.

Si la Master Salt no se proporciona, se establece en una cadena de ceros. Para fines de implementación, no proporcionar la Master Salt es lo mismo que establecerla en la cadena de bytes vacía. OSCORE establece el valor predeterminado de la Master Salt en una cadena de bytes vacía, que se convierte en una cadena de ceros. [59, pp. 12-13]

4.6.5.- Campos de mensajes protegidos

Una vez establecido el contexto de seguridad, OSCORE puede proteger ciertos campos de CoAP. Para ello ofrece tres niveles de seguridad dependiendo de la categoría a la que pertenezcan esos campos:

- **Clase E:** Proporciona integridad y encriptación
- **Clase I:** Solo proporciona integridad.
- **Clase U:** No proporciona protección.

En el mensaje OSCORE, el transmisor necesita enviar los campos del mensaje de Clase E en el texto cifrado del objeto COSE. Si un campo de mensaje de Clase I se incluye en los Datos Autenticados Adicionales (AAD por sus siglas en inglés) del algoritmo AEAD, el receptor puede determinar si ha cambiado o no durante la transmisión. Los campos de mensaje de Clase U no necesitan estar protegidos durante la transferencia ya que no se requiere que estén cifrados. Todos los valores de los campos de mensaje de Clase I y de Clase U se envían en el encabezado del mensaje OSCORE o en la sección de opciones, que es accesible para los proxies. Lo que se llama "Inner fields" (campos internos) son aquellos que no son visibles para los proxies, es decir, los que se transportan en el texto cifrado del objeto COSE (Clase E). "Outer fields" (Clase I o Clase U) se refiere a los campos de mensaje transmitidos en el encabezado o en la sección de opciones del mensaje OSCORE, que es accesible a los proxies. Actualmente, no hay opciones de Clase I.

Un campo de mensaje de CoAP puede tener tanto una instancia "Inner" como una "Outer" en un mensaje OSCORE. Los campos de mensaje "Outer" se utilizan para permitir actividades de los proxies, mientras que los campos de mensaje "Inner" están destinados para el receptor. [59, p. 15]

4.6.6.- Generación de la asociación de seguridad OSCORE

Para generar el contexto de seguridad de OSCORE se necesita generar una Master Secret, una Master Salt, el Sender ID y el Recipient ID. Ejecutar OSCORE también requiere una negociación de cipher suite que se llevará a cabo en el intercambio del EAP Request Identity y del EAP Response Identity y que servirá para asegurar el mensaje EAP Success.

Para esta negociación de cipher suite hay que negociar el algoritmo AEAD y el algoritmo HASH que utilizará OSCORE. Para negociar la cipher suite, CoAP-EAP sigue un enfoque simple: el EAP Authenticator envía una lista, en orden decreciente de preferencia, con los identificadores de las cipher suites admitidas (paso 1 del flujo de operación). En la respuesta a ese mensaje (paso 2 del flujo de operación), el EAP Peer envía una respuesta con la cipher suite elegida. Esta lista se incluye en la carga útil después del mensaje EAP a través de una estructura CBOR como se explicó anteriormente.

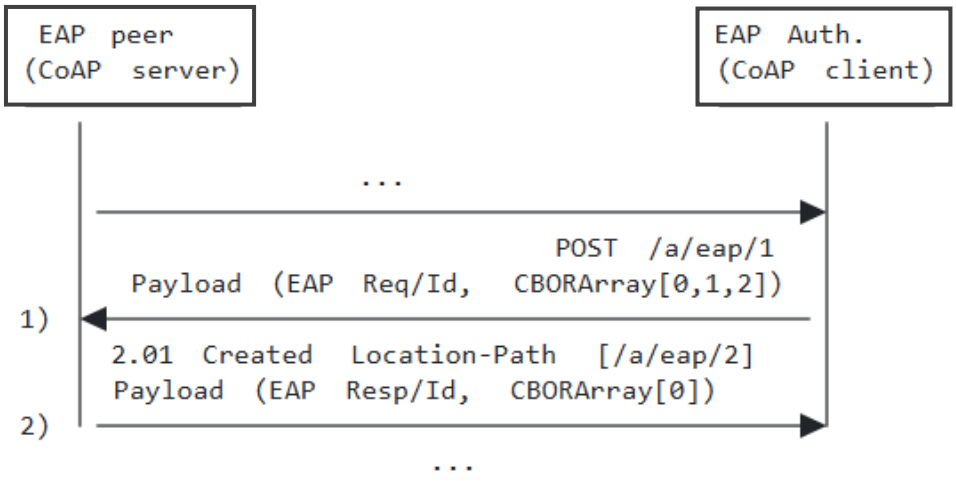


Figura 4.8.- Negociación de cipher suite

Entre las diferentes posibilidades, se tienen los siguientes pares de algoritmos AEAD y HASH:

0. AES-CCM-16-64-128, SHA-256 (predeterminado)
1. A128GCM, SHA-256
2. A256GCM, SHA-384
3. ChaCha20/Poly1305, SHA-256
4. ChaCha20/Poly1305, SHAKE256

Para generar el material clave necesario, se emplea una función de derivación de claves basada en HMAC conocida como HKDF. Este proceso utiliza el MSK (Master Session Key), que se considera material clave fresco, aplicando específicamente la fase de expansión de HKDF, denominada aquí como KDF para simplificar.

Con esta información, se puede generar la Master Secret y la Master Salt del contexto de seguridad de la siguiente manera:

Master Secret = KDF (MSK, CS | "COAP-EAP OSCORE Master Secret" , length)

Ecuación 4.4.- Obtención de la Master Secret

Master Salt = KDF (MSK, CS | "COAP-EAP OSCORE Master Salt", length)

Ecuación 4.5.- Obtención de la Master Salt

Donde:

- **MSK:** Es la Master Session Key que se exporta mediante el método EAP.
- **CS:** Es la concatenación del contenido de la negociación de cipher suite, es decir, el array CBOR que contiene la lista de cipher suites enviadas por el EAP Authenticator en el paso 1 (CS-C) concatenada al array CBOR que contiene la cipher suite seleccionada por el EAP Peer en el paso 2 (CS-I). Si CS-C o CS-I no se enviaron (es decir, se están utilizando los algoritmos predeterminados), el valor utilizado para generar CS será el mismo que si los algoritmos predeterminados se hubieran enviado explícitamente en CS-C o CS-I (es decir, un array CBOR con la cipher suite 0).

- **"COAP-EAP OSCORE Master Secret"** y **"COAP-EAP OSCORE Master Salt":** Son la representación del código ASCII de la cadena no terminada en NULL y excluyendo las comillas dobles que la rodean.
- **length:** Es el tamaño del material clave de salida.

Dado que la MSK se utiliza para derivar la Master Secret, la verificación correcta de la solicitud protegida con OSCORE (paso 7) y la respuesta (paso 8) confirma que el EAP Authenticator y el peer EAP tienen la misma Master Secret, logrando así la confirmación de claves.

Para prevenir un ataque de degradación, el contenido de la negociación de cipher suite (CS) se integra en la derivación de la Master Secret. Si un atacante modifica el valor de la negociación de la cipher suite, el resultado será la creación de contextos de seguridad OSCORE diferentes, lo que provocará un fallo en los pasos 7 y 8.

El EAP Authenticator utilizará el Recipient ID del EAP Peer (RID-I) como Sender ID para su Sender Context en OSCORE. El EAP Peer usará este valor como Recipient ID para su Recipient Context.

El EAP Peer utilizará el Recipient ID del EAP Authenticator (RID-C) como Sender ID para su Sender Context en OSCORE. El EAP Authenticator usará este valor como Recipient ID para su Recipient Context. [53, pp. 15-18]

De acuerdo con el estándar de OSCORE, cada extremo debe derivar los parámetros necesarios para establecer el contexto de seguridad basándose en un conjunto de parámetros de entrada. Los parámetros requeridos son: la Master Secret, el Sender ID y el Recipient ID. También hay otros parámetros que pueden ser proporcionados pero no son obligatorios, ya que si no se indican específicamente, la implementación utiliza valores predeterminados. Estos parámetros como ya se comentó en secciones anteriores son: el Algoritmo AEAD, que por defecto es AES-CCM-16-64-128; la Master Salt, que por defecto es la cadena de bytes vacía; el algoritmo HKDF, que por defecto es HKDF SHA-256 y, finalmente, la Replay Window, que por defecto es una ventana deslizante anti-repetición de 32 bytes. Aunque

todos los parámetros deben definirse para derivar el contexto de seguridad, la Replay Window puede ser diferente en cada extremo.

Lo que queda por definir son el Sender ID y el Recipient ID. Para obtener estos valores no es necesario llevar a cabo una negociación, por lo que se pueden elegir a voluntad siempre que sean diferentes.

Los posibles valores de los Sender IDs pueden variar desde valores muy cortos, incluidos los vacíos, hasta un tamaño máximo, en octetos, que es igual a la longitud del nonce AEAD menos 6. Por ejemplo, para AES-CCM-16-64-128, el tamaño máximo del Sender ID es de 7 bytes. En el caso del Recipient ID, el tamaño máximo permitido lo establece el Algoritmo AEAD. Por ejemplo, para AES-CCM-16-64-128, el tamaño máximo del Recipient ID será de 16 bytes.

La propuesta para generar el Sender ID y el Recipient ID es recurrir a la MSK para derivar un identificador que sea fresco y que pueda ser utilizado por ambas entidades. El identificador que resulta de esta operación se truncará de acuerdo con las limitaciones de OSCORE, dependiendo del Algoritmo AEAD utilizado.

Sender ID = HKDF (MSK, "OSCORE SENDER ID", length)

Ecuación 4.6.- Obtención del Sender ID

Recipient ID = HKDF (MSK, "OSCORE RECIPIENT ID", length)

Ecuación 4.7.- Obtención del Recipient ID

Donde los parámetros de entrada para el HKDF son:

- **MSK:** La Master Session Key derivada de la autenticación EAP.
- **"OSCORE SENDER ID"** y **"OSCORE RECIPIENT ID":** La cadena ASCII no terminada en NULL y sin incluir las comillas dobles.
- **length:** Es la longitud de la salida de la función.

Este método asegura que el Sender ID y el Recipient ID sean generados de manera segura y coherente para ambos extremos en la comunicación OSCORE. [57, pp. 39-40]

4.7.- EAP-PSK

EAP-PSK (Extensible Authentication Protocol - Pre-Shared Key) es un método de autenticación dentro del protocolo EAP que utiliza una clave precompartida (PSK por sus siglas en inglés) para autenticar a un usuario o dispositivo. Es uno de los métodos de autenticación más simples en términos de configuración, ya que requiere que ambas partes (cliente y servidor) compartan previamente una clave secreta.

4.7.1.- Terminología: Conceptos clave

En esta sección se van a definir algunos de los conceptos más importantes del protocolo EAP-PSK que van a ser nombrados en los próximos apartados [40, pp. 5-8]:

- **Identificador de acceso a la red (NAI):** Identificador utilizado para reconocer a las partes que intervienen en la comunicación.

- **Pre-Shared Key (PSK):** Una clave precompartida es simplemente una clave en criptografía simétrica. Esta clave se obtiene mediante algún mecanismo previo y se comparte entre las partes antes de que tenga lugar el protocolo que la utiliza. No es más que una secuencia de bits de una longitud determinada, cada uno de los cuales ha sido elegido al azar de manera uniforme e independiente. La PSK es la credencial a largo plazo de 16 bytes que se comparte entre el EAP Peer y el EAP Server. EAP-PSK supone que la PSK sólo la conocen el EAP Peer y el EAP Server. Las propiedades de seguridad del protocolo se ven comprometidas si tiene una distribución más amplia. EAP-PSK también asume que el EAP Server y el EAP Peer identifican la PSK correcta a utilizar el uno con el otro gracias a sus respectivos NAIs, esto significa que sólo debe haber como máximo una PSK compartida entre un EAP Server que utilice un NAI de server determinado y un EAP Peer que utilice un NAI de peer determinado. Esta PSK se utiliza para derivar dos subclaves estáticas de larga duración de 16 bytes, denominadas respectivamente Authentication Key

(AK) y Key-Derivation Key (KDK). Esta derivación sólo debe hacerse una vez, es lo que se denomina la configuración de la clave. PSK no se utiliza como una clave estática de larga duración, sino como el material clave inicial para derivar las claves estáticas de larga duración AK y KDK, que son realmente utilizadas por el protocolo EAP- PSK. [40, p. 13]

- **Authentication Key (AK):** EAP-PSK utiliza la AK para autenticar mutuamente al EAP Peer y al EAP Server. AK es una clave estática de larga duración de 16 bytes derivada de la PSK. AK no es una clave de sesión. El EAP Server y el EAP Peer identifican la AK correcta a utilizar el uno con el otro gracias a sus respectivos NAIs. Esto significa que sólo debe haber como máximo una AK compartida entre un EAP Server que utilice un NAI de server dado y un EAP Peer que utilice un NAI de peer dado. Este es el caso cuando hay como máximo una PSK compartida entre un EAP Server que utiliza un NAI de server dado y un EAP Peer que utiliza un NAI de peer dado. El EAP Peer elige la AK a utilizar basándose en el NAI del EAP Server, que ha sido enviado por el EAP Server en el campo ID_S del primer mensaje EAP-PSK y en el NAI del EAP Peer, que el EAP Peer elige incluir en el campo ID_P del segundo mensaje EAP-PSK. [40, p. 14]

- **Key-Derivation Key (KDK):** EAP-PSK utiliza la KDK para derivar claves de sesión compartidas por el EAP Peer y por el EAP Server (concretamente, la TEK, la MSK y la EMSK). La KDK no es una clave de sesión, es una clave estática de larga duración de 16 bytes derivada de la PSK.

- **Transient EAP Key (TEK):** EAP-PSK obtiene una clave de sesión TEK de 16 bytes gracias a un número aleatorio intercambiado durante la autenticación (RAND_P) y a la KDK. La TEK se utiliza para implementar un canal protegido (protected channel) entre el EAP Peer y el EAP Server durante el proceso de autenticación EAP para que se comuniquen de forma segura. La cipher suite utilizada para establecer el canal protegido entre el EAP Peer y el EAP Server durante la autenticación EAP no está relacionada con la utilizada posteriormente para proteger posteriormente los datos enviados entre el EAP Peer y el EAP Authenticator. La TEK de 16 bytes usada para el canal protegido es la única cipher suite disponible entre el EAP Peer y el EAP

Server para proteger la conversación EAP. Utiliza AES-128 en el modo de operación EAX.

- **Master Session Key (MSK):** Material clave derivado entre el EAP Peer y el EAP Server que es exportado por el método EAP. En las implementaciones existentes, un servidor AAA que actúa como EAP Server transporta la MSK al EAP-Authenticator. EAP-PSK obtiene una MSK gracias a un número aleatorio intercambiado durante la autenticación (RAND_P) y a la KDK. La MSK tiene una longitud de 64 bytes.

- **Extended Master Session Key (EMSK):** Material clave adicional derivado entre el EAP Peer y el EAP Server que es exportado por el método EAP. La EMSK se reserva para usos futuros que aún no están definidos y no se proporciona a terceros. La EMSK se obtiene gracias a un número aleatorio intercambiado durante la autenticación (RAND_P) y a la KDK. Tiene 64 bytes de longitud.

4.7.2.- Diseño criptográfico

EAP-PSK se basa en una única primitiva criptográfica, un cifrado por bloques, que se instancia con AES-128. AES-128 toma como entradas una clave precompartida de 16 bytes y un bloque de texto plano (plain text) de 16 bytes. Su salida es un bloque de texto cifrado (cipher text) de 16 bytes. Se ha elegido AES-128 porque está estandarizado, sus implementaciones están ampliamente disponibles, ha sido cuidadosamente revisado por la comunidad criptográfica y se cree que es seguro. EAP-PSK utiliza tres partes criptográficas [40, pp. 15-16]:

- Una configuración de claves para derivar la AK y la KDK de la PSK.

- Un protocolo de intercambio de claves que permite autenticar mutuamente a las partes que intervienen en la comunicación y derivar las claves de sesión (TEK, MSK y EMSK).

- Un protocolo que establece un canal protegido para la comunicación entre las dos partes que ya han sido autenticadas mutuamente.

4.7.3.- Configuración de claves

EAP-PSK requiere dos subclaves de 16 bytes, criptográficamente independientes, para la autenticación mutua y la derivación de claves de sesión. De hecho, es una regla general en criptografía utilizar claves diferentes para aplicaciones diferentes. Se podría haber implementado estas dos subclaves especificando una PSK de 32 bytes que luego se dividiría en dos subclaves de 16 bytes, o especificando una PSK de 16 bytes que luego se expandiría criptográficamente a dos subclaves de 16 bytes. Dado que proporcionar una credencial a largo plazo de 32 bytes es más engorroso que una de 16 bytes y que la longitud de las claves de sesión derivadas es de 16 bytes, se eligió esta última opción.

Por lo tanto, la PSK sólo es utilizada por EAP-PSK para derivar la AK y la KDK. Esta derivación sólo debe realizarse una vez, inmediatamente después de que se haya proporcionado la PSK. Tan pronto como se hayan derivado la AK y la KDK, se debe eliminar la PSK. Si se borra la PSK, debe hacerse de forma segura. La derivación de la AK y la KDK a partir del PSK se denomina configuración de claves:

- La entrada de la configuración de claves es la PSK.

- Las salidas de la configuración de claves son la AK y la KDK.

La AK y la KDK se obtienen a partir de la PSK utilizando el modo de funcionamiento de contador modificado de AES-128. El modo de contador modificado (modified counter mode) se trata de una función que incrementa la longitud de la clave, es decir, expande un bloque de entrada AES-128 en una salida más larga compuesta por t bloques, donde $t \geq 2$. Este modo fue seleccionado para la configuración de claves porque ya se había elegido previamente para la derivación de las claves de sesión.

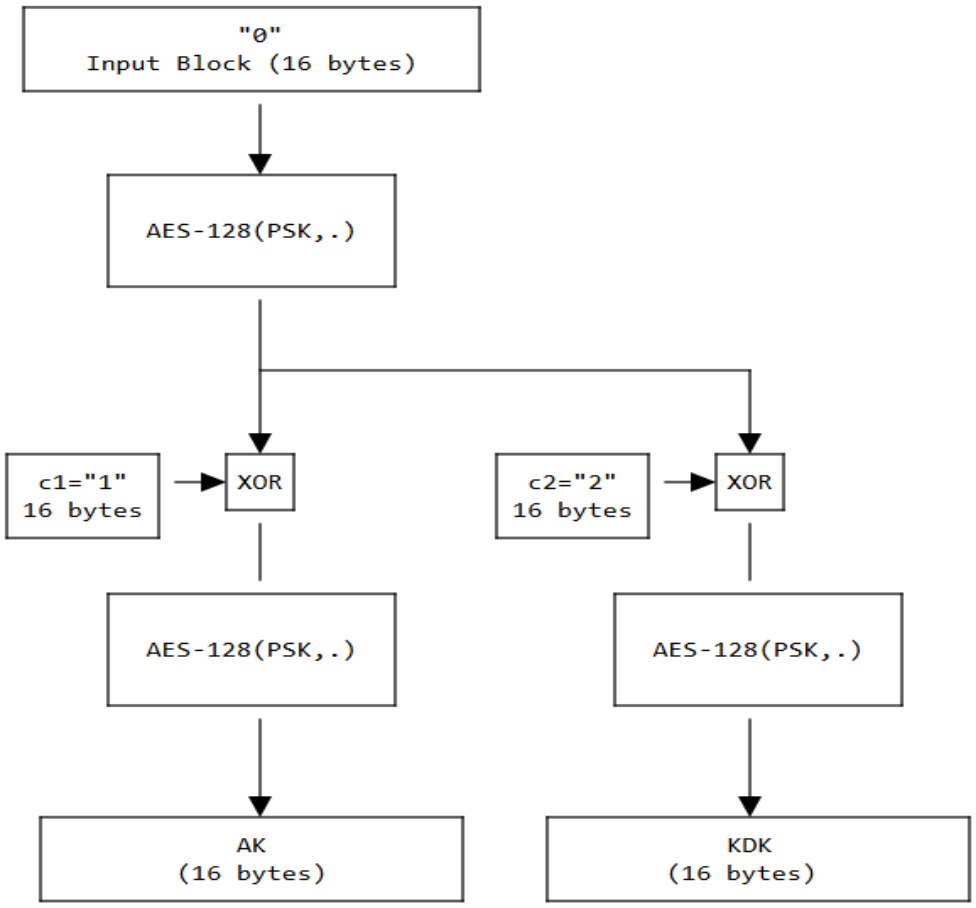


Figura 4.9.- Derivación detallada de la AK y de la KDK a partir de la PSK

El bloque de entrada es "0". En aras de la simplicidad, este bloque de entrada se ha elegido constante. Podría haberse establecido en un valor que dependiera del EAP Peer y del EAP Server, por ejemplo, el XOR de sus respectivos NAI convenientemente truncado o relleno de ceros, pero esto no parecía añadir mucha seguridad al esquema, mientras que añadía complejidad. Se podría haber elegido cualquier constante de 16 bytes ya que se supone que la seguridad no depende del valor concreto que tome la constante. El "0" se eligió arbitrariamente. [40, pp. 16-18]

4.7.4.- Intercambio de claves

El protocolo de autenticación utilizado por EAP-PSK se inspira en AKEP2, el cual consiste en un intercambio de ida y vuelta que consta de tres mensajes:

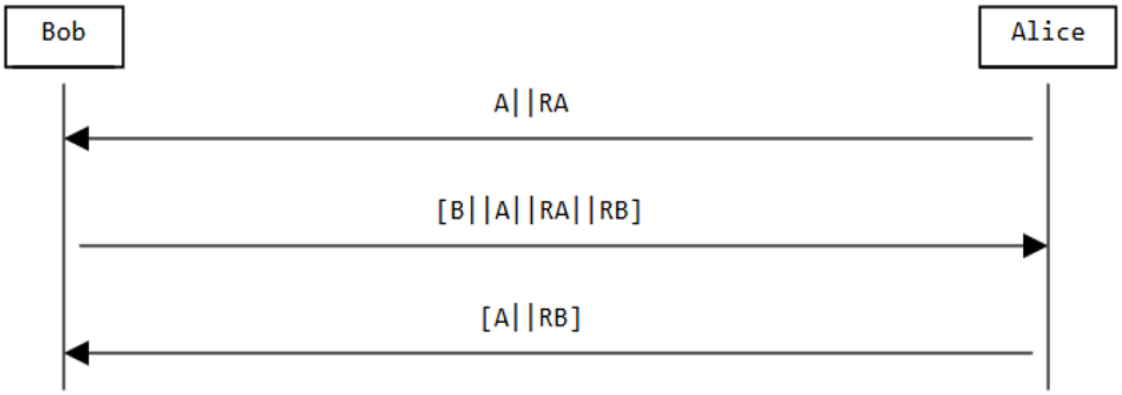


Figura 4.10.- Visión general de AKEP2

En AKEP2:

- RA y RB son números aleatorios elegidos respectivamente por Alice y Bob.
- A y B son los NAIs respectivos de Alice y Bob. Permiten a Alice y Bob recuperar la clave que tienen que utilizar para realizar un intercambio de claves autenticado entre ellos. Se incluyen en el protocolo por razones criptográficas.
- Las MACs se calculan usando una clave dedicada.

EAP-PSK instancia este protocolo con:

- El EAP Server como Alice y el EAP Peer como Bob.
- RA y RB como números aleatorios de 16 bytes. Esto significa RA=RAND_S y RB=RAND_P.
- A y B como los respectivos NAIs de Alice y Bob. Esto significa A=ID_S y B=ID_P.
- El algoritmo MAC es CMAC con AES-128 utilizando la AK y produciendo una longitud de etiqueta (tag) de 16 bytes.
- El modo de operación de contador modificado de AES-128 utilizando la KDK para derivar las claves de sesión como resultado de este intercambio.

Se eligió CMAC como algoritmo MAC porque es capaz de manejar mensajes de longitud arbitraria y su diseño es sencillo. También goza de una revisión actualizada por parte de la comunidad criptográfica, especialmente en lo que se refiere al uso de conceptos de seguridad. En AKEP2, el intercambio de claves es "implícito", esto quiere decir que las

claves de sesión se derivan de RB. En EAP-PSK, las claves de sesión se derivan de RAND_P utilizando la KDK y el modo de operación de contador modificado de AES-128. Este modo, como ya se comentó anteriormente, es una función que aumenta la longitud, es decir, expande un bloque de entrada AES-128 en una salida más larga compuesta por t bloques, donde $t \geq 2$.

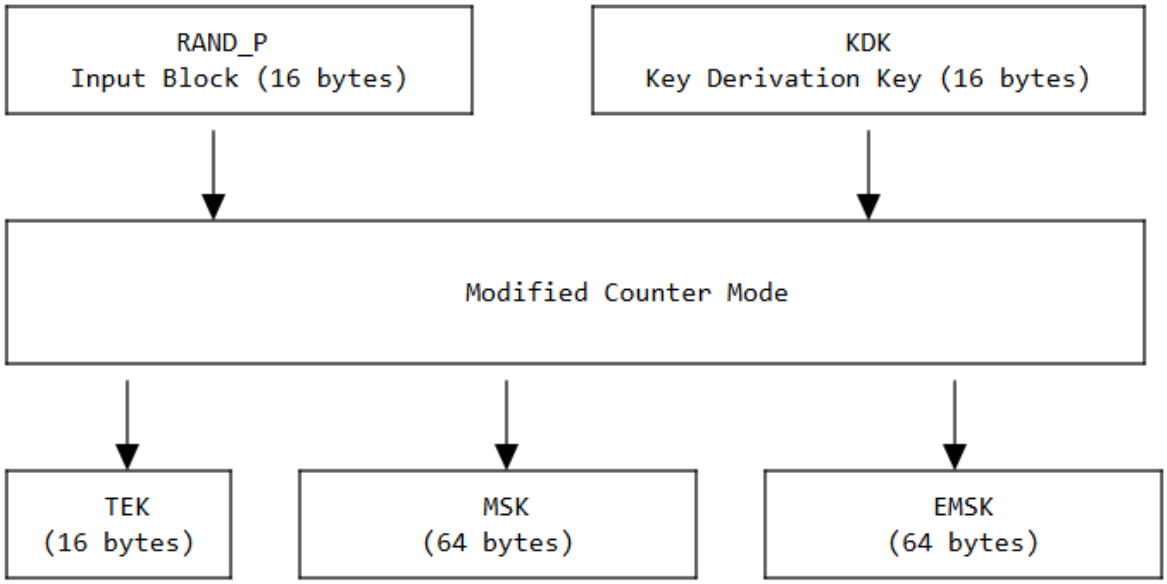


Figura 4.11.- Esquema de la derivación de las claves de sesión

La entrada para la derivación de las claves de sesión es RAND_P. Las salidas de la derivación de las claves de sesión son:

- La TEK de 16 bytes (el primer bloque de salida).
- La MSK de 64 bytes (la concatenación del segundo al quinto bloque de salida).
- La EMSK de 64 bytes (la concatenación del sexto al noveno bloque de salida).

Un esquema detallado de cómo se realiza la derivación de las claves de sesión se muestra en la siguiente figura:

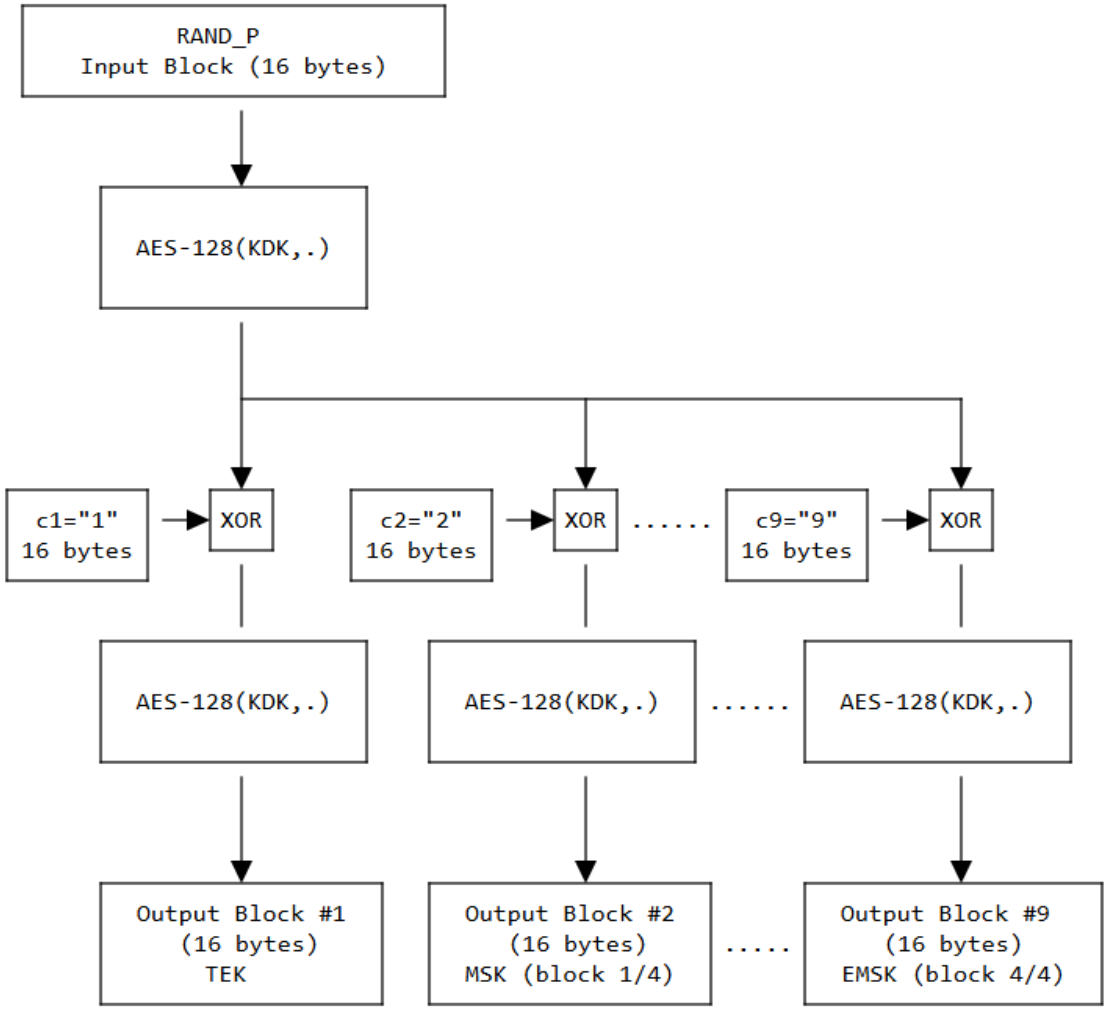


Figura 4.12.- Derivación detallada de las claves de sesión

Los valores del contador se establecen respectivamente en los primeros t enteros (es decir, ci="i", con i=1 a 9). El material clave es información sensible y debe tratarse en consecuencia. [40, pp. 19-22]

4.7.5.- Canal protegido

EAP-PSK proporciona un canal protegido para que ambas partes se comuniquen en caso de una autenticación exitosa. Este canal se utiliza actualmente para intercambiar indicaciones de resultado protegidas y puede utilizarse en el futuro para implementar extensiones. EAP-PSK utiliza el modo de funcionamiento EAX para proporcionarlo.

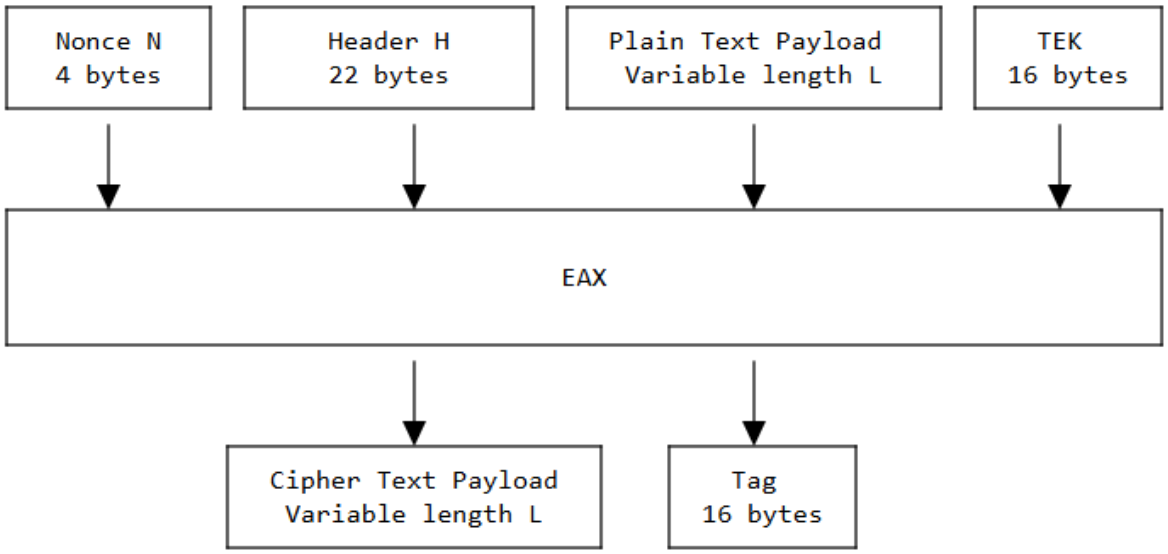


Figura 4.13.- Esquema de la obtención del canal protegido

Este canal protegido:

- Proporciona protección contra repeticiones.
- Cifra y autentica una carga útil en texto plano que se convierte en una carga útil cifrada. La carga útil (payload) en texto plano no debe superar los 960 bytes.
- Sólo autentica una cabecera (header) que se envía en texto plano.

EAX se instanciará con AES-128 utilizando la TEK como clave. EAX fue elegido principalmente porque su diseño se basa fuertemente en OMAC y porque OMAC1, una variante de OMAC, ya había sido elegida en EAP-PSK para la parte de autenticación (cabe mencionar que OMAC1 y CMAC con análogos). Además, su diseño es sencillo, cuenta con una prueba de seguridad y está libre de cualquier reclamación de derechos de propiedad intelectual.

El nonce N se utiliza para proporcionar seguridad criptográfica al cifrado y a la autenticación del origen de los datos, así como para proteger contra la reproducción. De hecho, N es un número de secuencia de 4 bytes que comienza en 0 y que se incrementa monótonamente en cada mensaje dentro de un diálogo EAP-PSK, excepto en las retransmisiones. Se eligió que N fuera de 4 bytes para evitar la aritmética de 16 bytes. Dado

que EAX utiliza un nonce de 16 bytes, N se rellena con 96 bits a cero para sus bits de orden superior. N no puede volver a su valor inicial después de alcanzar su límite. En otras palabras, N debe incrementarse de manera continua sin volver a cero o reiniciarse en algún punto. Esto se hace para evitar problemas de seguridad relacionados con la reutilización de valores de nonce, lo que podría comprometer la criptografía. El número máximo de mensajes que se pueden intercambiar a través del mismo canal protegido es de 2^{32} (lo que no debería suponer una limitación en la práctica, ya que equivale aproximadamente a 4.000 millones de mensajes).

El Header H consiste en los primeros 22 bytes del paquete de solicitud o respuesta EAP (es decir, los campos EAP Code, Identifier, Length y Type, seguidos del campo EAP-PSK Flags y del campo RAND_S).

El payload en texto plano es la carga útil que debe ser cifrada y protegida con integridad y el payload cifrado es el resultado de cifrar el payload de texto plano.

El Tag es una MAC que protege tanto el Header como el payload en texto plano. La verificación del Tag debe realizarse solo después de una verificación exitosa del Nonce para la protección contra posibles repeticiones. Si la verificación del Tag tiene éxito, entonces el payload cifrado se descifra para recuperar el payload en texto plano. Si la verificación del Tag falla, no se descifra el payload y se debe registrar el fallo de MAC. La longitud del Tag se ha establecido en 16 bytes para EAX dentro de EAP-PSK. [40, pp. 23-25]

4.7.6.- Flujo de EAP-PSK

EAP-PSK introduce el concepto de sesión para facilitar su análisis y proporcionar una interfaz más limpia a otras capas. Una sesión es una instancia particular de un diálogo EAP-PSK entre dos partes. Cada sesión se identifica mediante un identificador de sesión.

En el primer mensaje EAP-PSK, el EAP Server confirma su identidad. Dado que no se puede suponer que el EAP Request/Identity y el EAP Response/Identity se hayan producido antes de este envío y que la respuesta incluida en el EAP Response/Identity puede no contener el NAI real que el peer utilizará con EAP-PSK, un EAP Server que implemente

EAP-PSK debe utilizar el mismo NAI para todos los diálogos EAP-PSK con cualquier EAP Peer que implemente EAP-PSK.

La autenticación estándar EAP-PSK se comprime en 4 mensajes:

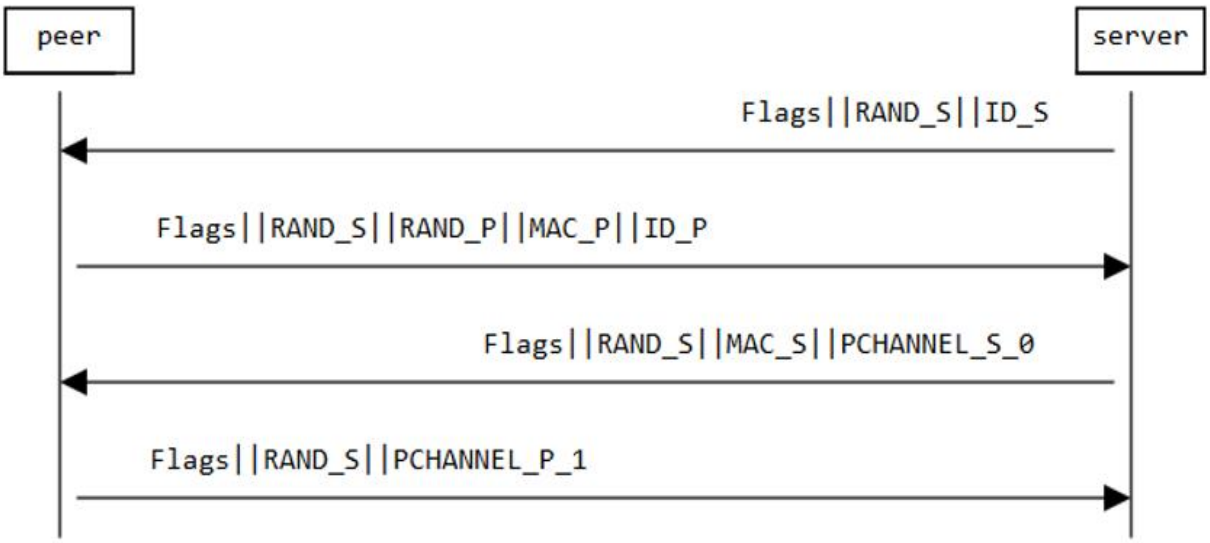


Figura 4.14.- Esquema de la autenticación estándar EAP-PSK

- El primer mensaje es enviado por el server al peer para enviar un desafío aleatorio de 16 bytes (RAND_S) y para indicar su identidad (ID_S).

- El segundo mensaje es enviado por el peer al server para enviar otro desafío aleatorio de 16 bytes (RAND_P), indicar su identidad (ID_P) y autenticarse ante el server demostrando que es capaz de calcular una MAC particular (MAC_P).

$$\mathbf{MAC\_P = CMAC\text{-}AES\text{-}128\ (AK,\ ID\_P||ID\_S||RAND\_S||RAND\_P)}$$

Ecuación 4.8.- Obtención de la MAC_P

- El tercer mensaje lo envía el server al peer para autenticar al peer demostrando que es capaz de calcular otra MAC (MAC_S).

$$\mathbf{MAC\_S = CMAC\text{-}AES\text{-}128\ (AK,\ ID\_S||RAND\_P)}$$

Ecuación 4.9.- Obtención de la MAC_S

Además, configura el canal protegido (P_CHANNEL_S_0) para que:

- ✓ Confirme que ha derivado las claves de sesión (al menos la TEK).
- ✓ Proporcione una indicación del resultado protegido de la autenticación.

- El cuarto mensaje es enviado por el peer al servidor para finalizar la configuración del canal protegido (P_CHANNEL_P_1) para que:

  - ✓ Confirme que ha derivado las claves de sesión (al menos la TEK).
  - ✓ Proporcione una indicación del resultado protegido de la autenticación.

Los campos PCHANNEL_S_0 y PCHANNEL_P_1 del tercer y del cuarto mensaje EAP- PSK contienen una MAC calculada gracias a TEK que protege la integridad de los mensajes. Todos los mensajes EAP-PSK incluyen una especie de cabecera que se compone de dos campos:

- **Flag:** Un campo de 1 byte que actualmente sólo se utiliza para numerar los mensajes EAP- PSK.
- **RAND_S:** Un desafío de 16 bytes enviado por el servidor que se utiliza como identificador de sesión.

Este flujo de mensajes estándar podría constar de sólo tres mensajes, como AKEP2, si no fuera porque la naturaleza solicitud/respuesta de EAP impide que el tercer mensaje sea el último. Dado que el cuarto mensaje es obligatorio, EAP-PSK optó por aprovecharse de ello y establecer un canal protegido.

El flujo de mensajes estándar también incluye una indicación por parte del EAP Peer de su identidad, además del EAP Response Identity que puede haber sido enviado. Se recomienda que el EAP Response Identity sea usado principalmente para propósitos de enrutamiento y para seleccionar qué método EAP usar, y por tanto que los métodos EAP incluyan un mecanismo específico para obtener la identidad, de forma que no tengan que depender del Response Identity.

Cuando una parte recibe un mensaje EAP-PSK, comprueba que el mensaje es sintácticamente válido de acuerdo con los formatos de mensaje. Si el mensaje es sintácticamente incorrecto, se descarta. La validez criptográfica de cada mensaje (es decir, la verificación de la/las MAC(s)) se comprueba de la siguiente manera:

- Si el mensaje recibido es el primer mensaje EAP-PSK, no hay MAC que comprobar ya que no se incluye ninguna.
- Si el mensaje recibido es el segundo mensaje EAP-PSK, se comprueba la validez de la MAC_P.
- Si el mensaje recibido es el tercer mensaje EAP-PSK, se comprueba la validez de la MAC_S y, a continuación, se comprueba la validez del Tag incluido en el P_CHANNEL_S_0. Las comprobaciones de validez deben realizarse en este orden para evitar derivar innecesariamente la TEK, la MSK y la EMSK en caso de que la MAC_S no sea válida, lo que significa que la autenticación mutua ha fallado. De hecho, la TEK se utiliza para verificar la validez del Tag incluido en el P_CHANNEL_S_0.
- Si el mensaje recibido es el cuarto mensaje EAP-PSK, se comprueba la validez del Tag incluido en el P_CHANNEL_P_1.

Si la comprobación de validez falla, el mensaje se descarta. Puede haber un contador para rastrear el número de mensajes descartados. Si hay un payload cifrado en el atributo PCHANNEL, entonces el payload cifrado se descifra. Si el payload descifrado es sintácticamente incorrecto, el mensaje se descarta silenciosamente, es decir, el sistema simplemente ignora o elimina el mensaje sin generar ningún tipo de notificación o advertencia visible. [40, pp. 25-28]

4.7.7.- Formato de los mensajes EAP-PSK

Para simplificar, EAP-PSK utiliza un formato de mensaje fijo. Hay cuatro tipos diferentes de mensajes EAP-PSK [40, p. 31]:

- El primer mensaje EAP-PSK, que envía el EAP Server al EAP Peer.
- El segundo mensaje EAP-PSK, que envía el EAP Peer al EAP Server.

- El tercer mensaje EAP-PSK, que envía el EAP Server al EAP Peer.
- El cuarto mensaje EAP-PSK, que envía el EAP Peer al EAP Server. Este es también esencialmente el tipo de mensaje que el EAP Server envía al EAP Peer en caso de una autenticación extendida. La única ligera modificación que se produce en este último caso es que el EAP Code se configura a 1 en lugar de a 2 como en los otros casos.

PRIMER MENSAJE

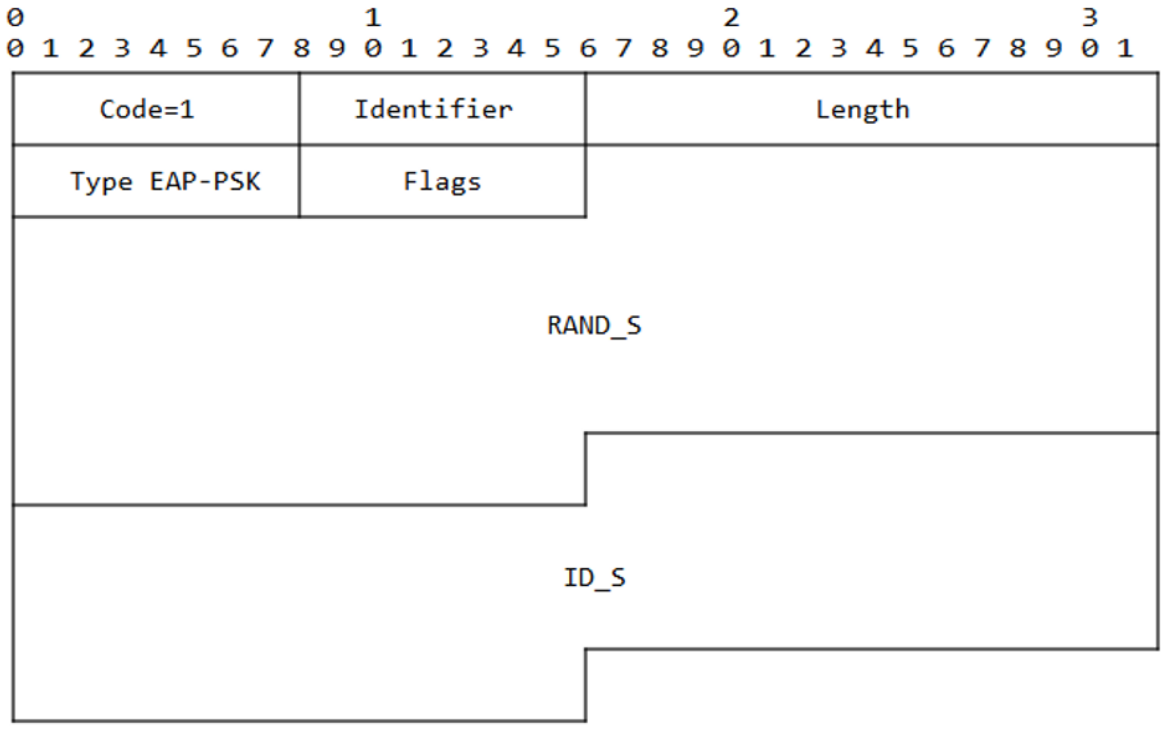


Figura 4.15.- Primer mensaje EAP-PSK

Dado que IANA ha asignado el tipo de método EAP (EAP method type) 47 para EAP-PSK, el campo Type para el primer mensaje, así como para el resto de los mensajes EAP-PSK, debe ser 47.

El primer mensaje EAP-PSK consta de:

- Un campo **Flags** de 1 byte.
- Un desafío aleatorio de 16 bytes: **RAND_S**.

- Un campo de longitud variable que contiene el NAI del EAP Server: **ID_S**. La longitud de este campo se deduce del campo Length de EAP. La longitud de este NAI no debe superar los 966 bytes. Esta restricción pretende evitar problemas de fragmentación.

El campo Flags se compone de dos subcampos:

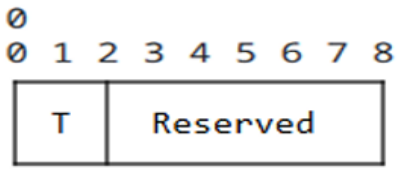


Figura 4.16.- Campo Flags de un mensaje EAP-PSK

- Un subcampo **T** de 2 bits, que indica el tipo de mensaje EAP-PSK:

  - T=0 para el primer mensaje EAP-PSK.
  - T=1 para el segundo mensaje EAP-PSK.
  - T=2 para el tercer mensaje EAP-PSK.
  - T=3 para el cuarto mensaje EAP-PSK y los siguientes mensajes EAP-PSK que puedan ser intercambiados durante la autenticación extendida.

- Un subcampo **Reserved** de 6 bits que se pone a cero en la transmisión y se ignora en la recepción.

El campo Nonce (N) del PCHANNEL se utiliza para distinguir entre los distintos mensajes EAP-PSK que pueden intercambiarse durante la autenticación extendida, los cuales tienen el subcampo T establecido en 3. [40, pp. 32-33]

SEGUNDO MENSAJE

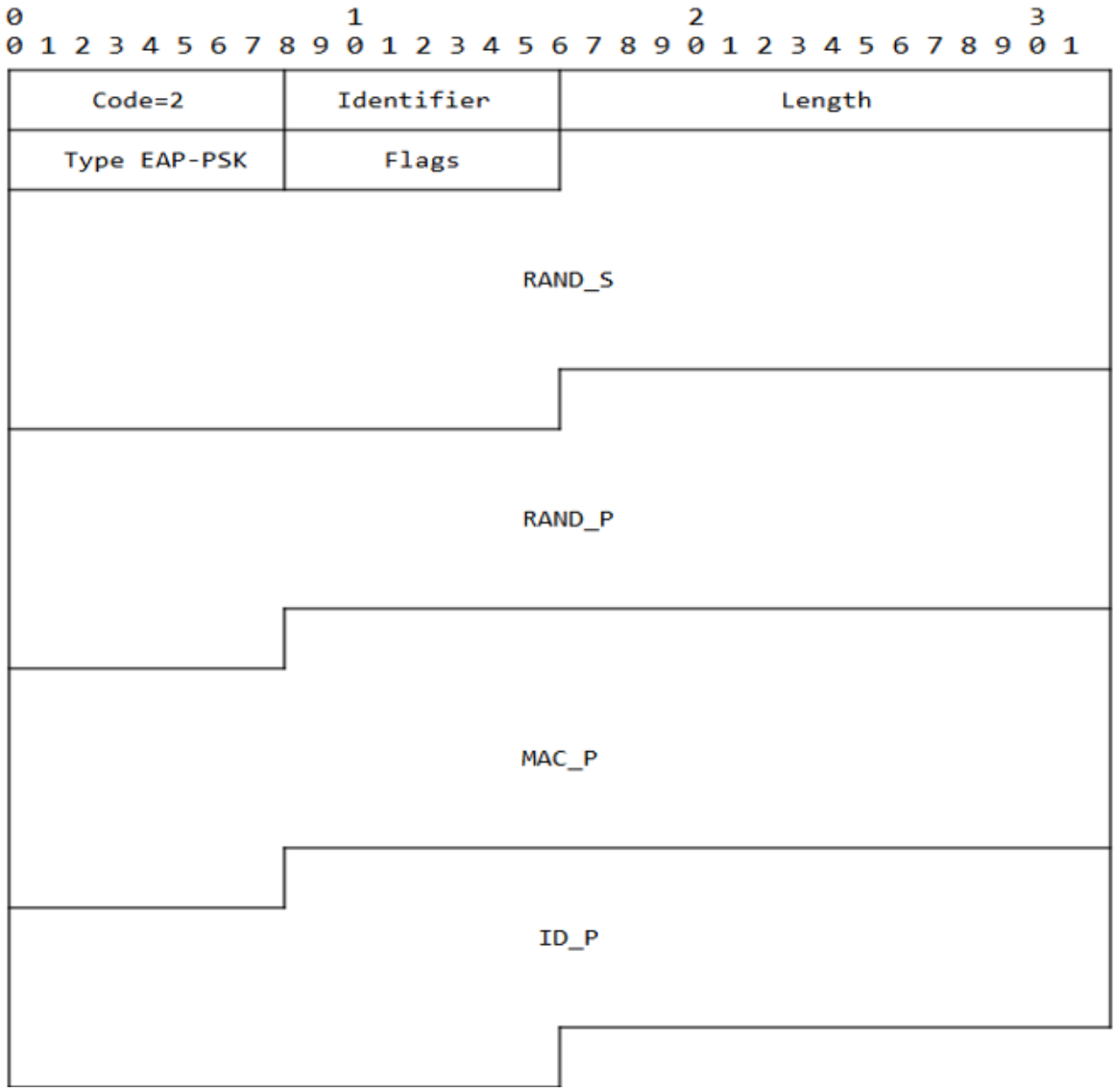


Figura 4.17.- Segundo mensaje EAP-PSK

El segundo mensaje EAP-PSK consta de [40, pp. 34-35]:

- Un campo **Flags** de 1 byte.
- El desafío aleatorio de 16 bytes enviado por el EAP Server en el primer mensaje EAP-PSK (**RAND_S**) que sirve como identificador de sesión.
- Un desafío aleatorio de 16 bytes: **RAND_P**.
- Una MAC de 16 bytes: **MAC_P**.

- Un campo de longitud variable que transmite el NAI del EAP Peer: **ID_P**. La longitud de este campo se deduce del campo Length de EAP. La longitud de este NAI no debe superar los 966 bytes. Esta restricción tiene como objetivo evitar problemas de fragmentación.

TERCER MENSAJE

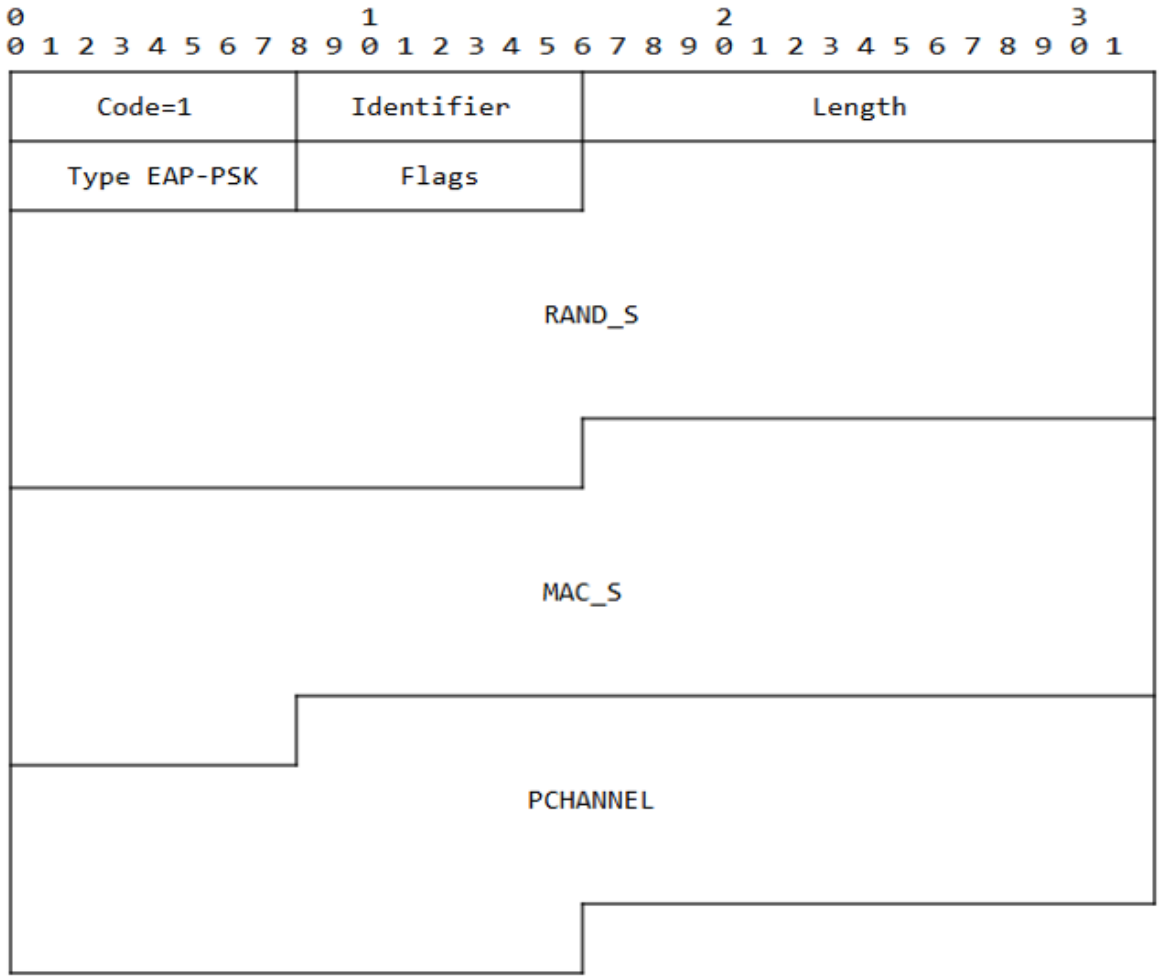


Figura 4.18.- Tercer mensaje EAP-PSK

El tercer mensaje EAP-PSK consta de:

- Un campo **Flags** de 1 byte.
- El desafío aleatorio de 16 bytes enviado por el EAP Server en el primer mensaje EAP-PSK (**RAND_S**) que se utiliza como identificador de sesión.

- Una MAC de 16 bytes: **MAC_S**.
- Un campo de longitud variable que constituye el canal protegido: **PCHANNEL**.

Si no hay extensión, es decir, si la autenticación es estándar, el campo PCHANNEL consta de:

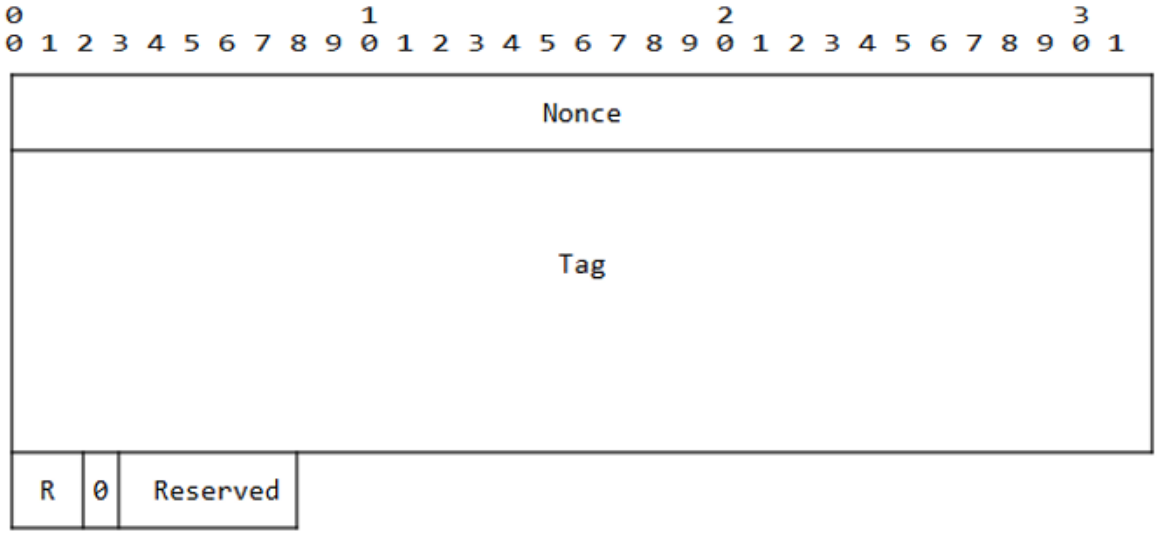


Figura 4.19.- Campo PCHANNEL del tercer mensaje EAP-PSK

- Un Nonce N de 4 bytes.
- Un Tag de 16 bytes.
- Un indicador de resultado R de 2 bits.
- Un indicador de extensión E de 1 bit, que se pone a 0.
- Un campo Reserved de 5 bits, que se pone a cero en la emisión y se ignora en la recepción.

R, E y Reserved se envían encriptados por el canal protegido. [40, pp. 36-39]

Si no hay extensión, el PCHANNEL tiene el formato presentado en la figura 4.19, donde R, E y Reserved se presentan en texto plano para mayor claridad, aunque en realidad se envían cifrados.

CUARTO MENSAJE

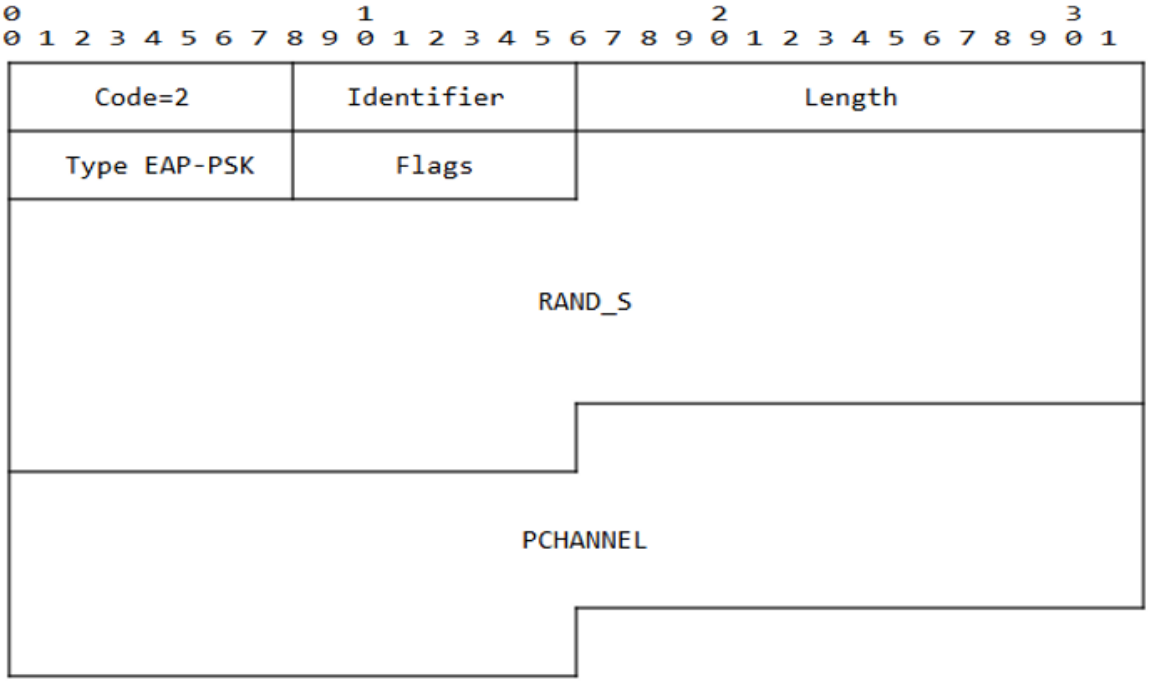


Figura 4.20.- Cuarto mensaje EAP-PSK

El cuarto mensaje de EAP-PSK consta de:

- Un campo **Flags** de 1 byte.
- El desafío aleatorio de 16 bytes enviado por el EAP Server en el primer mensaje EAP-PSK (**RAND_S**) que se utiliza como identificador de sesión.
- Un campo de longitud variable que constituye el canal protegido: **PCHANNEL**.

El campo PCHANNEL tiene la misma estructura que la que ya se describió en el mensaje anterior. [40, pp. 39-40]

4.7.8.- Indicadores de resultado protegido

El flag R del campo PCHANNEL del tercer y del cuarto mensaje EAP-PSK se utiliza para proporcionar indicaciones de resultado. Dado que este indicador de 2 bits se comunica a través del canal protegido, estará cifrado para que sólo el EAP Peer y el EAP Server puedan

conocer su valor, protegido contra integridad para que no pueda ser modificado por un atacante sin que el EAP Peer o el EAP Server detecten esta modificación y protegido contra repeticiones. [40, pp. 41-42]

Este flag R de 2 bits puede tomar los siguientes valores:

- **01** para significar **CONT**: CONT significa "Continuar". Indica que el diálogo EAP-PSK aún no ha tenido éxito y que la parte que lo envía desea continuar el diálogo para intentar alcanzarlo. [40, p. 42]

- **10** para significar **DONE_SUCCESS**: DONE_SUCCESS indica que la parte que lo envió considera que el diálogo EAP-PSK ha tenido éxito y, por lo tanto, propone finalizar este diálogo. El EAP Peer debe recibir primero un DONE_SUCCESS del EAP Server antes de que se le permita enviar otro DONE_SUCCESS. Después de que el EAP Peer haya recibido un DONE_SUCCESS del EAP Server, puede [40, p. 43]:

    - Enviar un CONT al EAP Server si no ha alcanzado el éxito por su parte. El EAP Server que recibe el CONT debe continuar el diálogo EAP-PSK.
    - Enviar un DONE_SUCCESS al EAP Server, el cual finalizará el diálogo EAP-PSK con éxito.
    - Enviar un DONE_FAILURE al EAP Server, el cual finalizará el diálogo EAP-PSK con fallo.

- **11** para significar **DONE_FAILURE**: DONE_FAILURE indica que la parte que lo ha enviado considera que el diálogo EAP-PSK no ha tenido éxito y propone terminar este diálogo porque nada le hará cambiar de opinión. Si el EAP Server es el primero en enviar un DONE_FAILURE, entonces el EAP Peer que recibe este DONE_FAILURE debe responder con otro DONE_FAILURE y fallar, lo que pone fin al diálogo EAP-PSK. Si el EAP Peer es el primero en enviar un DONE_FAILURE, entonces el EAP Server que recibe este DONE_FAILURE debe terminar inmediatamente este diálogo EAP-PSK sin enviar ningún otro mensaje EAP-PSK y fallar. [40, p. 43]

Tanto el EAP Peer como el EAP Server recuerdan cierta información sobre los valores de R que han enviado y los valores de R que han recibido. Es la combinación de ambos valores de R enviados y recibidos lo que indica el éxito o el fracaso del diálogo EAP-PSK. En el caso de una autenticación estándar, se deben intercambiar los siguientes valores de R:

- O bien el EAP Server envía un DONE_SUCCESS en el PCHANNEL del tercer mensaje EAP-PSK, al que el EAP Peer responde con un DONE_SUCCESS en el PCHANNEL del cuarto mensaje EAP-PSK, finalizando con éxito el diálogo EAP-PSK.
- O el EAP Server envía un DONE_FAILURE en el PCHANNEL del tercer mensaje EAP-PSK, al que el EAP Peer responde con un DONE_FAILURE en el PCHANNEL del cuarto mensaje EAP-PSK, finalizando sin éxito el diálogo EAP-PSK.

En el caso de una autenticación extendida, pueden producirse intercambios más complejos, razón por la cual se introdujo el valor CONT.

4.8.- RADIUS

4.8.1.- Qué es y características

RADIUS (Remote Authentication Dial-In User Service) es un protocolo de red utilizado para el acceso remoto, creado con el propósito concreto de supervisar y asegurar el acceso a recursos de computación heterogéneos. Se trata de un protocolo no orientado a conexión, que usa el puerto UDP 1812 para la autenticación RADIUS y que, al utilizar UDP como protocolo de transporte, no emplea conexiones directas persistentes entre el cliente y el servidor. [60] Esto significa que cada solicitud que el cliente envía al servidor es un paquete independiente, al igual que la respuesta correspondiente del servidor. Es decir, no se mantiene un estado continuo entre ellos ya que cada intercambio de mensajes es autónomo.

Fue desarrollado originalmente por Livingston Enterprises, luego fue adquirido por Ascend Communications y más tarde fue estandarizado por el IETF (Internet Engineering Task Force). [61]

Proporciona servicios de Authentication, Authorization y Accounting (AAA) para dispositivos de red, como servidores de acceso remoto, puntos de acceso inalámbricos y conmutadores de red.

- **Authentication:** RADIUS se utiliza principalmente para autenticar usuarios que intentan acceder a una red. Cuando un usuario intenta iniciar sesión, el dispositivo de red envía una solicitud de autenticación al servidor RADIUS, el cual intentará validar las credenciales del usuario utilizando una variedad de métodos de autenticación, como contraseñas almacenadas localmente, bases de datos externas (como LDAP o Active Directory), tokens de seguridad u otros mecanismos de autenticación.

- **Authorization:** Una vez que el usuario ha sido autenticado con éxito, el servidor RADIUS puede proporcionar información de autorización al dispositivo de red, especificando los servicios y recursos a los que el usuario tiene acceso. Esto puede incluir políticas de acceso basadas en roles, restricciones de ancho de banda, asignación de direcciones IP o configuraciones de VLAN, entre otros.

- **Accounting:** RADIUS también se utiliza para llevar un registro de las actividades de los usuarios en la red, lo que se conoce como contabilidad. Esto incluye información como la hora de inicio y de finalización de la sesión, la cantidad de datos transferidos, las direcciones IP asignadas, etc. Estos registros de contabilidad pueden ser utilizados para propósitos de facturación, auditoría, monitoreo de la red y cumplimiento de políticas.

RADIUS ofrece más de 50 atributos estandarizados para gestionar los servicios AAA, como User-Name, NAS-IP-Address, Framed-IP-Address y Session-Timeout. Estos atributos permiten configurar aspectos clave como las credenciales del usuario, los detalles de la red y la duración de las sesiones. Además, soporta atributos específicos de proveedores

(Vendor-Specific Attributes, VSAs), que permiten a fabricantes como Cisco o Huawei extender la funcionalidad del protocolo según sus necesidades.

RADIUS proporciona un mecanismo seguro para la autenticación de los usuarios y la transmisión de datos entre el dispositivo de red y el servidor RADIUS. Emplea un modelo de seguridad cliente-servidor que protege la comunicación entre el cliente y el servidor RADIUS mediante una clave compartida para autenticación mutua. Sin embargo, este modelo de seguridad tiene alguna limitación ya que solo cifra la contraseña del usuario empleando MD5, dejando otros atributos como el nombre de usuario en texto claro, lo que puede suponer que información sensible quede expuesta en alguna ocasión. Además, la seguridad no es extremo a extremo, ya que los clientes RADIUS deben ser confiables para transmitir las solicitudes de autenticación.

4.8.2.- Formato de los paquetes

RADIUS utiliza paquetes UDP debido, entre otros motivos, a su naturaleza stateless, lo que lo hace adecuado para su funcionamiento. Este protocolo se comunica a través del puerto 1812, que reemplazó al puerto 1645 especificado originalmente en el RFC para evitar interferencias con el servicio Datametrics. [61]

Los paquetes RADIUS se componen principalmente de dos partes:

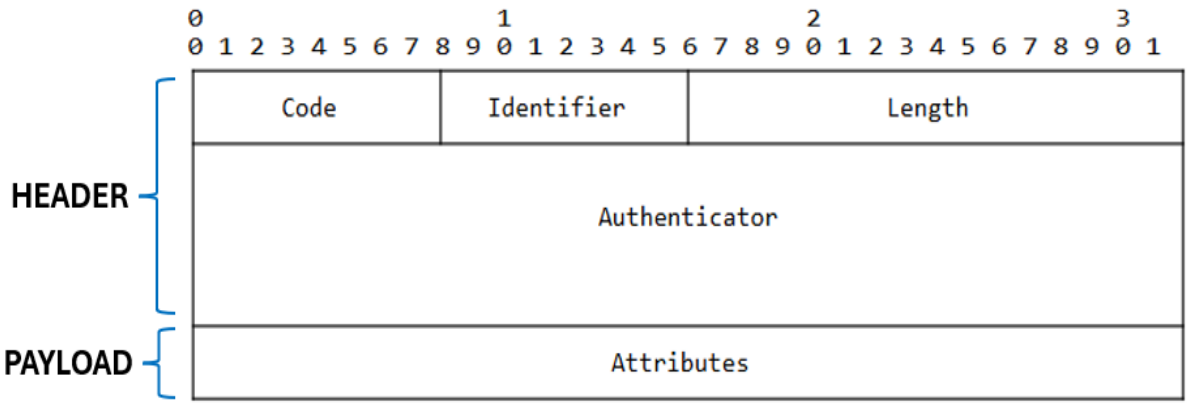


Figura 4.21.- Formato de los paquetes RADIUS

La primera parte es una **cabecera (header)** cuyos campos son los siguientes [62, pp. 13-16]:

- Código (Code): Se trata de un campo de un octeto (1 byte) que identifica el tipo de mensaje RADIUS que se envía o se recibe. Si se recibe un paquete con un campo Code inválido se descarta de manera silenciosa.
  Los códigos RADIUS (en decimal) se asignan de la siguiente manera:

  1: Access-Request
  2: Access-Accept
  3: Access-Reject
  4: Accounting-Request
  5: Accounting-Response
  11: Access-Challenge
  12: Status-Server (experimental)
  13: Status-Client (experimental)
  255: Reservado

- Identificador (Identifier): Octeto utilizado para encolar los mensajes, asegurando que se mantiene el orden de la conversación correctamente. El servidor RADIUS puede detectar una solicitud duplicada si tiene la misma dirección IP de origen del cliente, el mismo puerto UDP de origen y el mismo identificador.

- Longitud (Length): Campo que ocupa dos octetos (2 bytes) e indica la longitud del paquete incluyendo los campos Code, Identifier, Length, Authenticator y Attributes. Los octetos que estén fuera del rango indicado por el campo Length deben ser tratados como relleno y se deben ignorar al recibir el paquete. Si el paquete es más corto de lo que indica este campo, debe ser descartado de manera silenciosa. La longitud mínima de un paquete RADIUS es de 20 octetos y la longitud máxima es de 4096 octetos.

- Autenticador (Authenticator): Campo formado por 16 octetos (16 bytes) donde se inserta el control de integridad correspondiente al payload o carga útil, es decir, a los

datos principales del mensaje. Este valor se utiliza para autenticar la respuesta del servidor RADIUS y se emplea en el algoritmo de ocultación de contraseñas.

En los paquetes Access-Request, el valor de Authenticator es un número aleatorio de 16 octetos, llamado Request Authenticator. Este valor debería ser impredecible y único durante la vida útil del Shared Secret (clave compartida entre el cliente y el servidor RADIUS), ya que la repetición una solicitud junto con el mismo Shared Secret permitiría a un atacante responder con una respuesta interceptada previamente. Dado que se espera que el mismo Shared Secret pueda ser utilizado para autenticarse con servidores en diferentes regiones geográficas, el campo Request Authenticator debería mostrar unicidad global y temporal.

El valor del campo Authenticator en los paquetes Access-Accept, Access-Reject y Access-Challenge se llama Response Authenticator y contiene un hash MD5 unidireccional calculado sobre una secuencia de octetos que consiste en: el paquete RADIUS, comenzando con el campo Code, incluyendo el Identifier, el campo Length, el campo Request Authenticator del paquete Access-Request y los atributos de la respuesta, seguido del Shared Secret. Es decir:

ResponseAuth = MD5 (Code+ID+Length+RequestAuth+Attributes+Shared Secret)

Ecuación 4.10.- Obtención del campo Response Authenticator de un paquete RADIUS

La segunda parte es una **carga útil (payload)** que contiene todos los datos y está formada por [62, pp. 22-25]:

- Atributos (Attributes): Transportan información y detalles específicos relacionados con la autenticación, la autorización y la configuración tanto en las solicitudes como en las respuestas. El final de la lista de atributos se indica mediante el campo Length, que indica la longitud del paquete RADIUS. Algunos atributos pueden incluirse más de una vez. Cada atributo está formado por tres campos:

    - Tipo (Type): Ocupa un octeto. Si el servidor o el cliente RADIUS reciben un mensaje con un tipo de atributo desconocido, deben ignorarlo. Algunos tipos son: User-Name, User-Password, State, Framed-MTU, etc.

- Longitud (Length): El campo Length tiene un octeto e indica la longitud del atributo, incluyendo los campos Type, Length y Value. Si un atributo es recibido en un paquete Access-Accept, Access-Reject o Access-Challenge con una longitud inválida, el paquete debe ser tratado como un Access-Reject o bien descartado de manera silenciosa.

- Valor (Value): Este campo tiene un tamaño variable, que puede ser cero o más octetos, y almacena información específica del atributo. El formato y la longitud del campo Value están determinados por los campos Type y Length. El formato del campo Value es uno de estos cinco tipos de datos: text, string, address, integer o time.

4.8.3.- Tipos de paquetes

Access-Request

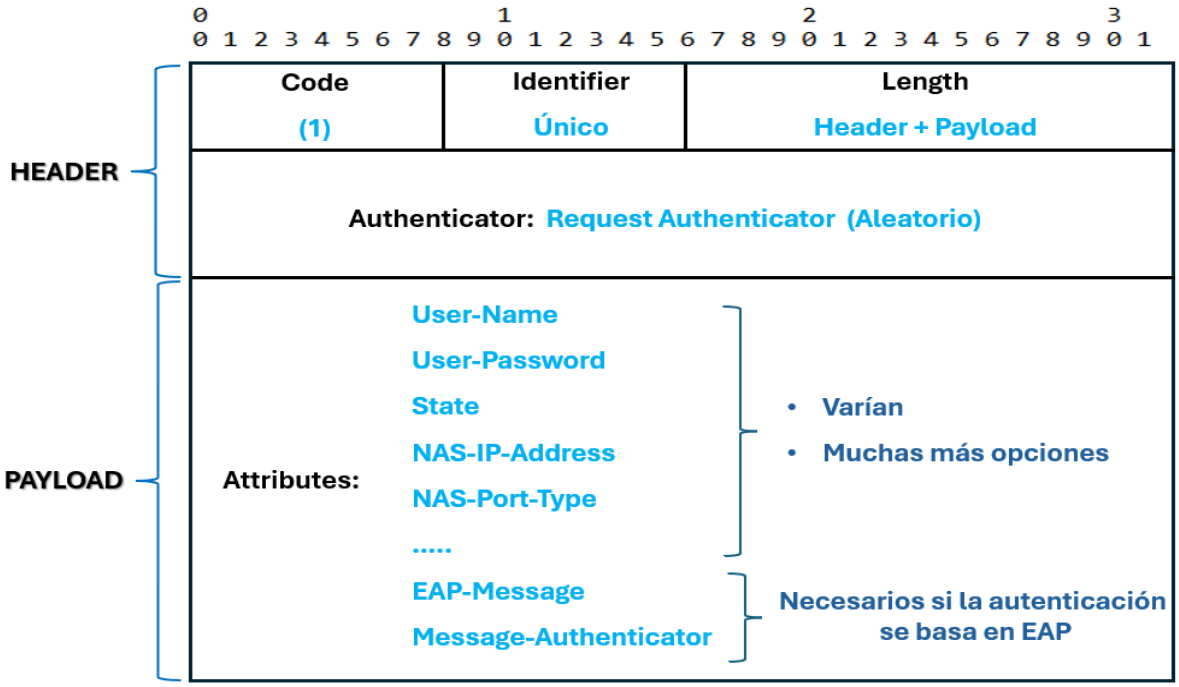


Figura 4.22.- Paquete RADIUS Access-Request

Los paquetes Access-Request son enviados a un servidor RADIUS para solicitar la autorización de acceso a un usuario a un NAS (Network Access Server). Estos paquetes

contienen información clave que el servidor usa para determinar si se permite el acceso y si se requieren servicios especiales para ese usuario. El campo Code para este tipo de solicitud debe tener siempre el valor 1.

Para autenticar a un usuario, el payload de un paquete Access-Request debe incluir el nombre de usuario (User-Name) y uno de los atributos que identifique al NAS, como NAS-IP-Address o NAS-Identifier. Además, debe contener información de autenticación, ya sea User-Password, CHAP-Password o State, pero no puede contener los atributos User-Password y CHAP-Password al mismo tiempo. Es importante que cuando se incluya un User-Password, este se proteja utilizando el algoritmo MD5 para garantizar que no se transmita en texto claro.

También puede incluir atributos adicionales, como NAS-Port o NAS-Port-Type para especificar el puerto de acceso, aunque esto no es obligatorio si no es relevante para la solicitud. Pueden incluirse otros atributos opcionales como sugerencias para el servidor, aunque no está obligado a seguirlas.

Por último, es importante mencionar que cuando RADIUS se usa para habilitar la autenticación EAP, los paquetes Access-Request, Access-Challenge, Access-Accept y Access-Reject deberían contener uno o más atributos EAP-Message. Si se incluye más de un atributo EAP-Message, se supone que los atributos se concatenan para formar un solo paquete EAP. Para evitar que los atacantes puedan manipular los resultados de autenticación (como hacer que un intento de acceso fallido sea reconocido como exitoso) es necesario que RADIUS proporcione autenticación por paquete y protección de integridad. Por lo tanto, el atributo Message-Authenticator debe usarse para proteger todos los paquetes Access-Request, Access-Challenge, Access-Accept y Access-Reject que contengan un atributo EAP-Message.

Los paquetes Access-Request que incluyan atributos EAP-Message sin un atributo Message-Authenticator serán descartados silenciosamente por el servidor RADIUS. Cuando un servidor RADIUS recibe un paquete con el atributo EAP-Message, debe calcular el valor del Message-Authenticator para verificar que el paquete no haya sido modificado. Si el valor calculado no coincide con el Message-Authenticator enviado en el paquete, el servidor debe

descartar el paquete de manera silenciosa, sin alertar al atacante, para asegurar la integridad de la comunicación.

El RFC especifica que el servidor RADIUS debe enviar una respuesta apropiada para cada paquete Access-Request válido, ya sea de autorización o de rechazo. Además, cada vez que se modifique algún atributo se deben generar nuevos paquetes (los campos Identifier y Request Authenticator van a cambiar), garantizando así la sincronización entre las solicitudes y las respuestas. [62, pp. 17-18]

Access-Accept

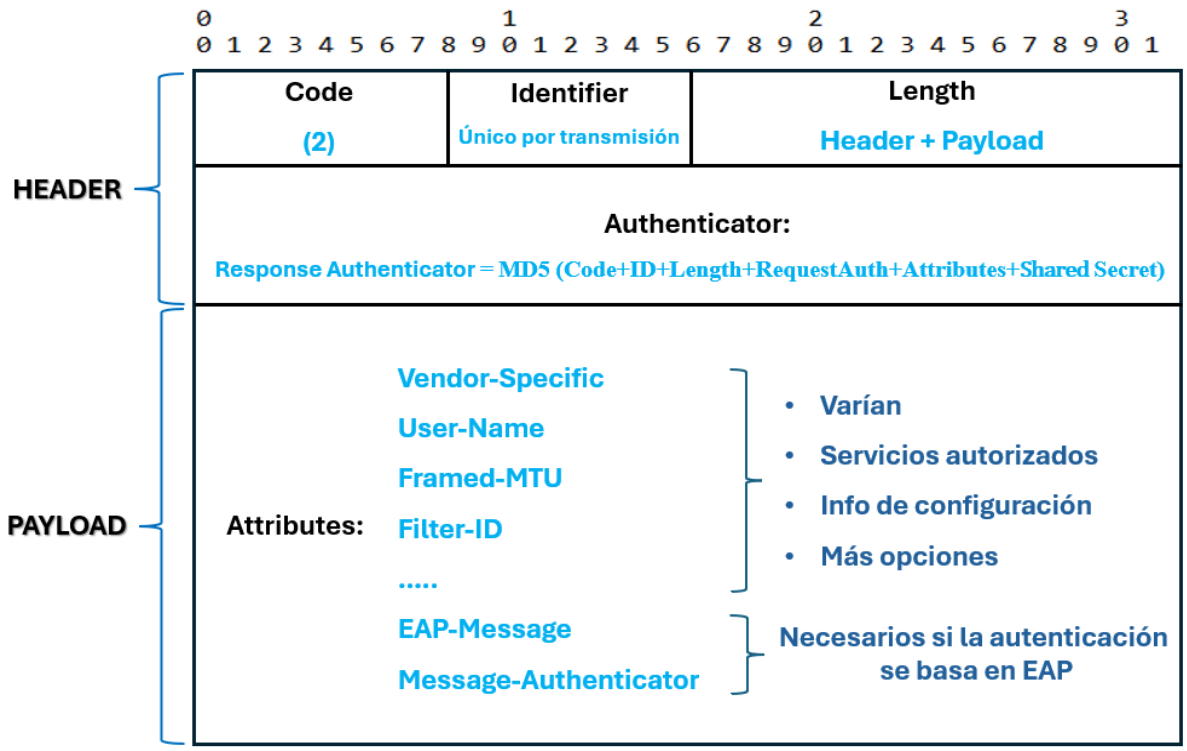


Figura 4.23.- Paquete RADIUS Access-Accept

Los paquetes Access-Accept son enviados por el servidor RADIUS para confirmar al cliente que se le ha concedido el acceso y contienen información de configuración específica necesaria para comenzar a ofrecer el servicio. Si todos los valores de los atributos recibidos en un Access-Request son aceptables, la implementación de RADIUS debe transmitir un paquete con el campo Code establecido en 2 (lo que indica que es un Access-Accept).

Cuando el cliente RADIUS recibe un paquete Access-Accept, verifica que el campo Identifier coincida con el del Access-Request que aún está pendiente de respuesta. Además, el campo Response Authenticator debe contener la respuesta adecuada para ese Access-Request específico. Si el paquete no es válido, se descarta sin generar ningún aviso o error visible.

Este paquete ofrece gran flexibilidad en lo relativo a la variedad y cantidad de atributos que puede incorporar. Generalmente, se especifican los servicios autenticados y autorizados para que el cliente pueda acceder a ellos y utilizarlos. Además, si el Access-Request tiene un EAP-Message, el Access-Accept debe incluir tanto un atributo EAP-Message como un atributo Message-Authenticator. [62, pp. 18-19]

Access-Reject

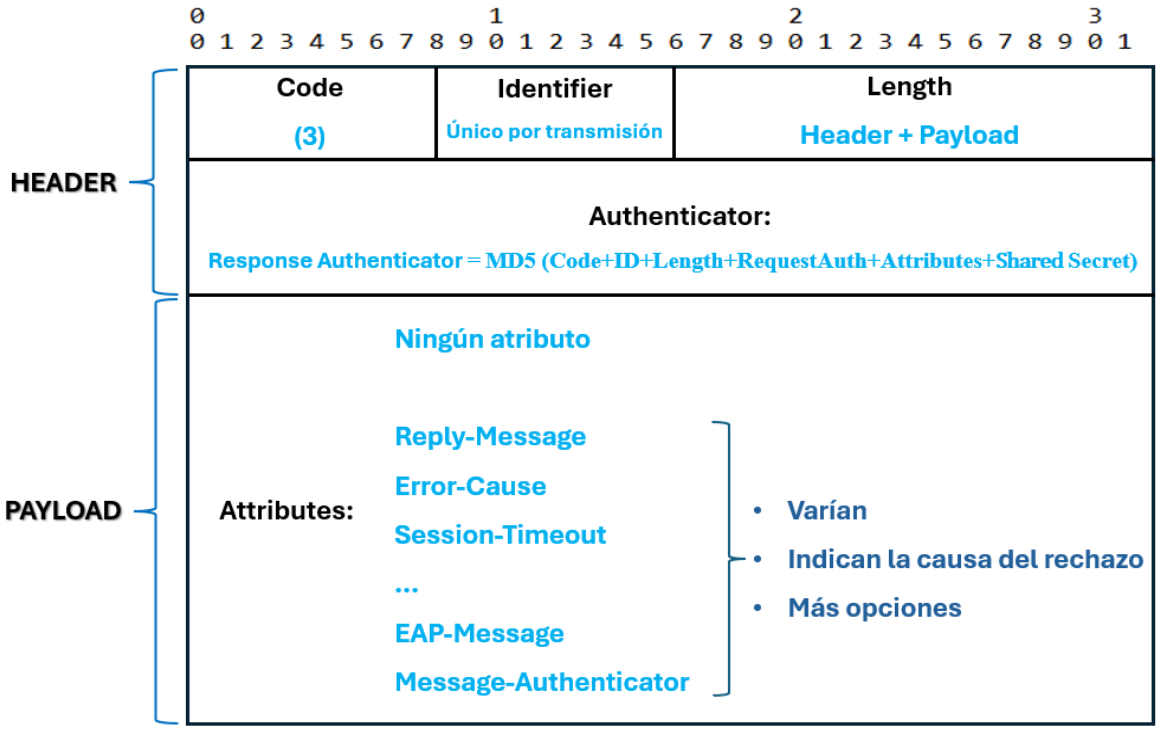


Figura 4.24.- Paquete RADIUS Access-Reject

Los paquetes Access-Reject son enviados por el servidor RADIUS para informa al cliente de que su petición de autenticación ha sido rechazada debido a credenciales inválidas, políticas de denegación o privilegios insuficientes. Si algún valor de los atributos recibidos

en el Access-Request no es aceptable, el servidor RADIUS debe enviar un paquete con el campo Code establecido en 3 (lo que indica que es un Access-Reject). Este tipo de paquete puede enviarse en cualquier momento de una sesión, algo que lo hace recomendable para obligar a respetar límites de conexiones. [62, p. 20]

El campo Identifier del Access-Reject debe coincidir con el del Access-Request pendiente de ser respondido y el campo Response Authenticator debe contener la respuesta adecuada para ese Access-Request específico. Si el paquete es inválido se descarta de forma silenciosa.

La carga útil o payload no es obligatorio que incluya atributos, pero puede incluir uno o más atributos Reply-Message con un mensaje de texto que el NAS pueda mostrar al usuario explicando la causa del rechazo. Además, si el Access-Request tiene un EAP-Message, el Access-Reject también puede incluir tanto un atributo EAP-Message como un atributo Message-Authenticator.

Access-Challenge

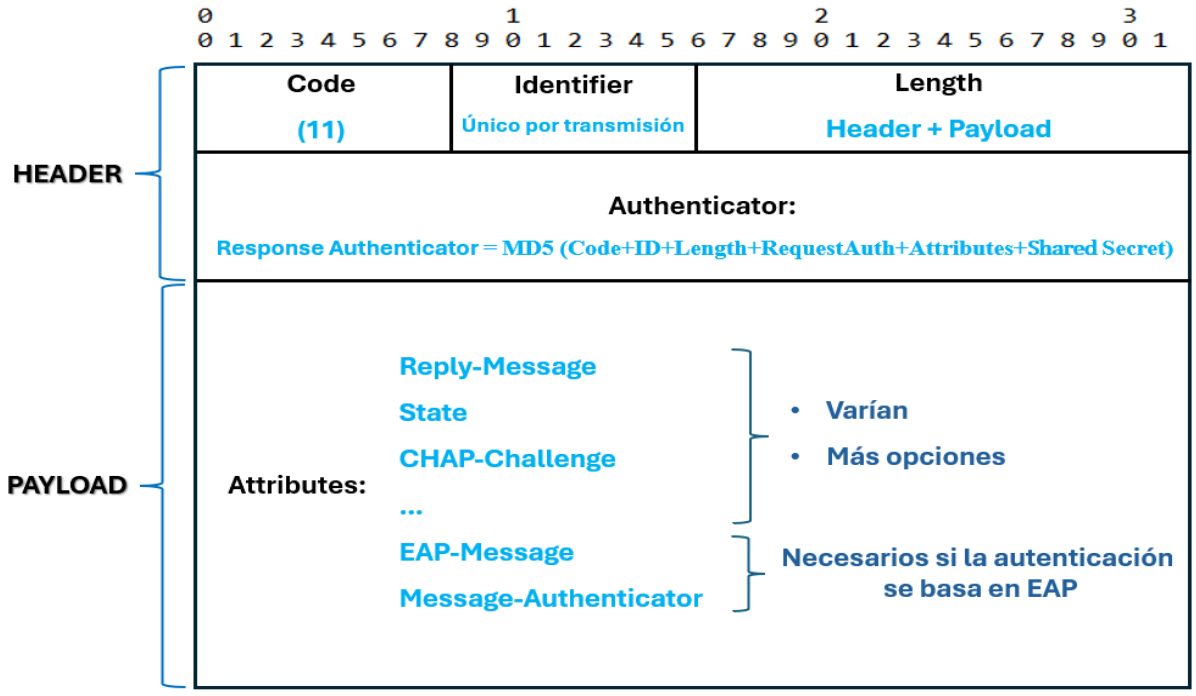


Figura 4.25.- Paquete RADIUS Access-Challenge

Los paquetes Access-Challenge pueden ser enviados por parte del servidor RADIUS, tras haber recibido un paquete Access-Request, en caso de que precise más información o quiera reducir el riesgo de fraude en la transacción. Si esto pasa, el servidor RADIUS debe enviar un paquete con el campo Code establecido en 11 (lo que indica que es un Access-Challenge) pidiéndole más información al cliente. Una vez que el cliente RADIUS conteste de nuevo con otro Access-Request correcto (incluyendo la información requerida y el valor del atributo State del Access-Challenge, si lo hay), el servidor decide si acepta, rechaza o pide nuevamente más datos. [62, pp. 21-22]

El campo Identifier del Access-Challenge debe coincidir con el del Access-Request pendiente de ser respondido y el campo Response Authenticator debe contener la respuesta adecuada para ese Access-Request específico. Si el paquete es inválido se descarta de forma silenciosa.

El payload puede incluir uno o más atributos Reply-Message y, opcionalmente, un atributo State (o ninguno). También se pueden incluir los atributos Vendor-Specific, Idle-Timeout, Session-Timeout y Proxy-State. Además, si el Access-Request tiene un EAP-Message, el Access-Challenge debe incluir tanto un atributo EAP-Message como un atributo Message-Authenticator.

4.8.4.- Funcionamiento

En estas arquitecturas, hay un componente intermedio denominado Servidor de Acceso a la Red (NAS, por sus siglas en inglés), que opera como un cliente RADIUS. Este cliente es el encargado de transmitir la información de los usuarios a los servidores RADIUS y, posteriormente, realizar las acciones correspondientes en función de la respuesta recibida.

El servidor RADIUS recibe la solicitud del cliente y autentica al usuario utilizando los datos proporcionados, enviando de vuelta la configuración con la información necesaria para permitir al cliente acceder al servicio asociado al usuario autenticado. Durante el proceso de autenticación y autorización, se verifica si el usuario es válido y se comprueba a qué recursos tiene derecho a acceder. Este proceso se complementa con la contabilidad, que

registra información clave de la sesión y se utiliza comúnmente para la generación de recursos de tarificación o para la realización de auditorías.

Cuando un cliente utiliza RADIUS para la autenticación, los usuarios pueden proporcionar sus credenciales de dos formas principales: mediante un formulario de inicio de sesión personalizable, en el que se solicita al usuario ingresar su nombre de usuario y su contraseña, o bien utilizando un protocolo de enlace como PPP (Point-to-Point Protocol), que incluye paquetes de autenticación que transportan dicha información.

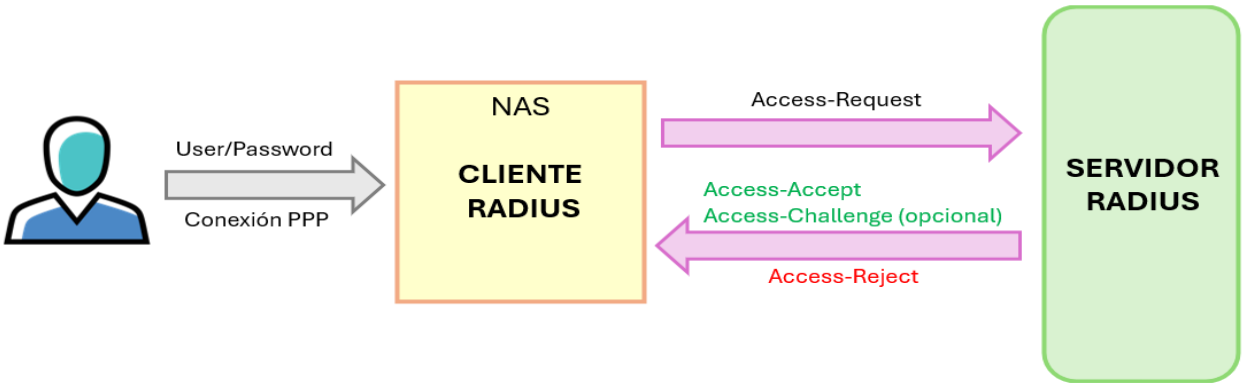


Figura 4.26.- Esquema del funcionamiento del protocolo RADIUS

Una vez que el cliente RADIUS ha obtenido dicha información, crea una solicitud Access-Request en la que se incluyen, entre otros atributos, el User-Name, la User-Password, el NAS-IP-Address, el NAS-Port-Type y el EAP-Message y el Message-Authenticator si se emplea EAP para la autenticación. Cuando hay una contraseña, se oculta utilizando un método basado en el algoritmo de resumen de mensajes RSA MD5. La solicitud Access-Request se envía al servidor RADIUS a través de la red. Si no se recibe una respuesta en un periodo de tiempo determinado, la solicitud se reenvía.

En el momento en el que el servidor RADIUS recibe la solicitud, validará al cliente que la envía. Una solicitud de un cliente para el cual el servidor RADIUS no tiene una clave compartida (Shared Secret) debe ser descartada silenciosamente. Si el cliente es válido, se procederá con la autenticación utilizando uno de los mecanismos soportados, como PAP, CHAP, EAP (basado en desafío-respuesta), Unix Login, LDAP, entre otros. Durante este proceso, se obtendrá la información relevante del usuario desde la base de datos, es decir,

los requisitos necesarios para concederle el acceso. Esto generalmente implica la verificación de la contraseña pero también puede incluir restricciones adicionales, como los clientes o puertos a los que el usuario tiene permitido acceder.

Si se cumplen todas las condiciones, el servidor RADIUS autorizará el acceso y responderá con un mensaje Access-Accept, adjuntando una lista de valores de configuración para el usuario. Estos valores especifican detalles como el tipo de servicio (por ejemplo, SLIP, PPP o formulario de inicio de sesión) y cualquier configuración adicional necesaria para proporcionar el servicio solicitado. En el caso de SLIP y PPP, los valores pueden incluir parámetros como la dirección IP, la máscara de subred, el MTU, la compresión requerida y los identificadores de filtros de paquetes. Para los usuarios que acceden a través de una interfaz de texto con un formulario, los valores podrían incluir el protocolo y el host deseados.

Si no se cumple alguna condición, se rechazará el acceso y se denegará la autenticación/autorización. Esto se notificará con un mensaje Access-Reject, que puede ser mostrado al usuario por el cliente e incluirá un texto que explique las razones del rechazo.

Si el cliente recibe un Access-Challenge y es compatible con el mecanismo de desafío/respuesta, puede mostrar al usuario el mensaje de texto, si lo hay, y luego generar una respuesta. Para ello, el cliente RADIUS creará una nueva solicitud Access-Request, que tendrá un Identifier y un Request-Authenticator diferentes a los de la solicitud Access-Request original. Esta nueva solicitud incluirá la información solicitada por el desafío en los atributos correspondientes, así como el valor del atributo State del Access-Challenge, si está presente. El servidor podrá responder a este Access-Request con un Access-Accept, un Access-Reject o con otro Access-Challenge.

Adicionalmente, una vez que el cliente ha sido autenticado y autorizado, puede enviar una solicitud de Accounting para iniciar una sesión. El servidor RADIUS responderá iniciando el registro de la conexión, donde se almacenarán datos sobre el inicio y el fin de la sesión, el volumen de datos transferidos y muchas otras cosas. La sesión se finalizará mediante una solicitud Accounting-Stop, que puede ser enviada tanto por el servidor como por el cliente. [59], [62, pp. 5-7]

5. Análisis.

5.1.- REQUISITOS

A continuación, se especifican los requisitos, tanto funcionales como no funcionales, necesarios para la implementación de la última versión del Internet Draft EAP-based Authentication Service for CoAP. Estos requisitos determinan las características que se deben cumplir para la correcta implementación del Draft.

5.5.1.- Requisitos funcionales

Se pueden observar en la tabla 5.1.

ID	Descripción
REQF01	Autenticación basada en EAP para dispositivos IoT: El sistema debe implementar el protocolo de autenticación EAP en dispositivos IoT utilizando CoAP como protocolo de transporte.
REQF02	Implementación de una arquitectura cliente-servidor: El sistema debe implementar una especie de arquitectura cliente-servidor, donde el cliente CoAP debe poder realizar solicitudes POST autenticadas utilizando EAP y el servidor CoAP debe aceptar y procesar esas solicitudes para devolver las respuestas correspondientes, de forma que se siga un modelo petición-respuesta.
REQF03	Implementación de la última versión del Internet Draft: El sistema debe implementar la última versión del Internet Draft EAP-based Authentication Service for CoAP de la forma más rigurosa posible, garantizando que se envíen todos los mensajes requeridos según las especificaciones del documento. Los mensajes deben estar en los formatos adecuados y cumpliendo con los campos obligatorios y opcionales definidos en el estándar. Todos los códigos de respuesta deben ser precisos, siguiendo las reglas del protocolo CoAP y reflejando correctamente el estado de cada

	solicitud o respuesta, como 2.01 (Created), 2.04 (Changed), 4.00 (Bad Request) y 4.04 (Not Found), entre otros.
REQF04	Integración con un servidor de AAA: El sistema debe integrarse con un servidor AAA que gestione la autenticación y la autorización de los dispositivos IoT a través del protocolo EAP-PSK. Debe ser capaz de enviar mensajes de autenticación y recibir las respuestas desde el servidor AAA utilizando el protocolo RADIUS correctamente.
REQF05	Compatibilidad con estándares de seguridad: El sistema debe cumplir con los estándares de seguridad recomendados por el IETF para la autenticación en IoT, como el uso de cifrado AES y algoritmos hash seguros (SHA-256 o superiores). Debe garantizar la confidencialidad e integridad de los mensajes intercambiados entre el cliente y el servidor durante el proceso de autenticación.
REQF06	Protección de mensajes: Los mensajes intercambiados durante el proceso de autenticación deben estar protegidos mediante los mecanismos de seguridad descritos en el draft, hasta donde sea posible, en función del nivel de soporte que proporcionen las bibliotecas utilizadas.
REQF07	Garantía de orden en las solicitudes CoAP: El sistema debe garantizar que las solicitudes CoAP (POST, DELETE, etc.) se procesen en el orden correcto utilizando las URIs, manteniendo la coherencia del estado entre cliente y servidor. Debe asegurarse que los mensajes se entreguen y procesen en el orden adecuado para evitar inconsistencias.
REQF08	Negociación de cipher suite utilizando estructuras CBOR: El sistema debe negociar el conjunto de cifrado (cipher suite) a utilizar para la protección de los mensajes EAP-CoAP utilizando estructuras CBOR para la codificación de los parámetros de negociación.
REQF09	Manejo de diferentes escenarios: El sistema debe manejar correctamente situaciones como un fallo durante la autenticación o como el envío de un mensaje POST a una URI que ya ha sido eliminada (mensaje duplicado). En ambas situaciones el sistema debe responder conforme a las especificaciones del Draft. En el caso de una solicitud DELETE exitosa para eliminar el estado CoAP-EAP del dispositivo IoT, el sistema también

	debe actuar adecuadamente, eliminando la URI y asegurando que la URI eliminada no pueda ser utilizada nuevamente ya que el dispositivo dejará de estar autenticado.
REQF10	Soporte para EAP-PSK: El sistema debe implementar el método de autenticación EAP-PSK siendo capaz de manejar los intercambios específicos de mensajes. Debe asegurarse de que la AK, la KDK, las claves de sesión, los canales protegidos y, por tanto, los cuatro mensajes del protocolo se generan correctamente.
REQF11	Visualización de los mensajes en Wireshark y en la consola de Python: Los mensajes intercambiados durante el proceso de autenticación deben ser capturables en Wireshark para poder ver la estructura completa de cada mensaje, incluyendo los campos relevantes del protocolo CoAP o RADIUS y otros metadatos. Además, en la consola de Python debe ser posible visualizar la construcción de los mensajes EAP-PSK, el contenido de cada mensaje CoAP intercambiado (código, URI y payload) y otros datos importantes, como los parámetros del contexto OSCORE.

Tabla 5.1.- Requisitos funcionales de la aplicación

5.1.2.- Requisitos no funcionales

Los requisitos no funcionales definen cómo debe comportarse el sistema en términos de calidad y eficiencia sin enfocarse en las funciones específicas que realiza.

5.1.2.1.- Requisitos de usuario

ID	Descripción
REQNF01	El usuario que quiera inspeccionar el código debe saber español, puesto que los comentarios y la salida por consola están en ese idioma.
REQNF02	El usuario debe tener conocimientos básicos de informática, de los protocolos EAP, CoAP, OSCORE y RADIUS y de dispositivos IoT para poder comprender el código.
REQNF03	El usuario que desee ampliar, actualizar o mejorar las capacidades de la implementación ha de tener conocimientos de Python y C.

REQNF04	El usuario que desee inspeccionar y analizar los paquetes intercambiados durante el proceso de autenticación debe tener familiaridad con Wireshark u otras herramientas similares.

Tabla 5.2.- Requisitos no funcionales de usuario

5.1.2.2.- Requisitos tecnológicos

ID	Descripción
REQNF05	Es necesario disponer de un ordenador con PyCharm para ejecutar el código del EAP Peer y del EAP Controller, así como de una máquina virtual para poder ejecutar el Servidor AAA lanzando FreeRadius.
REQNF06	El equipo debe tener instalada la versión 3.8 de Python o superior.
REQNF07	El equipo debe tener instalada principalmente la biblioteca aiocoap. También es interesante que tenga instaladas las bibliotecas asyncio, pycryptodome, cryptography y scapy entre otras.

Tabla 5.3.- Requisitos no funcionales tecnológicos

5.1.2.3.- Requisitos de usabilidad

ID	Descripción
REQNF08	El código debe ser intuitivo, legible y modular, con nombres de variables y funciones que reflejen su propósito claramente. Además, debe seguir buenas prácticas de programación (por ejemplo, el uso de comentarios o la separación de la lógica y de las funciones), para que otros desarrolladores puedan entenderlo y modificarlo fácilmente.
REQNF09	El proyecto debe incorporar un “manual” que explique cómo se llevó a cabo la implementación, cómo poder lanzarla o ejecutarla y cómo verificar los resultados.
REQNF10	El sistema debe permitir al usuario realizar pruebas y obtener resultados de manera rápida, sin necesidad de configuraciones complejas, para asegurar una experiencia de usuario eficiente y sin fricciones.

Tabla 5.4.- Requisitos no funcionales de usabilidad

5.1.2.4.- Requisitos de escalabilidad

ID	Descripción
REQNF11	El sistema debe ser escalable, es decir, el código debe estar diseñado de forma que sea fácil agregar nuevas funcionalidades o realizar modificaciones en la implementación sin necesidad de reestructurarlo completamente.

Tabla 5.5.- Requisitos no funcionales de escalabilidad

5.1.2.5.- Requisitos de rendimiento

ID	Descripción
REQNF12	El sistema no debe hacer uso de una gran cantidad de recursos del equipo (como memoria o procesamiento) y el tiempo de autenticación debe ser lo más rápido posible.
REQNF13	Dado que los dispositivos IoT suelen operar en redes limitadas, el sistema debe minimizar el uso de ancho de banda, optimizando los mensajes y evitando el envío de datos innecesarios.

Tabla 5.6.- Requisitos no funcionales de rendimiento

5.2.- ANÁLISIS DEL SOFTWARE

En esta sección se van a comentar las tecnologías empleadas durante el desarrollo del proyecto que formarán parte del software y, a su vez, se justificará el por qué se han elegido en base a sus características y a las ventajas que presentan frente a otras similares.

5.2.1.- Python

Python es el lenguaje de programación elegido para la implementación del código de la última versión del Internet Draft EAP-based Authentication Service for CoAP. Se trata de un lenguaje de programación de alto nivel, interpretado, imperativo, funcional, orientado a objetos y de propósito general, que debido a su simplicidad, legibilidad y versatilidad se ha hecho muy popular en la comunidad de desarrollo de software. [63]

El ser un lenguaje de alto nivel significa que se enfoca en la legibilidad y simplicidad del código, lo que permite escribir programas de manera más rápida y eficiente ya que el desarrollador se puede centrar en la lógica del programa en lugar de en los detalles de bajo nivel.

Al ser interpretado, el código se ejecuta línea por línea mediante un intérprete, sin necesidad de compilarlo previamente. Esto simplifica la depuración y permite a los desarrolladores observar los resultados de inmediato.

En cuanto a su paradigma imperativo, Python permite modificar el estado del programa mediante condiciones o instrucciones que indican al computador cómo realizar una tarea. Además, admite la programación funcional, lo que permite operar con datos de entrada y salida, brindando flexibilidad para procesar y transformar datos a lo largo del programa.

Python es también un lenguaje orientado a objetos, lo que facilita la creación y manipulación de objetos con atributos y métodos, promoviendo la reutilización del código y la modularidad. Esto hace que sea ideal para proyectos grandes que requieren un mantenimiento sencillo y escalabilidad, como el que aborda este TFG.

Python es libre y de código abierto, destaca por su sintaxis clara y legible, que facilita la escritura de código de manera eficiente y su comprensión, lo que agiliza el desarrollo y el trabajo. Con su tipado dinámico no es necesario declarar explícitamente el tipo de las variables, ya que el intérprete lo deduce automáticamente. Además, cuenta con un sistema de recolección de basura automática que gestiona la memoria liberando recursos innecesarios.

Algo muy importante a la hora de elegir este lenguaje es la gran comunidad que respalda a Python, la cual ofrece bibliotecas, tutoriales y soporte en foros, lo que facilita tanto el aprendizaje como la resolución de problemas. También dispone de una extensa biblioteca estándar que cubre diversas necesidades, lo que ahorra tiempo en el desarrollo de aplicaciones.

Python es un lenguaje multiplataforma, por lo que el código puede ejecutarse en distintos sistemas operativos sin grandes modificaciones. Además, es interesante destacar que permite un rápido desarrollo de prototipos, ideal para que poder probar ideas antes de la implementación final. Por último, su capacidad de integración con otros lenguajes, como C y C++, facilita el aprovechamiento de código existente y mejora el rendimiento cuando es necesario. Al haber tenido que "replicar" en Python varias funciones y trozos de códigos desarrollados en C para la implementación del protocolo EAP-PSK, esta característica fue de gran utilidad para agilizar la programación. [64]

5.2.2.- PyCharm

PyCharm es un IDE (Entorno de Desarrollo Integrado) muy potente y completo, diseñado específicamente para Python. Cuenta con una amplia gama de herramientas que facilitan el desarrollo de software, como el resaltado de la sintaxis, la finalización automática del código, la depuración, la refactorización y el control de versiones, lo que mejora notablemente la eficiencia y precisión en la escritura de código. [65]

Además, PyCharm se integra fácilmente con herramientas populares de desarrollo como sistemas de control de versiones (VCS por sus siglas en inglés) como Git, lo cual ha sido útil para gestionar y descargar diversas bibliotecas necesarias para el proyecto. También permite integrar herramientas de virtualización como Docker, proporcionando un entorno de desarrollo más versátil. [66]

Una de las razones adicionales para elegir PyCharm es la experiencia previa con la versión gratuita, PyCharm Community Edition, la cual hay que utilizar en otras asignaturas de la titulación como Fundamentos de Informática o Redes y Servicios Móviles. Esta familiaridad con el entorno brinda mayor agilidad y destreza en la implementación del código, ahorrando tiempo y facilitando el flujo de trabajo.

Por último, PyCharm ofrece un análisis estático avanzado capaz de detectar errores en tiempo real mientras se escribe el código. Esto permite identificar y corregir errores de sintaxis, problemas de estilo y referencias no utilizadas de manera proactiva, evitando que

pequeños problemas se conviertan en errores más graves. Gracias a esto, se puede mantener un código más limpio y eficiente durante todo el proceso de desarrollo.

5.2.3.- VMWare

VMware Workstation es una plataforma de virtualización que permite a los usuarios crear y ejecutar máquinas virtuales en el ordenador, facilitando la ejecución de múltiples sistemas operativos simultáneamente en un solo equipo físico. En este caso sirve para ejecutar la máquina virtual "TFG Daniel" que aloja el Servidor AAA (un Servidor RADIUS), que va a actuar como el EAP Server de la arquitectura CoAP-EAP. Esta herramienta es ampliamente utilizada por desarrolladores, administradores de sistemas y profesionales de TI para probar aplicaciones, ejecutar diferentes entornos operativos y realizar simulaciones sin la necesidad de hardware adicional.

Una de las principales razones para elegir VMware Workstation frente a otros programas similares es su estabilidad y rendimiento. VMware es conocido por ofrecer un alto nivel de optimización en la ejecución de máquinas virtuales, lo que permite a los usuarios aprovechar al máximo los recursos de su hardware físico. Además, es compatible con una amplia gama de sistemas operativos, tanto como host (el sistema en el que se instala) como guest (los sistemas operativos que se ejecutan dentro de las máquinas virtuales), lo que lo convierte en una solución flexible y adaptable a diferentes necesidades. [67]

VMware Workstation también destaca por sus avanzadas funciones de gestión y configuración de red, lo que lo convierte en una herramienta ideal para probar entornos complejos que requieren la simulación de redes con múltiples máquinas virtuales. Su capacidad para manejar redes virtuales avanzadas, junto con la facilidad para configurar instantáneas, clonar máquinas virtuales y usar diferentes opciones de almacenamiento, lo hace especialmente útil en entornos de desarrollo y pruebas de software.

En comparación con otras soluciones como VirtualBox, VMware Workstation suele ofrecer un rendimiento más sólido, mayor robustez, mejor soporte técnico y una mayor integración con herramientas empresariales, lo que la convierte en una elección preferida

para profesionales que buscan fiabilidad y funcionalidades adicionales como la integración con servicios en la nube a coste cero.

5.2.4.- Wireshark

Wireshark es el programa empleado para visualizar los paquetes intercambiados durante todo el proceso de autenticación y comprobar si la autenticación ha tenido éxito o no.

Es el analizador de protocolos de red más destacado a nivel mundial y permite ver lo que está sucediendo en la red a nivel microscópico [68]. Es además una herramienta gratuita y de código abierto utilizada para examinar y depurar el tráfico de red, tanto en redes locales como en redes de área amplia. Este software es capaz de abordar una amplia variedad de problemas que incluyen desde la pérdida de paquetes, retransmisiones, fragmentación y problemas de latencia, hasta actividades maliciosas en la red tales como ataques, intentos de escaneo de puertos, tráfico malicioso o vulnerabilidades como SSL stripping, ataques man-in-the-middle (MITM) y otros.

La captura y el análisis profundo de los paquetes es una de las principales características que hacen que este programa sea la mejor opción, ya que permite capturar el tráfico de red en tiempo real y desglosar cada paquete a nivel de bit. Esto da acceso detallado a la información de cada capa del modelo OSI (desde la capa física hasta la capa de aplicación). Para cualquier protocolo, Wireshark muestra los encabezados, los datos y cualquier posible anomalía o error en los paquetes. [69]

Otra característica importante es que ofrece filtros de visualización potentes que permiten enfocarse en paquetes específicos. Se pueden usar filtros de captura y visualización para aislar tráfico de ciertos protocolos, direcciones IP, puertos, etc., lo que facilita identificar patrones o resolver problemas sin sobrecargarse con información innecesaria.

Además, es compatible con más de 1000 protocolos de red, como UDP, TCP, HTTP o DNS entre otros, y también es compatible con una amplia variedad de sistemas operativos, incluyendo Linux, Windows y macOS. De modo que, al implementar y probar protocolos

de red como CoAP, EAP o RADIUS, Wireshark permite capturar los mensajes intercambiados y verificar que se ajustan al estándar esperado, confirmando que los mensajes se están enviando y recibiendo correctamente.

En la industria es muy utilizado para resolver problemas de red, tanto en un entorno de desarrollo como en uno de producción. La educación también hace mucho uso de él para el aprendizaje y la comprensión de los protocolos de red, siendo otro de los programas más utilizados en las asignaturas del área de telemática que se imparten en el grado.

5.2.5.- Aiocoap

La biblioteca aiocoap es la implementación más actual y completa en Python del protocolo CoAP (Constrained Application Protocol), diseñada específicamente para entornos con recursos limitados como dispositivos IoT. [70]

Es una biblioteca de código abierto mantenida por la comunidad, con actualizaciones periódicas y mejoras. Esto asegura que la biblioteca evoluciona en consonancia con las necesidades de los desarrolladores y los avances en el protocolo CoAP. Además, al estar implementada en Python, permite una curva de aprendizaje más suave en comparación con otras bibliotecas en lenguajes más complejos como C o C++. Esto hace que sea más fácil desarrollar, mantener y depurar aplicaciones CoAP.

La biblioteca puede ser usada tanto para crear clientes que envían peticiones CoAP como para implementar servidores que responden a esas peticiones. Esto permite cubrir todos los aspectos de la comunicación dentro de una red de dispositivos IoT. Además, utiliza un enfoque asíncrono basado en asyncio, lo que permite manejar múltiples solicitudes de red de manera concurrente y eficiente, optimizando el uso de recursos y mejorando el rendimiento en dispositivos con limitaciones de procesamiento.

Aunque la biblioteca está diseñada para funcionar en entornos de dispositivos IoT, también puede ser utilizada en ordenadores de escritorio y servidores, lo que la convierte en una herramienta flexible para pruebas y desarrollo en diferentes entornos.

Por último, otro detalle que hace que sea la biblioteca elegida, es que es la única que implementa (aunque no completamente) la extensión de seguridad OSCORE del protocolo COAP.

5.2.6.- RADIUS

RADIUS es el protocolo empleado para el intercambio de mensajes entre el cliente RADIUS presente en el EAP Authenticator y el Servidor AAA, que actúa como el EAP Server, el cual en este TFG se trata de un servidor RADIUS presente en una máquina virtual.

Elegir RADIUS como protocolo frente a otros similares depende de varios factores como los requisitos de la red, la seguridad, la escalabilidad y la compatibilidad de los dispositivos. Una de las principales razones para optar por este protocolo es su madurez y amplio soporte en la industria. Al ser uno de los protocolos de autenticación más antiguos y probados, ha demostrado ser una opción segura y confiable en diversos entornos de red a lo largo de décadas.

Otra ventaja es su compatibilidad con una amplia gama de dispositivos de red, como routers, firewalls, puntos de acceso inalámbricos y conmutadores. Esto facilita su integración en infraestructuras de red heterogéneas sin necesidad de realizar modificaciones importantes.

RADIUS es muy flexible en cuanto a las opciones de autenticación que ofrece. Los administradores de red pueden elegir entre contraseñas, bases de datos externas como LDAP o Active Directory, así como otras opciones, incluyendo tokens de seguridad, certificados digitales y métodos EAP, adaptándose así a las necesidades específicas de cada entorno.

No solo se limita a la autenticación, sino que también proporciona funciones de autorización y contabilidad, lo que permite a los administradores controlar el acceso a los recursos y llevar un registro detallado de las actividades de los usuarios. Esto es útil para auditorías, facturación y cumplimiento de políticas. Además, RADIUS cuenta con mecanismos de seguridad y cifrado bastante sólidos para proteger tanto las credenciales de los usuarios como la integridad de los datos transmitidos, garantizando un acceso seguro y confiable a la red.

Por último, uno de los factores clave para elegir este protocolo fue que el servidor RADIUS ya estaba completamente implementado en la máquina virtual proporcionada por el tutor del TFG, quien había trabajado previamente en el desarrollo de CoAP-EAP. De este modo, solo fue necesario desarrollar un cliente RADIUS para la comunicación con dicho servidor, en lugar de tener que implementar tanto el cliente como el servidor a ciegas y desde cero, lo que simplificó considerablemente el proceso.

5.2.7.- EAP-PSK

EAP-PSK es el método EAP elegido por el EAP Server para llevar a cabo la autenticación. Fue diseñado con varios objetivos clave en mente [40, pp. 4-5]. El primero es la simplicidad, lo que implica que debe ser fácil de implementar y desplegar sin necesidad de una infraestructura previa. Para lograr esta simplicidad, EAP-PSK se basa únicamente en el algoritmo criptográfico AES-128, evitando el uso de criptografía asimétrica como el intercambio de claves Diffie-Hellman. Esta elección simplifica la implementación al eliminar la necesidad de negociaciones criptográficas, haciéndolo ideal para dispositivos con limitaciones de procesamiento y memoria. Sin embargo, este enfoque limita algunas capacidades, como la protección de identidad y el Perfect Forward Secrecy (PFS), priorizando la simplicidad sobre características de seguridad avanzadas.

Además, EAP-PSK utiliza un formato de mensaje fijo en lugar del diseño basado en Tipo-Longitud-Valor (TLV), lo que contribuye a su facilidad de uso. Otro objetivo clave es su amplia aplicabilidad, ya que está diseñado para autenticar en cualquier red, incluyendo redes LAN inalámbricas IEEE 802.11.

En términos de seguridad, EAP-PSK adopta un enfoque conservador al basar su diseño en primitivas criptográficas y protocolos ya existentes y probados por la comunidad criptográfica. De esta manera, se evita la necesidad de tener que inventar nuevos mecanismos criptográficos, lo que reduce el riesgo de errores en la implementación.

Por último, EAP-PSK ha sido diseñado para ser extensible. Proporciona un mecanismo explícito que permite futuras extensiones dentro de su canal protegido, lo que permitirá agregar servicios más sofisticados en el futuro según las necesidades.

6. Diseño e implementación.

Tras la fase de análisis del sistema, se pasa a la de diseño e implementación, donde se definirá la estructura final del proyecto junto con el entorno en el que se desarrollará. También, se explicarán con detalle las partes más relevantes del código de los diferentes archivos (scripts) que componen los tres elementos de la arquitectura encargados de llevar a cabo el flujo de operación de la implementación final de la última versión del Draft.

6.1.- ENTORNO TECNOLÓGICO DE DESARROLLO

En este apartado se detallará el equipo hardware y software utilizado para el desarrollo, así como una descripción de los archivos que conforman la implementación.

6.1.1.- Equipo hardware

El equipo hardware consiste en un ordenador portátil Gaming HP VICTUS 16-E0015NS con las siguientes características:

- Procesador AMD RyzenTM 7 5800 H
- Sistema Operativo Windows 10
- Tarjeta gráfica NVIDIA® GeForce GTX™ 1650
- Memoria RAM de 16 GB
- Disco duro de 512 GB

6.1.2.- Equipo software

El equipo software utilizado para el desarrollo incluye:

- Pycharm Community 2023.3.4
- Python 3.12
- VMWare Workstation 17 Player
- Wireshark

6.1.3.- Archivos del proyecto

El proyecto se compone varios archivos .py, necesarios para implementar la funcionalidad al completo que deberán poseer todos los componentes que conforman la arquitectura para que la autenticación tenga éxito, cumpliendo en la medida de lo posible y de la mejor forma las especificaciones de la última versión del Draft. A continuación, se ofrece una breve descripción de cada uno de ellos:

- **Peer.py:** Este código implementa el EAP Peer del protocolo CoAP-EAP y tiene varias responsabilidades clave. Primero, envía el mensaje POST de activación a la URI /.well-known/coap-eap. Luego, tras recibir el mensaje EAP Request ID del EAP Authenticator, genera y envía el EAP Response ID. Posteriormente, parsea y procesa los mensajes EAP-PSK procedentes del EAP Authenticator (ya sea el primero o el tercero) para generar y enviar la respuesta correspondiente (el segundo o el cuarto mensaje EAP-PSK) utilizando el protocolo CoAP como transporte. Finalmente, si recibe el mensaje EAP Success protegido por OSCORE desde el EAP Authenticator, genera un mensaje con código 2.04 Changed, también protegido por OSCORE, concluyendo así el proceso de autenticación.

- **Controller.py:** Este código implementa el Controlador (EAP Authenticator) del protocolo CoAP-EAP. Su función comienza enviando el mensaje EAP Request ID tras recibir el mensaje de activación desde el EAP Peer. Posteriormente, procesa el EAP Response ID y los mensajes EAP-PSK encapsulados en CoAP que recibe del EAP Peer y los reenvía a través de un cliente RADIUS al EAP Server, que actúa como un servidor RADIUS. De igual forma, recibe los mensajes RADIUS con los mensajes EAP-PSK generados por el EAP Server, los procesa y los retransmite al cliente utilizando el protocolo CoAP como transporte. Finalmente, si no hubo ningún error, envía al EAP Peer el mensaje de éxito (EAP Success) protegido por OSCORE.

- **Cliente_radius.py:** Este código implementa un cliente RADIUS integrado en el EAP Authenticator, diseñado para comunicarse con un servidor RADIUS desarrollado en C y ubicado en una máquina virtual, que actúa como EAP Server. El cliente incluye la lógica necesaria para construir y enviar mensajes RADIUS al

servidor, así como para recibir y procesar las respuestas. En esta comunicación, el cliente enviará al servidor en el atributo EAP-Message de un mensaje RADIUS Access-Request, el EAP Response ID, el segundo y el cuarto mensaje EAP-PSK. También recibirá del servidor, en el atributo EAP-Message de un mensaje RADIUS Access-Challenge, el primer y el tercer mensaje EAP-PSK y en el atributo EAP-Message de un mensaje RADIUS Access-Accept, el mensaje EAP Success.

- **Radiusattr.py:** Este código se ocupa de definir, construir y manejar los atributos de un paquete RADIUS.

- **Radiusext.py:** Este código se ocupa de definir, construir y manejar los campos estándar de la cabecera (header) de un paquete RADIUS.

- **RadiusMsgGenerator.py:** Este código se encarga de construir y enviar solicitudes de autenticación RADIUS (Access-Requests), manejando la creación de atributos como el Message-Authenticator y asegurando que el formato de los paquetes es correcto.

- **Mensajes_EAP_PSK.py:** Este código define e inicializa todas las variables relacionadas con EAP-PSK. También implementa todas las funciones necesarias para obtener las claves estáticas de larga duración derivadas de la PSK (AK y KDK), las claves de sesión (TEK, MSK y EMSK) y el canal protegido. Además, implementa todas las funciones necesarias para parsear el primer y el tercer mensaje EAP-PSK y para construir el segundo y el cuarto mensaje EAP-PSK en el EAP Peer.

- **Calculos_OSCORE.py:** Este código contiene la función derive_OSCORE_keys, la cual permite obtener algunos de los parámetros necesarios para crear el contexto de seguridad del protocolo OSCORE (la Master Secret, la Master Salt, el Sender ID y el Recipient ID) a partir de un Master Secret (MSK) y una cadena (CS) en formato hexadecimal. La cadena CS son los valores concatenados de la negociación de cipher suite tanto del Controlador como de la respuesta del dispositivo IoT.

- **Redes.py:** Este código contiene la función get_wipi_ip, la cual se utiliza para obtener la dirección IP del adaptador Wi-Fi, que será la dirección en la que escucha el contexto de servidor CoAP del Controlador.

- **Decrypt_ms_key.py:** Este código implementa una función destinada a descifrar las claves MS contenidas en los atributos RADIUS Vendor-Specific con el Vendor ID de Microsoft, incluidos en el mensaje Access-Accept enviado por el EAP Server. Al concatenar la MS-MPPE-Recv-Key descifrada con la MS-MPPE-Send-Key descifrada, se obtiene la MSK, la cual es fundamental para que el EAP Authenticator pueda derivar su contexto de seguridad OSCORE.

6.2.- PATRONES DE DISEÑO

Para abordar problemas que aparecen con frecuencia en el diseño de software se aplican unas soluciones reutilizables y probadas conocidas como patrones de diseño. Básicamente son plantillas que proporcionan un enfoque estructurado para optimizar el diseño del código, basándose en el conocimiento y la experiencia adquirida por desarrolladores que previamente han abordado problemas similares. No se trata de modelos rígidos, más bien son guías flexibles que se adaptan a las necesidades específicas de cada proyecto. [71]

Los principales objetivos que tienen los patrones de diseño son: estandarizar el lenguaje entre programadores, evitar perder tiempo en dar soluciones a problemas ya resueltos o conocidos y crear código reusable.

En este proyecto se emplea el modelo Petición-Respuesta, el cual, aunque no es un patrón de diseño clásico en el sentido tradicional de patrones como Singleton o Factory, puede considerarse un patrón de diseño arquitectónico ampliamente utilizado en sistemas distribuidos y de comunicación. Su simplicidad y efectividad lo convierten en una opción natural para aplicaciones como la autenticación y el control de acceso. Al tratarse de un modelo arquitectónico, facilita la estructuración de la comunicación entre los componentes del sistema, garantizando eficiencia y claridad en el intercambio de mensajes.

6.2.1.- Modelo petición-respuesta

El modelo de petición-respuesta es una arquitectura de comunicación en la que un cliente envía una petición a un servidor y el servidor le responde con una respuesta. Este modelo es la base de muchos protocolos de red, como HTTP o CoAP entre otros. [72]

Consta de los siguientes componentes:

Figura 6.1.- Esquema del modelo petición-respuesta

- **Cliente:** Es quien inicia la comunicación enviando una petición al servidor. El cliente puede ser un dispositivo, una aplicación o cualquier entidad que requiera algún servicio del servidor.
- **Petición:** Es el mensaje que el cliente envía al servidor. Contiene:
  - Un **método** (como GET, POST… en HTTP o CoAP) que indica la acción que el cliente quiere realizar.
  - Una **URI** (Uniform Resource Identifier) o recurso que identifica lo que se está solicitando o dónde se va a realizar la operación.
  - Un **payload** (opcional) que incluye los datos adicionales necesarios para la operación, como parámetros o contenidos específicos.
  - **Encabezados u opciones** que proporcionan información adicional sobre la petición (como autenticación, formato de contenido, etc.).
- **Servidor:** Es quien recibe la petición del cliente y ejecuta la operación solicitada. Luego, envía de vuelta una respuesta al cliente.

- **Respuesta:** Es el mensaje que el servidor envía de vuelta al cliente. Contiene:
  - Un **código de estado** que indica si la operación fue exitosa o si ocurrió un error (por ejemplo, 200 OK en HTTP o 2.04 Changed en CoAP).
  - Un **payload** (opcional), que contiene los datos o resultados que el cliente solicitó.
  - **Encabezados u opciones** que proporcionan información adicional, como formato de los datos, autenticación, etc.

Por tanto, el funcionamiento de la comunicación entre el EAP Peer y el EAP Authenticator (cabe recordar que el EAP Peer solo se comporta como cliente para el envío de la petición POST a la URI /.well-known/coap-eap, el resto del tiempo se comportará como servidor) sería este:

1. El cliente envía una petición POST a la URI indicada (/.well-known/coap-eap, /auth/eap/1, /auth/eap/2, etc.) al servidor.
2. El servidor recibe la petición, la procesa, realiza las acciones correspondientes (por ejemplo, generar el mensaje EAP-PSK (EAP-PSK-2 o EAP-PSK-4), crear un nuevo recurso asociado a una nueva URI, eliminar el recurso anterior…) y luego crea una respuesta.
3. El servidor envía la respuesta al cliente.
4. El cliente recibe la respuesta y la procesa (mostrando los datos por consola y almacenándolos).

6.3.- ARQUITECTURA

La arquitectura propuesta para la implementación de la última versión del Draft consta de tres componentes: un EAP Peer (dispositivo IoT), un EAP Authenticator (Controlador) y un EAP Server (Servidor RADIUS).

El flujo de operación que se ha implementado, llevado a cabo entre estos componentes es el siguiente:

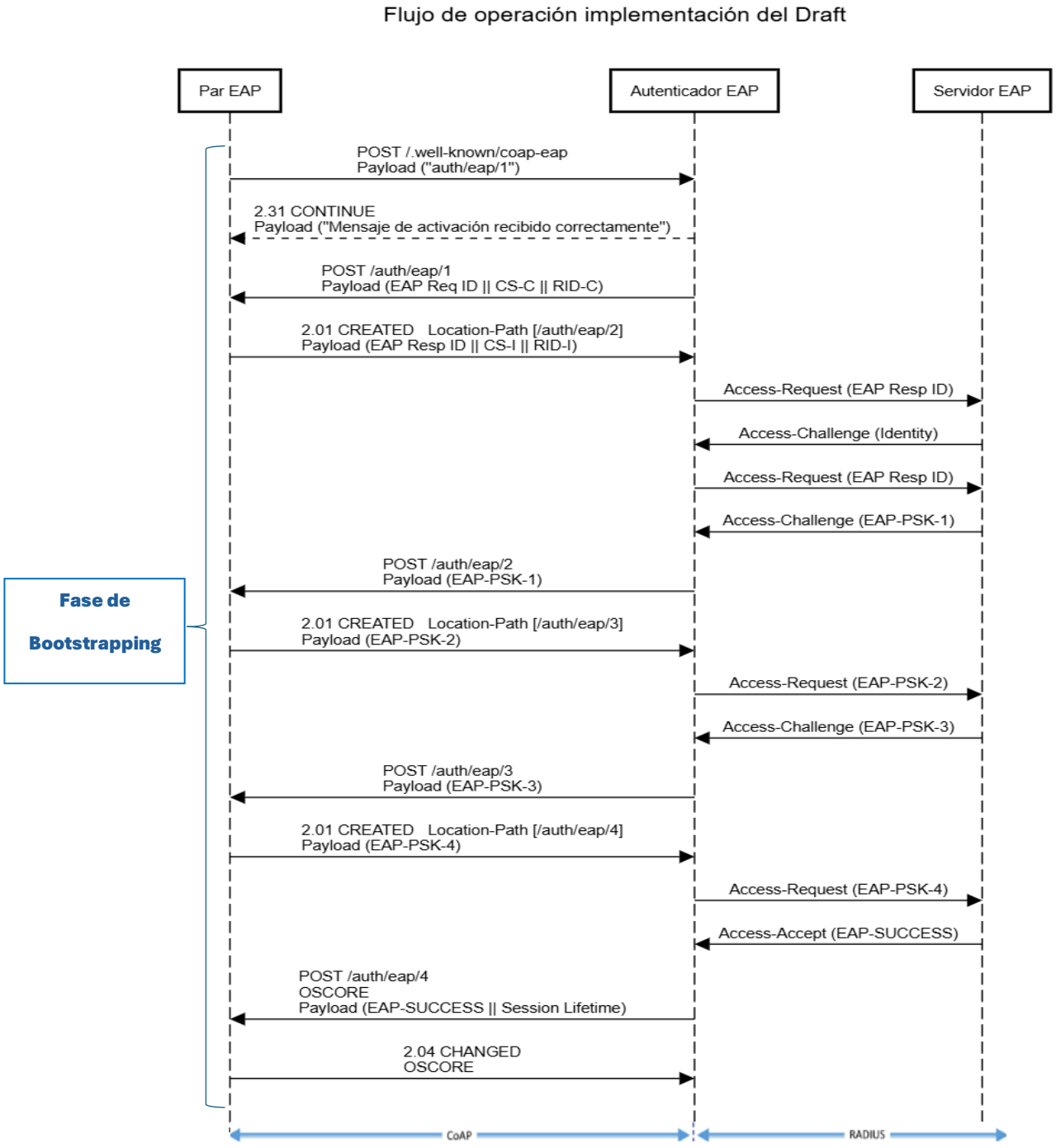


Figura 6.2.- MSC del flujo de operación implementado

A continuación, se comentarán las partes más relevantes del código desarrollado para el EAP Peer, el EAP Authenticator (incluyendo el cliente RADIUS) y el EAP Server y, a su vez, se justificarán los motivos por los que el aspecto del flujo de operación y algunos detalles difieren ligeramente de los comentados en la parte teórica.

6.3.1.- EAP Peer

El EAP Peer sólo actúa como un cliente para enviar el mensaje de activación (mensaje POST a la URI /.well-known/coap-eap), por lo que lo primero que hago dentro del main del script peer.py es crear un contexto de cliente CoAP para que pueda enviar ese mensaje al EAP Authenticator, definir el formato del contenido (content_format) de este primer mensaje (será application/link-format) y crear el payload de dicho mensaje. Este payload es una URI que sirve para indicar a qué recurso (‘auth/eap/counter’) debe enviar el EAP Authenticator el primer mensaje con la autenticación EAP (EAP Request ID). ‘auth’ es la ruta del dispositivo IoT elegida para hacer que la URI sea única y que esté relacionada con la autenticación, ‘eap’ indica que es una URI para una solicitud EAP y ‘counter’ es un número único aleatorio entre 1 y 100 que se incrementará con cada nueva solicitud EAP. Vamos a suponer que el valor aleatorio de counter obtenido es 1, por lo que el payload sería “auth/eap/1”.

```python
async def main():
    # El par EAP unicamente actuara como cliente para enviar el primer mensaje a la uri ".well-known/coap-eap"
    # Creo un contexto de cliente CoAP que me permitira enviar peticiones CoAP al controller
    context = await Context.create_client_context()
    # Establezco el formato del primer mensaje a LINKFORMAT (cumpliendo con lo establecido en el Draft)
    content_format = ContentFormat.LINKFORMAT
    # Genero un numero entero aleatorio entre 1 y 100
    counter = random.randint( a: 1,  b: 100)
    # Creo una URI concatenando el valor aleatorio generado con 'auth/eap/'. 'auth' es la ruta local del dispositivo IoT
    # que hace que la ruta sea unica, 'eap' indica que la URI es para el par EAP y counter es un numero unico que se
    # incrementara con cada nueva solicitud EAP
    URI_aleatoria = 'auth/eap/' + str(counter)
    # Imprimo la URI generada para ver el resultado
    print(URI_aleatoria)
    # Divido la URI en partes separadas por el caracter '/'
    URI_aleatoria_partes = URI_aleatoria.split('/')
    # El payload en este primer mensaje será la URI creada en el paso anterior
    payload_hex = URI_aleatoria.encode('utf-8').hex()
    # Imprimo el payload del mensaje well-known/coap-eap
    print("Payload del mensaje well-known/coap-eap:", payload_hex)
    # Convierto a bytes el payload
    payload = bytes.fromhex(payload_hex)
```

Figura 6.3.- Main del script peer.py parte 1

A continuación, construyo el mensaje CoAP de activación con código POST, con el content-format LINKFORMAT y el payload (“auth/eap/1”) definidos anteriormente, y con la uri "coap://" + wifi + "/.well-known/coap-eap", donde wifi es la dirección IPv4 del Adaptador de LAN inalámbrica Wi-Fi 2. Esta dirección IPv4 se emplea para vincular el mensaje con el contexto de servidor del EAP Authenticator, ya que escucha en esa dirección

y se obtiene llamando a la función get_wipi_ip del script redes.py implementada por mí. También asigno la opción No_Response a este mensaje para que no sea necesario recibir una respuesta, ya que como comenté en la parte teórica, el mensaje de activación no requiere una respuesta necesariamente. Sin embargo, debido a cómo está implementada la biblioteca aiocoap, es necesario que toda petición reciba una respuesta.

Una vez que envío el mensaje de activación y recibo la respuesta del EAP Authenticator, el EAP Peer dejará de comportarse como un cliente y comenzará a actuar como un servidor hasta que se complete la autenticación. De modo que lo último que hago dentro del main del script peer.py es crear un contexto de servidor CoAP, el cual permanece a la espera de recibir peticiones del EAP Authenticator en la dirección localhost y en el puerto 5683. Además, a este contexto de servidor le paso un objeto de tipo Site llamado root1, que actúa como el contenedor principal para los recursos del servidor CoAP del EAP Peer. A este objeto root le agrego un recurso (FirstResource) en la ruta '/auth/eap/1', para que cuando el EAP Authenticator realice una petición POST a ese recurso (será el mensaje EAP Request ID), el EAP Peer use la lógica implementada en la clase FirstResource para procesar esa petición y responderla.

```
    # Construyo el mensaje de solicitud con codigo POST a la uri indicada, con el payload y el content_format establecidos anteriormente
    # La uri indicada contiene la direccion IPv4 del Adaptador de LAN inalambrica Wi-Fi 2, que se obtiene llamando a la
    # funcion get_wipi_ip del script redes.py, para vincularlo con el contexto del servidor del controller ya que
    # escucha en esa direccion
    wifi = get_wifi_ip()
    request = Message(code=Code.POST, payload=payload, uri="coap://" + wifi + "/.well-known/coap-eap", content_format=content_format)
    # Marco la opción no_response a true para que no sea necesario recibir una respuesta a este primer mensaje
    # (por motivos de la implementacion de la biblioteca es necesario de todas formas que se reciba una respuesta)
    request.opt.no_response = True
    # Indico que voy a enviar el mensaje well-known/coap-eap al autenticador EAP
    print("Enviando mensaje well-known/coap-eap al controlador...")
    # Recibo la respuesta del controlador a este mensaje
    response = not await context.request(request).response
    # Creo el recurso raiz del servidor CoAP. Este objeto es el que luego se pasa a create_server_context para manejar las solicitudes que lleguen al servidor
    root1 = resource.Site()
    # Agrego un recurso (FirstResource) al servidor CoAP en la ruta especificada por la URI aleatoria. Cuando un cliente
    # CoAP realice una peticion a esa URI el servidor usara la logica implementada en FirstResource para procesar la
    # solicitud y responderla
    root1.add_resource( path: [URI_aleatoria_partes[0], URI_aleatoria_partes[1], URI_aleatoria_partes[2]], FirstResource(root1, URI_aleatoria, counter))
    # A partir de este momento el par EAP comenzara a comportarse como un servidor
    # Creo un contexto de servidor CoAP que esta configurando para escuchar en la direccion 'localhost' y el puerto 5683
    # Escuchara las peticiones enviadas por el autenticador EAP (cuando este actua como cliente) al par EAP (cuando este actua como servidor)
    await aiocoap.Context.create_server_context(root1, bind=('localhost', 5683))
    # Bucle infinito que permanece a la espera de recibir peticiones del EAP Controller
    await asyncio.get_running_loop().create_future()

if __name__ == "__main__":
    # Ejecuto el main del peer
    asyncio.run(main())
```

Figura 6.4.- Main del script peer.py parte 2

La clase FirstResource maneja el mensaje EAP Request ID enviado por el EAP Authenticator y genera como respuesta el mensaje EAP Response ID. Consta de un método constructor que recibe tres parámetros: root (root1), uri_to_remove (‘auth/eap/1’) y counter (1) y los asigna a los atributos de instancia self.root, self.uri_to_remove y self.counter respectivamente. De esta forma se define el estado inicial del constructor de esta clase.

```
class FirstResource(resource.Resource):
    # Defino el metodo constructor __init__, que recibe tres parametros: root, uri_to_remove y counter
    def __init__(self, root, uri_to_remove, counter):
        # Llamo al constructor de la clase base para asegurarme de que toda inicializacion en la clase padre 'Resource'
        # se realice correctamente.
        super().__init__()
        # Asigno el recurso raiz 'root' recibido como parametro al atributo de instancia 'self.root'.
        # 'root' representa el recurso principal que actua como contenedor para acceder a todos los recursos en el
        # servidor CoAP.
        self.root = root
        # Asigno la URI que se desea eliminar ('uri_to_remove') recibida como parametro al atributo de instancia
        # 'self.uri_to_remove'. Esta URI representa el recurso especifico que se espera eliminar en el contexto de esta
        # instancia.
        self.uri_to_remove = uri_to_remove
        # Asigno el valor del counter recibido como parametro (counter perteneciente a la URI del EAP Request ID) al
        # atributo de instancia 'self.counter'
        self.counter = counter
```

Figura 6.5.- Constructor de la clase FirstResource del script peer.py

La clase FirstResource también consta de un método render_post que se encarga de manejar todas las solicitudes CoAP con código POST enviadas por el EAP Authenticator a este primer recurso (auth/eap/1). Lo primero que hago en este método es mostrar por consola la información más importante (código y payload) relativa al mensaje EAP Request ID enviado por el EAP Authenticator, guardar el payload de este mensaje en la instancia root para que sea accesible desde la clase FourthResource, donde lo voy a tener que utilizar para construir la cadena CS consistente en la concatenación del contenido de la negociación de cipher suite y deserializar la estructura CBOR del payload del EAP Request ID recibido para extraer la información que necesito para OSCORE, en este caso el RID-C (Recipient ID del EAP Authenticator).

```python
async def render_post(self, request):
    # Muestro la informacion del mensaje POST recibido con el EAP Request ID
    print("MENSAJE POST CON EL EAP REQUEST ID RECIBIDO CON:")
    # Imprimo el codigo
    print("Código:", request.code)
    # Imprimo el payload en binario
    print("Payload:", request.payload)
    # Imprimo el payload en hexadecimal
    print("Payload hexadecimal:", request.payload.hex())
    # Guardo el payload del mensaje EAP Request ID en la instancia root para que sea accesible desde la clase FourthResource
    self.root.first_resource_payload = request.payload
    # Deserializo la estructura CBOR del payload del EAP Request ID recibido (son los ultimos 11 bytes) para extraer la
    # informacion que necesito para OSCORE
    deserialized_info1 = cbor2.loads(request.payload[-11:])
    # Extraigo el RID_C y lo asigno a la variable rid_c
    rid_c = deserialized_info1[2]
```

Figura 6.6.- Método render_post de la clase FirstResource del script peer.py parte 1

A continuación, comienzo a obtener todos los parámetros necesarios para construir la respuesta (EAP Response ID). Para ello, construyo la estructura CBOR que forma parte del payload del mensaje EAP Response ID, la cual está compuesta por dos campos. El primer campo corresponde al índice de la cipher suite elegida por el EAP Peer; en este caso, he elegido el índice 0, que representa la cipher suite predeterminada compuesta por AES-CCM-16-64-128 como algoritmo AEAD y SHA-256 como función hash. El segundo campo es el RID-I (Recipient ID del EAP Peer), el cual he decidido definir manualmente con un tamaño de 1 byte. Cabe destacar que para obtener ambos IDs (RID-C y RID-I) de la forma recomendada (Sender ID = HKDF (MSK, ”OSCORE RECIPIENT ID”, length) y Recipient ID = HKDF (MSK, ”OSCORE SENDER ID”, length)), como ya comenté en la parte de los fundamentos teóricos, se necesita la MSK, la cual se obtiene en el paso 7 del flujo de operación. No obstante, el uso de ambos IDs se requiere mucho antes, es decir, en los pasos 2 y 3 para ser enviados en la estructura CBOR del EAP Request ID y del EAP Response ID, por lo que hacerlo correctamente conllevaría un lógica bastante compleja que queda fuera del alcance de este TFG.

Una vez que ya tengo serializada la información en la estructura CBOR, la guardo en la instancia root para que sea accesible desde la clase FourthResource, donde la voy a utilizar también para obtener la cadena CS, y establezco el payload de la respuesta EAP Response ID formado por la concatenación de una cadena fija (por ejemplo "0237000a01", obtenida

de un ejemplo de paquete EAP Response ID en una traza de Wireshark), el NAI del dispositivo IoT (“usera”) y la estructura CBOR.

```
# Defino los conjuntos de cifrado en un diccionario, donde la clave es un indice y el valor es una lista que
# contiene el algoritmo de cifrado y el algoritmo de hash
conjuntos_cifrado = {
    0: ["AES-CCM-16-64-128", "SHA-256"],  # Este es el conjunto de cifrado predeterminado
    1: ["A128GCM", "SHA-256"],  # Este es el segundo conjunto de cifrado
    2: ["A256GCM", "SHA-384"],  # Este es el tercer conjunto de cifrado
    3: ["ChaCha20/Poly1305", "SHA-256"],  # Este es el cuarto conjunto de cifrado
    4: ["ChaCha20/Poly1305", "SHAKE256"]  # Este es el quinto conjunto de cifrado
}
# Defino de forma manual y con tamaño de 1 byte el ID de Destinatario del par EAP
RID_I = b'\x02'
# Defino la estructura CoAP-EAP_Info como un diccionario que contiene informacion sobre el conjuntos de
# cifrado elegido por el par (en este caso el predeterminado, es decir, 0) y el ID de Destinatario del par EAP
coap_eap_info = {
    # Indice del conjunto de cifrado elegido por el par EAP
    1: [0],
    # RID-I como una cadena binaria
    2: RID_I,
}
# Serializo la estructura CoAP-EAP_Info a formato CBOR para su transporte o almacenamiento
cbor_payload = cbor2.dumps(coap_eap_info)
# Guardo la estructura de datos CBOR en la instancia root para que sea accesible desde la clase FourthResource
self.root.cbor_payload = cbor_payload
# Establezco el payload de la respuesta (cadena hexadecimal fija + identidad del dispositivo IoT ("usera")
# codificado a hexadecimal + estructura de datos CBOR codificada a hexadecimal)
payload = "0237000a01" + "usera".encode('utf-8').hex() + cbor_payload.hex()
# Imprimo el payload de la respuesta en hexadecimal
print("Payload del EAP Response ID:", payload)
```

Figura 6.7.- Método render_post de la clase FirstResource del script peer.py parte 2

Deserializo la estructura CBOR del payload del EAP Response ID para extraer el resto de la información que necesito para OSCORE, en este caso el algoritmo EAED y la función HASH de la cipher suite elegida por el EAP Peer y el RID-I. Imprimo toda esta información por consola y la guardo en la instancia root para que sea accesible desde la clase FourthResource, ya que la necesitaré cuando cree el contexto de seguridad de OSCORE (el RID-C será el Sender ID, el RID-I será el Recipient ID y la cipher suite elegida por el EAP Peer será el algoritmo AEAD y el algoritmo HKDF utilizados por el contexto de seguridad OSCORE del EAP Peer).

```
# Deserializo la estructura CBOR del payload del EAP Response ID para extraer la informacion que necesito para OSCORE
deserialized_info2 = cbor2.loads(cbor_payload)
# Extraigo la lista de indices de los conjuntos de cifrado y la asigno a la variable cifrado_indices
cifrado_indices = deserialized_info2[1]
# Extraigo el primer indice de la lista de conjuntos de cifrado (en este caso el indice 0)
indice_cifrado = cifrado_indices[0]
# Asigno el nombre del algoritmo de cifrado AEAD correspondiente al indice extraido
algoritmo_AEAD = conjuntos_cifrado[indice_cifrado][0]
# Asigno el nombre del algoritmo hash correspondiente al indice extraido
algoritmo_hash = conjuntos_cifrado[indice_cifrado][1]
# Imprimo el nombre del algoritmo de cifrado correspondiente al índice extraído
print("Algoritmo de cifrado AEAD elegido:", algoritmo_AEAD)
# Imprimo el nombre del algoritmo hash correspondiente al índice extraído
print("Algoritmo hash elegido:", algoritmo_hash)
# Imprimo el RID-C extraido (ID de Destinatario del autenticador EAP)
print("RID-C:", rid_c)
# Imprimo el RID-I (ID de Destinatario del par EAP)
print("RID-I:", RID_I)
# Guardo el algoritmo AEAD en la instancia root para que sea accesible desde la clase FourthResource
self.root.algoritmo_AEAD = algoritmo_AEAD
# Guardo el algoritmo hash en la instancia root para que sea accesible desde la clase FourthResource
self.root.algoritmo_hash = algoritmo_hash
# Guardo el RID_I en la instancia root para que sea accesible desde la clase FourthResource
self.root.rid_i = RID_I
# Guardo el RID_C en la instancia root para que sea accesible desde la clase FourthResource
self.root.rid_c = rid_c
```

Figura 6.8.- Método render_post de la clase FirstResource del script peer.py parte 3

Después, establezco el content format a OCTETSTREAM (application/octet-stream) para que se muestre la carga útil del mensaje (payload) en Wireshark en hexadecimal. Teóricamente el content format debería de ser COAP-EAP (application/coap-eap, esta opción la añadí dentro del script contentformat.py de la biblioteca aiocoap de forma que se pueda seleccionar también si se desea) pero al ser muy reciente, Wireshark no la tiene implementada y por tanto no muestra el contenido de los mensajes ya que no reconoce este formato. Tras esto, construyo el mensaje de respuesta con código CREATED y con el payload y el content-format que acabo de establecer, creo la nueva URI (‘auth/eap/2’) y la añado a la opción Location_Path del mensaje de respuesta. Esta nueva URI será la que utilice el EAP Authenticator para enviar la petición POST con el primer mensaje del método EAP elegido por el EAP Server (EAP-PSK-1), en este caso será el método EAP-PSK. Seguidamente, al atributo de instancia self.root le agrego un recurso (SecondResource) en la ruta ‘/auth/eap/2’, para que cuando el EAP Authenticator realice una petición POST a esa URI (será el mensaje EAP-PSK-1), el EAP Peer usará la lógica implementada en SecondResource para procesar esa petición y responderla. Finalmente, envío el mensaje EAP Response ID al EAP Authenticator y elimino el recurso actual (auth/eap/1).

```
content = ContentFormat.OCTETSTREAM
# Construyo el mensaje de respuesta con codigo CREATED y con el payload y el content_format establecidos anteriormente
resp = aiocoap.Message(code=Code.CREATED, payload=bytes.fromhex(payload), content_format=content)
# Incremento el valor de la variable counter en 1
counter = self.counter+1
# Creo una nueva URI concatenando 'auth/eap/' con el nuevo valor de counter (sera la que se utilizara para enviar
# el POST con el mensaje EAP-PSK-1)
URI = 'auth/eap/' + str(counter)
# Divido la nueva URI en partes separadas por el caracter '/'
URI_partes = URI.split('/')
# Añado la URI creada a la opcion location_path del mensaje de respuesta
resp.opt.location_path = [URI_partes[0], URI_partes[1], URI_partes[2]]
# Indico que estoy enviando la respuesta al autenticador EAP
print("Enviando EAP Response ID al controlador...")
# Asocio la URI creada anteriormente con un recurso (SecondResource) que manejara las peticiones que lleguen a esa ruta
self.root.add_resource([URI_partes[0], URI_partes[1], URI_partes[2]], SecondResource(self.root, URI, counter))
# Indico que se ha agregado un nuevo recurso asociado a la uri generada
print("Nuevo recurso agregado con URI:", URI)
# Divido la URI que se necesita eliminar en partes separadas por el caracter '/'
uri_to_remove_partes = self.uri_to_remove.split('/')
# Indico que se va a eliminar el recurso FirstResource una vez que envio la respuesta y añado el nuevo recurso
print(f"Eliminando recurso con URI: {uri_to_remove_partes[0] + '/' + uri_to_remove_partes[1] + '/' + uri_to_remove_partes[2]}")
# Elimino el recurso con la uri asociada a este primer recurso (FirstResource)
self.root.remove_resource([uri_to_remove_partes[0], uri_to_remove_partes[1], uri_to_remove_partes[2]])
print("")
# Retorno el mensaje de respuesta
return resp
```

Figura 6.9.- Método render_post de la clase FirstResource del script peer.py parte 4

La clase SecondResource maneja la solicitud POST que contiene el primer mensaje del método EAP (EAP-PSK-1), transportado en la carga útil del mensaje CoAP enviado por el EAP Authenticator y genera un mensaje de respuesta con código CREATED que contiene el segundo mensaje del método EAP (EAP-PSK-2), que irá transportado en la carga útil del mensaje CoAP enviado por el EAP Peer. Consta de un método constructor que recibe tres parámetros: root (self.root de la clase FirstResource), uri_to_remove (‘auth/eap/2’) y counter (2), y los asigna a los atributos de instancia self.root, self.uri_to_remove y self.counter respectivamente. De esta forma se define el estado inicial del constructor de esta clase.

```python
class SecondResource(resource.Resource):
    # Defino el metodo constructor __init__, que recibe tres parametros: root, uri_to_remove y counter
    def __init__(self, root, uri_to_remove, counter):
        # Llamo al constructor de la clase base para asegurarme de que toda inicializacion en la clase padre 'Resource'
        # se realice correctamente.
        super().__init__()
        # Asigno el recurso raiz 'root' recibido como parametro al atributo de instancia 'self.root'.
        # 'root' representa el recurso principal que actua como contenedor para acceder a todos los recursos en el
        # servidor CoAP.
        self.root = root
        # Asigno la URI que se desea eliminar ('uri_to_remove') recibida como parametro al atributo de instancia
        # 'self.uri_to_remove'. Esta URI representa el recurso especifico que se espera eliminar en el contexto de esta
        # instancia.
        self.uri_to_remove = uri_to_remove
        # Asigno el valor del counter recibido como parametro (counter perteneciente a la URI del EAP-PSK-1) al
        # atributo de instancia 'self.counter'
        self.counter = counter
```

Figura 6.10.- Constructor de la clase SecondResource del script peer.py

La clase SecondResource también consta de un método render_post que se encarga de manejar todas las solicitudes CoAP con código POST enviadas por el EAP Authenticator a este segundo recurso (auth/eap/2). Lo primero que hago en este método es mostrar por consola la información más importante (código y payload) relativa al mensaje EAP-PSK-1 enviado por el EAP Authenticator y definir como variables globales la KDK, la AK, el ID_S y el RAND_P (las inicialicé a nivel de módulo como cadenas vacías tras los imports y antes de la clase FirstResource).

```python
from redes import get_wifi_ip

# Inicializo cuatro variables globales como cadenas vacias. Al estar definidas fuera de cualquier clase o funcion, estas
# variables pueden ser accedidas y modificadas desde cualquier parte del modulo en el que estan definidas. Esto sera
# util mas adelante, ya que los metodos de las clases usaran estas variables para compartir informacion entre diferentes
# partes del programa.
AK = ""
KDK = ""
ID_S = ""
RAND_P = ""

# Clase FirstResource que manejara el EAP Request ID y generara la EAP Response ID
1 usage
class FirstResource(resource.Resource):
```

Figura 6.11.- Definición de las variables globales a nivel de módulo en el script peer.py

Al estar definidas fuera de cualquier clase o función, estas variables pueden ser accedidas y modificadas desde cualquier parte del módulo en el que están definidas. En este caso actualizaremos su valor dentro de este método render_post, es decir, dejarán de ser cadenas vacías ya que la AK y la KDK tomarán el valor obtenido tras derivar la PSK en esta

línea de código: AK, KDK = eap_psk_key_setup (bytes.fromhex(PSK)), el ID_S tomará el valor obtenido del parseo del mensaje EAP-PSK-1 en esta línea de código: ID_1, T_1, RAND_S, ID_S = parser_mensaje1(EAP_PSK_MSG_1) y el RAND_P tomará un valor aleatorio de 16 bytes calculado en esta línea de código: RAND_P = os.urandom(16).hex().

Como el mensaje EAP-PSK-1 va transportado en la carga útil de la petición CoAP recibida con código POST, asigno la variable EAP_PSK_1 al payload del mensaje enviado por el EAP Authenticator. Además, compruebo que la cabecera de este mensaje EAP-PSK-1 es correcta mediante la función eap_hdr_validate presente en el script mensajes_EAP_PSK.py. Un dato importante a destacar es que todas las funciones que se van a utilizar en estas clases para construir y validar los mensajes EAP-PSK se encuentran en el script mensajes_EAP_PSK.py y han sido implementadas completamente por mí.

```
async def render_post(self, request):
    # Defino como variables globales la KDK, la AK, el ID_S y el RAND_P para almacenar sus valores
    global AK, KDK, ID_S, RAND_P
    # Muestro la información del mensaje POST recibido con el EAP-PSK-1
    print("MENSAJE POST CON EL EAP-PSK-1 RECIBIDO CON:")
    # Imprimo el codigo
    print("Código:", request.code)
    # Imprimo el payload en binario
    print("Payload:", request.payload)
    # Imprimo el payload en hexadecimal
    print("Payload hexadecimal:", request.payload.hex())
    # Asigno a la variable EAP_PSK_MSG_1 el valor hexadecimal del payload del mensaje recibido
    EAP_PSK_MSG_1 = request.payload.hex()
    # Compruebo que la cabecera del mensaje EAP-PSK-1 sea correcta
    payload1, payload_len1 = eap_hdr_validate(EAP_VENDOR_IETF, EAP_PSK_TYPE, binascii.unhexlify(EAP_PSK_MSG_1))
    # Si es correcta muestro su payload en hexadecimal (datos a partir de la cabecera) y el tamaño de su payload
    if payload1 is not None:
        print("Cabecera EAP correcta para el mensaje EAP-PSK-1!")
        print("Payload EAP del primer mensaje:", payload1.hex())
        print("Longitud del primer mensaje EAP:", payload_len1)
    else:
        print("Cabecera EAP invalida para el mensaje EAP-PSK-1")
```

Figura 6.12.- Método render_post de la clase SecondResource del script peer.py parte 1

A continuación, parseo el mensaje EAP-PSK-1 para obtener los campos que serán necesarios para construir el mensaje EAP-PSK-2. Entre estos campos se encuentran el Message ID (ID_1), el subcampo T del Flag que indica el tipo de mensaje (T_1), el desafío aleatorio de 16 bytes enviado por el servidor (RAND_S) y la identidad del servidor (ID_S); los imprimo por consola y compruebo que ninguno de estos valores es NULL. Posteriormente compruebo que el Flag es correcto, para ello el subcampo T del Flag debería

de ser 0 para el primer mensaje EAP-PSK y llevo a cabo la parte criptográfica relativa a la configuración de claves, en la cual derivo la AK y la KDK a partir de la PSK, definida en el script mensajes_EAP_PSK.py, mediante la función eap_psk_key_setup.

```
# Parseo el primer mensaje EAP-PSK para obtener los campos que me interesan (el ID, el flag, el RAND_S y el ID_S)
ID_1, T_1, RAND_S, ID_S = parser_mensaje1(EAP_PSK_MSG_1)
print("")
# Imprimo los valores devueltos por la funcion parser_mensaje1
print("Valores resultantes de parsear el mensaje EAP-PSK-1:")
print("ID_1:", ID_1)
print("T_1:", T_1)
print("RAND_S:", RAND_S)
print("ID_S:", ID_S)
# Si alguna variable es None, imprimo un mensaje de error y regreso
if None in (ID_1, T_1, RAND_S, ID_S):
    print("Error: Una o mas variables devueltas son None.")
    return
# Compruebo que el flag es el correcto (T1 debe de ser igual a 0)
if T_1 != 0:
    print("El flag recibido es incorrecto, se esperaba que T=0")
else:
    print("Flag correcto, se trata del primer mensaje EAP-PSK")
# Derivo la PSK para obtener la AK y la KDK
AK, KDK = eap_psk_key_setup(bytes.fromhex(PSK))
# Imprimo la AK y la KDK
print("AK:", AK)
print("KDK:", KDK)
```

Figura 6.13.- Método render_post de la clase SecondResource del script peer.py parte 2

Una vez hecho esto, comienzo a construir el segundo mensaje EAP-PSK. Para ello defino el campo EAP Code (2 ya que es una respuesta), el campo Message ID (mismo que el del primer mensaje EAP-PSK) y el campo Type (47 para EAP-PSK), establezco el Flag (en hexadecimal “40” y en binario “01000000”, ya que los dos primeros bits corresponden al subcampo T y los seis siguientes bits corresponden al subcampo Reserved y se ponen todos a 0 en la transmisión) con la función EAP_PSK_FLAGS_SET_T, sabiendo que el subcampo T del Flag para el segundo mensaje EAP-PSK tiene que ser 1 y calculo el desafío aleatorio de 16 bytes enviado por el par (RAND_P).

```
# Defino el codigo que va a ser 2 ya que es un respuesta
Code_2 = "02"
# Defino el ID que va a ser el mismo que el del primer mensaje EAP-PSK
ID_2 = ID_1
# Defino el tipo que va a tener el mensaje (EAP-PSK) y elimino el prefijo "0x" que se añade automaticamente en Python
Type = hex(EAP_PSK_TYPE)[2:]
# Defino el flag de este mensaje que debe ser 1
T_2 = 1
# Establezco ese flag empleando la función EAP_PSK_FLAGS_SET_T
set_t_result_2 = EAP_PSK_FLAGS_SET_T(T_2)
# Convierto a hexadecimal el flag y elimino el prefijo "0x" que se añade automaticamente en Python
Flag_2 = hex(set_t_result_2)[2:]
# Imprimo el flag para que tras el paso anterior se muestre en este caso así (T_2:40)
print("Flag_2:", Flag_2)
# Calculo el RAND_P en hexadecimal de forma aleatoria con un tamaño de 16 bytes
RAND_P = os.urandom(16).hex()
# Imprimo el RAND_P
print("RAND_P:", RAND_P)
```

Figura 6.14.- Método render_post de la clase SecondResource del script peer.py parte 3

Seguidamente, defino el identificador del EAP Peer (“client”), calculo la MAC del EAP Peer (**MAC_P = CMAC-AES-128(AK, ID_P||ID_S||RAND_S||RAND_P)**) mediante la función omac1_aes_128 a la que se le pasan como parámetros la AK y los campos ID_P, ID_S, RAND_S y RAND_P concatenados. Tras esto, obtengo el tamaño total del mensaje (campo Length de la cabecera del mensaje EAP), que es el único campo que me falta, mediante la función calcular_tamano_total_hexadecimal. Esta función suma el tamaño de todos los campos definidos anteriormente que componen el segundo mensaje EAP-PSK (EAP Code, Message ID, Type, Flag, RAND_S, RAND_P, MAC_P e ID_P) más 2 bytes que es el tamaño del campo Length. Ahora que ya tengo todos los campos definidos, construyo el mensaje EAP-PSK-2 concatenando cada uno de esos campos en el orden correcto (EAP Code, Message ID, Length, Type, Flag, RAND_S, RAND_P, MAC_P e ID_P) y ese será el payload de la respuesta.

```
# Defino el ID_P (se que es "client") y lo convierto a hexadecimal
ID_P = binascii.hexlify("client".encode()).decode()
# Imprimo el ID_P
print("ID_P:", ID_P)
# Concateno todos los campos que compondran los datos necesarios para calcular la MAC_P
data_macp = bytes.fromhex(ID_P + ID_S + RAND_S + RAND_P)
# Calculo la MAC_P con la funcion omac1_aes_128 pasando como parametros la AK y los datos
MAC_P = omac1_aes_128(bytes.fromhex(AK), data_macp)
# Imprimo el MAC_P
print("MAC_P:", MAC_P)
# Obtengo el tamaño total del mensaje (campo Length de la cabecera de un mensaje EAP)
Length_2 = calcular_tamano_total_hexadecimal( *args: Code_2, ID_2, Type, Flag_2, RAND_S, RAND_P, MAC_P, ID_P)
# Imprimo el tamaño total del mensaje
print("Campo Length de EAP-PSK-2:", Length_2)
# Convierto Length_2 a hexadecimal y me aseguro de que ocupe 2 bytes (4 caracteres hexadecimales)
Length_2_hex = f'{Length_2:04x}'
print("Campo Length de EAP-PSK-2 hexadecimal:", Length_2_hex)
# Concateno todos los campos (en hexadecimal) para formar el mensaje completo EAP-PSK 2
EAP_PSK_MSG_2 = Code_2 + convertir_a_hexadecimal(ID_2) + Length_2_hex + Type + Flag_2 + RAND_S + RAND_P + MAC_P + ID_P
# Imprimo el mensaje EAP-PSK-2
print("Mensaje EAP-PSK-2:", EAP_PSK_MSG_2)
# Asigno el payload de la respuesta al valor del mensaje EAP-PSK-2 en hexadecimal
payload = EAP_PSK_MSG_2
```

Figura 6.15.- Método render_post de la clase SecondResource del script peer.py parte 4

Después, establezco el content format a OCTETSTREAM (debería de ser COAP-EAP al igual que en el EAP Response ID). Tras esto, construyo el mensaje de respuesta con código CREATED y con el payload y el content format que acabo de establecer, creo la nueva URI (‘auth/eap/3’), que será la que utilice el EAP Authenticator para enviar la petición POST con el tercer mensaje del método EAP (EAP-PSK-3) y la añado a la opción Location_Path del mensaje de respuesta. Seguidamente, al atributo de instancia self.root le agrego un recurso (ThirdResource) en la ruta ‘/auth/eap/3’, para que cuando el EAP Authenticator realice una petición POST a esa URI (será el mensaje EAP-PSK-3), el EAP Peer usará la lógica implementada en ThirdResource para procesar esa petición y responderla. Finalmente, envío el mensaje de respuesta con el código CREATED que contiene el segundo mensaje del método EAP (EAP-PSK-2) al EAP Authenticator y elimino el recurso actual (auth/eap/2).

```
# Establezco el formato del contenido a OCTETSTREAM (deberia de ser COAP-EAP)
content = ContentFormat.OCTETSTREAM
# Construyo el mensaje de respuesta con codigo CREATED y con el payload y el content_format establecidos anteriormente
resp = aiocoap.Message(code=Code.CREATED, payload=bytes.fromhex(payload), content_format=content)
# Incremento el valor de la variable counter en 1
counter = self.counter + 1
# Creo una nueva URI concatenando 'auth/eap/' con el nuevo valor de counter (sera la que se utilizara para enviar
# el POST con el mensaje EAP-PSK-3)
URI = 'auth/eap/' + str(counter)
# Divido la nueva URI en partes separadas por el caracter '/'
URI_partes = URI.split('/')
# Añado la URI creada a la opcion location_path del mensaje de respuesta
resp.opt.location_path = [URI_partes[0], URI_partes[1], URI_partes[2]]
# Indico que estoy enviando la respuesta al autenticador EAP
print("Enviando respuesta con el segundo mensaje del metodo EAP (EAP-PSK-2) al controlador...")
# Asocio la URI creada anteriormente con un recurso (ThirdResource) que manejara las peticiones que lleguen a esa ruta
self.root.add_resource([URI_partes[0], URI_partes[1], URI_partes[2]], ThirdResource(self.root, URI, counter))
# Indico que se ha agregado un nuevo recurso asociado a la uri generada
print("Nuevo recurso agregado con URI:", URI)
# Divido la URI que se necesita eliminar en partes separadas por el caracter '/'
uri_to_remove_partes = self.uri_to_remove.split('/')
# Indico que se va a eliminar el recurso SecondResource una vez que envio la respuesta y añado el nuevo recurso
print(f"Eliminando recurso con URI: {uri_to_remove_partes[0] + '/' + uri_to_remove_partes[1] + '/' + uri_to_remove_partes[2]}")
# Elimino el recurso con la uri asociada a este segundo recurso (SecondResource)
self.root.remove_resource([uri_to_remove_partes[0], uri_to_remove_partes[1], uri_to_remove_partes[2]])
print("")
# Retorno el mensaje de respuesta
return resp
```

Figura 6.16.- Método render_post de la clase SecondResource del script peer.py parte 5

La clase ThirdResource maneja la solicitud POST que contiene el tercer mensaje del método EAP (EAP-PSK-3), transportado en la carga útil del mensaje CoAP enviado por el EAP Authenticator y genera un mensaje de respuesta con código CREATED que contiene el cuarto mensaje del método EAP (EAP-PSK-4), que irá transportado en la carga útil del mensaje CoAP enviado por el EAP Peer. Consta de un método constructor que recibe tres parámetros: root (self.root de la clase SecondResource), uri_to_remove (‘auth/eap/3’) y counter (3), y los asigna a los atributos de instancia self.root, self.uri_to_remove y self.counter respectivamente. De esta forma se define el estado inicial del constructor de esta clase.

```python
class ThirdResource(resource.Resource):
    # Defino el metodo constructor __init__, que recibe tres parametros: root, uri_to_remove y counter
    def __init__(self, root, uri_to_remove, counter):
        # Llamo al constructor de la clase base para asegurarme de que toda inicializacion en la clase padre 'Resource'
        # se realice correctamente
        super().__init__()
        # Asigno el recurso raiz 'root' recibido como parametro al atributo de instancia 'self.root'.
        # 'root' representa el recurso principal que actua como contenedor para acceder a todos los recursos en el
        # servidor CoAP.
        self.root = root
        # Asigno la URI que se desea eliminar ('uri_to_remove') recibida como parametro al atributo de instancia
        # 'self.uri_to_remove'. Esta URI representa el recurso específico que se espera eliminar en el contexto de esta
        # instancia.
        self.uri_to_remove = uri_to_remove
        # Asigno el valor del counter recibido como parametro (counter perteneciente a la URI del EAP-PSK-3) al
        # atributo de instancia 'self.counter'
        self.counter = counter
```

Figura 6.17.- Constructor de la clase ThirdResource del script peer.py

La clase ThirdResource también consta de un método render_post que se encarga de manejar todas las solicitudes CoAP con código POST enviadas por el EAP Authenticator a este tercer recurso (auth/eap/3). Lo primero que hago en este método es mostrar por consola la información más importante (código y payload) relativa al mensaje EAP-PSK-3 enviado por el EAP Authenticator y definir como variables globales la KDK, la AK, el ID_S y el RAND_P para poder acceder y utilizar los valores que tomaron dichas variables en el método render_post de la clase SecondResource, ya que son necesarios para construir el mensaje EAP-PSK-4.

Como el mensaje EAP-PSK-3 va transportado en la carga útil de la petición CoAP recibida con código POST, asigno la variable EAP_PSK_3 al payload del mensaje enviado por el EAP Authenticator. Además, compruebo que la cabecera de este mensaje EAP-PSK-3 es correcta mediante la función eap_hdr_validate.

```python
async def render_post(self, request):
    # Variables globales KDK, AK, ID_S y RAND_P que me permiten utilizar sus valores almacenados
    global AK, KDK, ID_S, RAND_P
    # Muestro la informacion del mensaje POST recibido con el EAP-PSK-3
    print("MENSAJE POST CON EL EAP-PSK-3 RECIBIDO CON:")
    # Imprimo el codigo
    print("Código:", request.code)
    # Imprimo el payload en binario
    print("Payload:", request.payload)
    # Imprimo el payload en hexadecimal
    print("Payload hexadecimal:", request.payload.hex())
    # Asigno a la variable EAP_PSK_MSG_3 el valor hexadecimal del payload del mensaje recibido
    EAP_PSK_MSG_3 = request.payload.hex()
    # Compruebo que la cabecera del mensaje EAP-PSK-3 sea correcta
    payload3, payload_len3 = eap_hdr_validate(EAP_VENDOR_IETF, EAP_PSK_TYPE, bytes.fromhex(EAP_PSK_MSG_3))
    # Si es correcta muestro su payload en hexadecimal (datos a partir de la cabecera) y el tamaño de su payload
    if payload3 is not None:
        print("Cabecera EAP correcta para el mensaje EAP-PSK-3!")
        print("Payload EAP del tercer mensaje:", payload3.hex())
        print("Longitud del tercer mensaje EAP:", payload_len3)
    else:
        print("Cabecera EAP inválida para el mensaje EAP-PSK 3")
```

Figura 6.18.- Método render_post de la clase ThirdResource del script peer.py parte 1

A continuación, parseo el mensaje EAP-PSK-3 para obtener los campos que serán necesarios para construir el mensaje EAP-PSK-4. Entre estos campos se encuentran el Message ID (ID_3), el subcampo T del Flag que indica el tipo de mensaje (T_3), el desafío aleatorio de 16 bytes enviado por el servidor (RAND_S), la MAC del servidor (MAC_S), el canal protegido del server (Pchannel_Servidor), el campo Nonce del canal protegido del server (Nonce_Servidor), el campo Tag del canal protegido del server (Tag_Servidor) y el campo R del canal protegido del server (R_Servidor); los imprimo por consola y compruebo que ninguno de estos valores es NULL. Posteriormente, compruebo que el Flag es correcto, para ello el subcampo T del Flag debería de ser 2 para el segundo mensaje EAP-PSK y llevo a cabo la parte criptográfica relativa a la derivación de las claves de sesión, en la cual derivo la TEK, la MSK y la EMSK a partir del RAND_P y de la KDK, mediante la función eap_psk_derive_keys. Tras hacer esto, guardo la MSK que acabo de obtener en la instancia root para que sea accesible desde la clase FourthResource, donde la voy a tener que utilizar para derivar los parámetros Master Secret y Master Salt necesarios para generar el contexto de seguridad OSCORE del EAP Peer.

```
# Parseo el tercer mensaje EAP-PSK para obtener los campos que me interesan (el ID, el flag, el RAND_S,
# el MAC_S y el Pchannel completo, el nonce, el tag y el bit R del servidor)
ID_3, T_3, RAND_S, MAC_S, Pchannel_Servidor, Nonce_Servidor, Tag_Servidor, R_Servidor = parser_mensaje3(EAP_PSK_MSG_3)
# Imprimo los valores devueltos por la funcion parser_mensaje3
print("Valores resultantes de parsear el mensaje EAP-PSK-3:")
print("ID_3:", ID_3)
print("T_3:", T_3)
print("RAND_S:", RAND_S)
print("MAC_S:", MAC_S)
print("Pchannel_Servidor:", Pchannel_Servidor)
print("Nonce_Servidor:", Nonce_Servidor)
print("Tag_Servidor:", Tag_Servidor)
print("R_Servidor:", R_Servidor)
# Si alguna variable es None, imprimo un mensaje de error y regreso
if None in (ID_3, T_3, RAND_S, MAC_S, Pchannel_Servidor, Nonce_Servidor, Tag_Servidor, R_Servidor):
    print("Error: Una o mas variables devueltas son None.")
    return
# Compruebo que el flag es el correcto (T3 debe de ser igual a 2)
if T_3 != 2:
    print("El flag recibido es incorrecto, se esperaba que T=2")
else:
    print("Flag correcto, se trata del tercer mensaje EAP-PSK")
# Obtengo la TEK, la MSK y la EMSK a partir de la KDK y del RAND_P
TEK, MSK, EMSK = eap_psk_derive_keys(bytes.fromhex(KDK), bytes.fromhex(RAND_P))
print("TEK:", TEK)
print("MSK:", MSK)
print("EMSK:", EMSK)
# Guardo la MSK en la instancia root para que sea accesible desde la clase FourthResource
self.root.msk = MSK
```

Figura 6.19.- Método render_post de la clase ThirdResource del script peer.py parte 2

Seguidamente, calculo la MAC del servidor (**MAC_S = CMAC-AES-128(AK, ID_S||RAND_P)**) mediante la función omac1_aes_128, a la que se le pasan como parámetros la AK y los campos ID_S y RAND_P concatenados y la comparo con la MAC_S obtenida al parsear el mensaje EAP-PSK-3 para comprobar que coinciden y que, por tanto, es correcta. Continuo con las comprobaciones asegurándome de que la longitud del canal protegido del server es la adecuada, debería ser de 21 bytes.

```
# Concateno todos los campos que compondran los datos necesarios para calcular el MAC_S
data_macs = bytes.fromhex(ID_S + RAND_P)
# Calculo la MAC_P con la funcion omac1_aes_128 pasando como parametros la AK y los datos
MAC_S_Calculada = omac1_aes_128(bytes.fromhex(AK), data_macs)
# Imprimo el MAC_S
print("MAC_S:", MAC_S)
# Compruebo que la MAC_S calculada coincide con la recibida
if MAC_S_Calculada == MAC_S:
    print("La MAC_S calculada y la MAC_S recibida coinciden")
print("Pchannel Servidor:", Pchannel_Servidor)
# Compruebo que la longitud del Pchannel del servidor es la adecuada (21 bytes)
if len(bytes.fromhex(Pchannel_Servidor)) < 21:
    print("La longitud del Pchannel del servidor es incorrecta, es menor de 21 bytes")
else:
    print("La longitud del Pchannel del servidor es correcta, es de 21 bytes")
```

Figura 6.20.- Método render_post de la clase ThirdResource del script peer.py parte 3

La última comprobación que voy a hacer al mensaje EAP-PSK-3 va a ser calcular el canal protegido del server y comprobar que coincide con el obtenido al parsear el mensaje.

Para ello, lo primero que hago es conseguir los datos cifrados del canal protegido del server (es el último byte del EAP-PSK-3, ya que el EAP Server envía un sólo byte de información) y el campo Header (hdr3) que son los primeros 22 bytes del EAP-PSK-3, es decir, la concatenación de todos los campos previos a la MAC_S.

Una vez que tengo eso, utilizo la función aes_128_eax_decrypt, que recibe como parámetros la TEK, un Nonce (en este caso nonce3 de tamaño 16 bytes todos ellos a 0), la longitud de ese Nonce, el Header de 22 bytes del mensaje EAP-PSK-3 (hdr3), la longitud de ese Header, la carga útil cifrada (son los datos cifrados obtenidos del mensaje EAP-PSK-3 de 1 byte, es decir, data3), la longitud de esa carga útil cifrada y el Tag del canal protegido del server. Lo que hace esta función es devolver la carga útil en texto plano, es decir, los datos de 1 byte descifrados (decrypted_data) y comprobar que el Tag proporcionado coincide con el calculado dentro de la propia función. De modo que si alguno de los datos proporcionados es erróneo, los Tags no coincidirán y por tanto, no retornará los datos en texto plano.

Finalmente, miro a ver si el campo R del canal protegido del server es DONE_SUCCESS (lo que nos indicaría que la autenticación por el momento está siendo exitosa) o DONE_FAILURE (lo que nos indicaría que hubo un error y la autenticación falló) gracias a la función verificar_r_flag.

```
# Obtengo los datos cifrados que son el resultado de realizar el modo EAX (es el ultimo byte del mensaje EAP-PSK-3)
data3 = bytes.fromhex(EAP_PSK_MSG_3)[-1:]
# Imprimo en hexadecimal los datos cifrados del Pchannel del servidor
print("Datos cifrados Pchannel del servidor:", data3.hex())
# Obtengo el header que son los primeros 22 bytes del mensaje EAP-PSK 3 (todos los campos anteriores al MAC_S)
hdr3 = bytes.fromhex(EAP_PSK_MSG_3)[:22]
# Imprimo el header del mensaje EAP-PSK-3
print("Header EAP-PSK 3:", hdr3.hex())
# Obtengo los datos (descifrados) que se pasaron para crear el Pchannel del servidor y el tag desencriptando del
# Pchannel. Para ello defino un nonce que sera un array de bytes de tamaño 16 todos ellos a 0 necesario para los
# calculos posteriores
nonce3 = bytes(16)
# Paso a la funcion aes_128_eax_decrypt en bytes la TEK, el nonce, el header del EAP-PSK-3, los datos cifrados,
# el tag del Pchannel del servidor y las longitudes del nonce, del header y de los datos cifrados
decrypted_data = aes_128_eax_decrypt(bytes.fromhex(TEK), nonce3, len(nonce3), hdr3, len(hdr3), data3, len(data3),
                                     bytes.fromhex(Tag_Servidor))
# Si todos los parametros proporcionados son correctos, el tag del servidor pasado como parametro debera
# coincidir con el resultante tras ejecutarse la funcion previa
if decrypted_data == -2:
    print("El tag no coincide.")
else:
    # De modo que obtengo e imprimo en hexadecimal los datos descifrados
    print("Datos desencriptados:", decrypted_data.hex())
# Miro a ver si el campo R del Pchannel del servidor es un DONE_SUCCESS o un DONE_FAILURE
verificar_r_flag(decrypted_data.hex())
```

Figura 6.21.- Método render_post de la clase ThirdResource del script peer.py parte 4

Recordemos que el canal protegido cifra y autentica una carga útil en texto plano que se convierte en una carga útil cifrada usando el modo de funcionamiento EAX, al que se le tiene que pasar como parámetros de entrada un Nonce, un Header de 22 bytes, una carga útil de longitud variable en texto plano y la TEK. Este funcionamiento lo replico utilizando la función aes_128_eax_encrypt, que recibe como parámetros la TEK, el nonce3, la longitud de ese Nonce, el Header de 22 bytes del mensaje EAP-PSK-3 (hdr3), la longitud de ese Header, la carga útil en texto plano obtenida en el paso anterior (decrypted_data), la longitud de esa carga útil en texto plano y un Tag (tag3) de 16 bytes todos ellos a 0. Lo que hace esta función es devolver la carga útil cifrada y actualizar el valor del Tag proporcionado (tag3) con el calculado dentro de la propia función.

Ahora ya tengo todo lo necesario para construir el canal protegido del server, que está compuesto por la concatenación de los últimos 4 bytes del nonce3 (“00000000”), el tag3 y la carga útil cifrada obtenida en el paso anterior (encrypted_data_calc_serv).

```
# Por ultimo calculo el Pchannel del servidor
# Para ello defino un tag que será inicialmente un array de bytes de tamaño 16 todos ellos a 0
tag3 = bytearray(16)
# Paso a la funcion aes_128_eax_encrypt en bytes la TEK, el nonce, el header del EAP-PSK-3, los datos
# descifrados, el tag y las longitudes del nonce, del header y de los datos descifrados para así obtener los
# datos cifrados de forma calculada
encrypted_data_calc_serv = aes_128_eax_encrypt(bytes.fromhex(TEK), nonce3, len(nonce3), hdr3, len(hdr3),
                                               decrypted_data, len(decrypted_data), tag3)
# Construyo el Pchannel del servidor concatenando los últimos 4 bytes del nonce, los 16 bytes del tag resultante
# tras ejecutarse la funcion anterior y los datos encriptados
encrypted_pchannel_calc_serv = nonce3[-4:] + tag3 + encrypted_data_calc_serv
# Imprimo en hexadecimal los datos encriptados calculados que compondran el Pchannel del servidor
print("Datos encriptados del Pchannel del servidor calculados:", encrypted_data_calc_serv.hex())
# Imprimo en hexadecimal el tag calculado que compondra el Pchannel del servidor
print("Tag del Pchannel del servidor calculado:", tag3.hex())
# Imprimo en hexadecimal el Pchannel del servidor encriptado calculado
print("Pchannel del servidor encriptado calculado:", encrypted_pchannel_calc_serv.hex())
# Compruebo si el Pchannel del servidor calculado coincide con el recibido
if encrypted_pchannel_calc_serv.hex() == Pchannel_Servidor:
    print("El Pchannel del servidor recibido coincide con el calculado, es correcto!")
print("")
```

Figura 6.22.- Método render_post de la clase ThirdResource del script peer.py parte 5

Una vez hecho esto, comienzo a construir el cuarto mensaje EAP-PSK. Para ello defino el EAP Code (2 ya que es una respuesta), el Message ID (mismo que el del tercer mensaje), defino el Type (47 para EAP-PSK), establezco el Flag (en hexadecimal “c0” y en binario “11000000” ya que los dos primeros bits corresponden al subcampo T y los seis siguientes bits corresponden al subcampo Reserved y se ponen todos a 0 en la transmisión) con la función EAP_PSK_FLAGS_SET_T, sabiendo que el subcampo T del Flag para el cuarto mensaje EAP-PSK tiene que ser 3 y obtengo el tamaño total del mensaje (campo Length de la cabecera del mensaje) mediante la función calcular_tamano_total_hexadecimal que suma el tamaño de todos los campos que componen el cuarto mensaje EAP-PSK (EAP Code, Message ID, Type, Flag, RAND_S y canal protegido del peer (su tamaño debe ser el mismo que el del canal protegido del server, es decir, 21 bytes)) más 2 bytes que es el tamaño del campo Length.

```python
# Construcción del cuarto mensaje EAP-PSK
# Defino el código que va a ser 2 que que es el segundo mensaje
Code_4 = "02"
# Defino el ID que va a ser el mismo que el del tercer mensaje EAP-PSK
ID_4 = ID_3
# Defino el tipo que va a tener el mensaje (EAP-PSK) y elimino el prefijo "0x" que se añade automáticamente en Python
Type = hex(EAP_PSK_TYPE)[2:]
# Defino el flag de este mensaje que debe ser 3
T_4 = 3
# Establezco ese flag empleando la funcion EAP_PSK_FLAGS_SET_T
set_t_result_4 = EAP_PSK_FLAGS_SET_T(T_4)
# Convierto a hexadecimal el flag y elimino el prefijo "0x" que se añade automaticamente en Python
Flag_4 = hex(set_t_result_4)[2:]
print("Flag_4:", Flag_4)
# Obtengo el tamaño total del mensaje EAP-PSK 4 (tamaño de los campos de la cabecera (Code, ID y Type) + tamaño
# del Flag_4 + tamaño del RAND_S + tamaño del Pchannel del peer (21 bytes))
Length_4 = calcular_tamano_total_hexadecimal( *args: Code_4, ID_4, Type, Flag_4, RAND_S) + 21
# Imprimo el tamaño total del mensaje EAP-PSK-4
print("Campo Length de EAP-PSK-4:", Length_4)
# Convierto Length_4 a hexadecimal y me aseguro de que ocupe 2 bytes (4 caracteres hexadecimales)
Length_4_hex = f'{Length_4:04x}'
print("Campo Length de EAP-PSK-4 en hexadecimal:", Length_4_hex)
```

Figura 6.23.- Método render_post de la clase ThirdResource del script peer.py parte 6

A continuación, obtengo el campo Header (hdr4) que son los primeros 22 bytes del cuarto mensaje EAP-PSK, es decir, la concatenación de todos los campos previos al canal protegido del peer y compruebo si, efectivamente, el tamaño de los datos que componen la carga útil del canal protegido del peer es de 1 byte y, por tanto, el tamaño del canal protegido del peer es de 21 bytes como era de esperar.

```python
# Obtengo el header que son los primeros 22 bytes del mensaje EAP-PSK 4 (todos los campos anteriores al Pchannel del par)
hdr4 = bytes.fromhex(Code_4 + convertir_a_hexadecimal(ID_4) + Length_4_hex + Type + Flag_4 + RAND_S)
# Imprimo el header del mensaje EAP-PSK-4
print("Header de EAP-PSK-4:", hdr4.hex())
# Compruebo si el tamaño de los datos que van a formar parte del Pchannel del peer es de 1 byte o más
left = Length_4 - 4 - 16 - len(hdr4)
data_len = 1
if (decrypted_data[0] & EAP_PSK_E_FLAG) and left > 1:
    data_len += 1
plen = 4 + 16 + data_len
# Imprimo el tamaño de los datos del Pchannel del peer
print("Tamaño de los datos del Pchannel del peer:", data_len)
# Imprimo el tamaño total del Pchannel del peer
print("Tamaño del Pchannel del peer:", plen)
```

Figura 6.24.- Método render_post de la clase ThirdResource del script peer.py parte 7

Sólo me queda calcular el canal protegido del peer para poder construir al completo el mensaje EAP-PSK-4. Para ello, lo primero que necesito hacer es obtener los datos descifrados que compondrán la carga útil en texto plano que se va a cifrar a través del canal protegido del peer, en función de si los datos descifrados obtenidos anteriormente (decrypted_data) tienen el indicador de extensión (E) activado y de si hubo algún fallo durante el proceso o no. De modo que si no hubo fallo y el indicador de extensión está desactivado, construyo los datos descifrados con el indicador de resultado (R) indicando éxito (DONE_SUCCES). Convierto estos datos a hexadecimal y posteriormente a bytes y los asigno a la variable data4.

```
# Obtengo el dato descifrado (texto plano) que va a ir en el Pchannel del peer en funcion de si R es un DONE_SUCCES u otro tipo
# Inicializo la variable failed para rastrear si ha ocurrido un error durante el proceso
failed = 0
# Verifico si el primer byte de los datos desencriptados tiene activada la bandera "E" (extension)
if decrypted_data[0] & EAP_PSK_E_FLAG:
    # Si la bandera "E" esta activada, imprimo un mensaje de advertencia indicando que no es compatible
    print("EAP-PSK: Flag E (Ext) no soportado")
    # Marco el proceso como fallido
    failed = 1
    # Construyo los datos desencriptados indicando un fallo, desplazando la bandera de fallo 6 bits e incluyendo la bandera "E"
    decrypted_data_ = (EAP_PSK_R_FLAG_DONE_FAILURE << 6) | EAP_PSK_E_FLAG
# Si no hay una bandera "E", pero hubo un fallo en algun paso anterior
elif failed:
    # Construyo los datos desencriptados indicando fallo, sin incluir la bandera "E"
    decrypted_data_ = EAP_PSK_R_FLAG_DONE_FAILURE << 6
# Si no hubo fallos y la bandera "E" no esta activada
else:
    # Construyo los datos desencriptados indicando exito, desplazando la bandera de exito 6 bits
    decrypted_data_ = EAP_PSK_R_FLAG_DONE_SUCCESS << 6
# Asigno a la variable data4 el valor de los datos desencriptados en bytes que formaran el Pchannel del par
data4 = bytes.fromhex(hex(decrypted_data_)[2:])
# Imprimo si hubo fallo
print("¿Hubo fallo?:", "Si" if failed == 1 else "No")
# Imprimo los datos descifrados del Pchannel del peer
print("Datos descifrados (texto plano) del Pchannel del peer:", data4.hex())
```

Figura 6.25.- Método render_post de la clase ThirdResource del script peer.py parte 8

Tras esto, tengo que obtener los datos cifrados que compondrán la carga útil cifrada utilizando la función aes_128_eax_encrypt, que recibe como parámetros la TEK, un Nonce (en este caso nonce4, de tamaño 16 bytes y con todos los bits a 0 a excepción del último que está a 1), la longitud de ese Nonce, el Header de 22 bytes del mensaje EAP-PSK-4 (hdr4), la longitud de ese Header, la carga útil en texto plano obtenida en el paso anterior (data4), la longitud de esa carga útil en texto plano y un Tag (tag4) de 16 bytes todos ellos a 0. Lo que

hace esta función es devolver la carga útil cifrada y actualizar el valor del Tag proporcionado (tag4) con el calculado dentro de la propia función.

Ahora ya tengo todo lo necesario para construir el canal protegido del peer, que está compuesto por la concatenación de los últimos 4 bytes del nonce4 (“00000001”), el tag4 y la carga útil cifrada obtenida en el paso anterior (encrypted_data_peer).

Finalmente, una vez que consigo tener todos los campos definidos, construyo el mensaje EAP-PSK-4 concatenando cada uno de esos campos en el orden correcto (EAP Code, Message ID, Length, Type, Flag, RAND_S, RAND_P y canal protegido del peer (Pchannel_peer)) y ese será el payload de la respuesta.

```
# Calculo el Pchannel del peer
# Para ello defino un nonce que sera un array de bytes de tamaño 16 todos ellos a 0 menos el ultimo bit que sera un 1
nonce4 = bytearray(16)
nonce4[-1] = 1
# Tambien defino un tag que sera inicialmente un array de bytes de tamaño 16 todos ellos a 0
tag4 = bytearray(16)
# Paso a la funcion aes_128_eax_encrypt en bytes la TEK, el nonce, el header del EAP-PSK-4, los datos descifrados, el tag
# y las longitudes del nonce, del header y de los datos descifrados para así calcular los datos cifrados
encrypted_data_peer = aes_128_eax_encrypt(bytes.fromhex(TEK), nonce4, len(nonce4), hdr4, len(hdr4), data4, len(data4), tag4)
# Construyo el Pchannel del par EAP concatenando los últimos 4 bytes del nonce, los 16 bytes del tag resultante
# tras ejecutarse la funcion anterior y los datos encriptados
Pchannel_Peer = nonce4[-4:] + tag4 + encrypted_data_peer
# Imprimo los datos encriptados calculados del Pchannel del peer
print("Datos encriptados del Pchannel del peer:", encrypted_data_peer.hex())
# Imprimo el Tag calculado del Pchannel del peer
print("Tag del Pchannel del peer:", tag4.hex())
# Imprimo en hexadecimal el Pchannel del peer encriptado calculado
print("Pchannel del peer encriptado calculado:", Pchannel_Peer.hex())
# Concateno todos los campos (en hexadecimal) para formar el mensaje completo EAP-PSK 4
EAP_PSK_MSG_4 = Code_4 + convertir_a_hexadecimal(ID_4) + Length_4_hex + Type + Flag_4 + RAND_S + Pchannel_Peer.hex()
# Imprimo el mensaje EAP-PSK-4
print("Mensaje EAP-PSK-4:", EAP_PSK_MSG_4)
# Asigno el payload de la respuesta al valor del mensaje EAP-PSK-4 en hexadecimal
payload = EAP_PSK_MSG_4
```

Figura 6.26.- Método render_post de la clase ThirdResource del script peer.py parte 9

Después, establezco el content format a OCTETSTREAM (debería de ser COAP-EAP al igual que en las respuestas de las clases anteriores). Tras esto, construyo el mensaje de respuesta con código CREATED y con el payload y el content-format que acabo de establecer, creo la nueva URI (‘auth/eap/4’) que será la que utilice el EAP Authenticator para enviar la petición POST con el mensaje Success del método EAP (EAP Success) y la

añado a la opción Location_Path del mensaje de respuesta. Seguidamente, al atributo de instancia self.root le agrego un recurso (FourthResource) en la ruta '/auth/eap/4', para que cuando el EAP Authenticator realice una petición POST a esa URI (será el mensaje EAP Success), el EAP Peer usará la lógica implementada en FourthResource para procesar esa petición y responderla. Finalmente, envío el mensaje de respuesta con el código CREATED que contiene el cuarto mensaje del método EAP (EAP-PSK-4) al EAP Authenticator y elimino el recurso actual (auth/eap/3).

```
# Establezco el formato del contenido a OCTETSTREAM (deberia de ser COAP-EAP)
content = ContentFormat.OCTETSTREAM
# Construyo el mensaje de respuesta con codigo CREATED y con el payload y el content_format establecidos anteriormente
resp = aiocoap.Message(code=Code.CREATED, payload=bytes.fromhex(payload), content_format=content)
# Incremento el valor de la variable counter en 1
counter = self.counter + 1
# Creo una nueva URI concatenando 'auth/eap/' con el nuevo valor de counter (sera la que se utilizara para enviar
# el POST con el mensaje EAP-Success)
URI = 'auth/eap/' + str(counter)
# Divido la nueva URI en partes separadas por el caracter '/'
URI_partes = URI.split('/')
# Añado la URI creada a la opcion location_path del mensaje de respuesta
resp.opt.location_path = [URI_partes[0], URI_partes[1], URI_partes[2]]
# Indico que estoy enviando la respuesta al autenticador EAP
print("Enviando respuesta con el cuarto mensaje del metodo EAP (EAP-PSK-4) al controlador...")
# Asocio la URI creada anteriormente con un recurso (FourthResource) que manejara las peticiones que lleguen a esa ruta
self.root.add_resource([URI_partes[0], URI_partes[1], URI_partes[2]], FourthResource(self.root, URI, counter))
# Indico que se ha agregado un nuevo recurso asociado a la uri generada
print("Nuevo recurso agregado con URI:", URI)
# Divido la URI que se necesita eliminar en partes separadas por el caracter '/'
uri_to_remove_partes = self.uri_to_remove.split('/')
# Indico que se va a eliminar el recurso ThirdResource una vez que envio la respuesta y añado el nuevo recurso
print(f"Eliminando recurso con URI: {uri_to_remove_partes[0] + '/' + uri_to_remove_partes[1] + '/' + uri_to_remove_partes[2]}")
# Elimino el recurso con la uri asociada a este tercer recurso (ThirdResource)
self.root.remove_resource([uri_to_remove_partes[0], uri_to_remove_partes[1], uri_to_remove_partes[2]])
print("")
# Retorno el mensaje de respuesta
return resp
```

Figura 6.27.- Método render_post de la clase ThirdResource del script peer.py parte 10

La clase FourthResource maneja la solicitud POST que contiene el mensaje Success del método EAP (EAP Success) protegido con OSCORE, transportado en la carga útil del mensaje CoAP enviado por el EAP Authenticator y genera un mensaje de respuesta con código CHANGED, también protegido con OSCORE, que será enviada por el EAP Peer. Consta de un método constructor que recibe tres parámetros: root (self.root de la clase ThirdResource), uri_to_remove ('auth/eap/4') y counter (4), y los asigna a los atributos de instancia self.root, self.uri_to_remove y self.counter respectivamente. De esta forma se define el estado inicial del constructor de esta clase.

```python
class FourthResource(resource.Resource):
    # Defino el metodo constructor __init__, que recibe tres parametros: root, uri_to_remove y counter
    def __init__(self, root, uri_to_remove, counter):
        # Llamo al constructor de la clase base para asegurarme de que toda inicializacion en la clase padre 'Resource'
        # se realice correctamente
        super().__init__()
        # Asigno el recurso raiz 'root' recibido como parametro al atributo de instancia 'self.root'.
        # 'root' representa el recurso principal que actua como contenedor para acceder a todos los recursos en el
        # servidor CoAP.
        self.root = root
        # Asigno la URI que se desea eliminar ('uri_to_remove') recibida como parametro al atributo de instancia
        # 'self.uri_to_remove'. Esta URI representa el recurso especifico que se espera eliminar en el contexto de esta
        # instancia.
        self.uri_to_remove = uri_to_remove
        # Asigno el valor del counter recibido como parametro (counter perteneciente a la URI del EAP-Success) al
        # atributo de instancia 'self.counter'
        self.counter = counter
```

Figura 6.28.- Constructor de la clase FourthResource del script peer.py

La clase FourthResource también consta de un método render_post que se encarga de manejar todas las solicitudes CoAP con código POST enviadas por el EAP Authenticator a este cuarto recurso (auth/eap/4). Lo primero que hago en este método, dentro de un bloque try (coloco dentro de try el código que se va a ejecutar normalmente, pero que podría causar un error. Si no ocurre ningún error, Python ignora el bloque except y sigue ejecutando el programa. Pero si ocurre un error en el bloque try, Python salta directamente al bloque except), es mostrar por consola el mensaje POST protegido con OSCORE con el EAP SUCCESS enviado por el EAP Authenticator y definir las variables que voy a necesitar para, posteriormente, construir el contexto de seguridad OSCORE del EAP Peer.

Para empezar, accedo a la MSK obtenida en la clase ThirdResource a través de root y la empleo para calcular la Master Secret de tamaño 16 bytes (**Master Secret = KDF (MSK, CS | ”COAP-EAP OSCORE Master Secret”, length)**) y la Master Salt de tamaño 8 bytes (**Master Salt = KDF (MSK, CS | ”COAP-EAP OSCORE Master Salt”, length)**) empleando la función derive_OSCORE_keys desarrollada en el script cálculos_OSCORE.py. Esta función recibe como parámetros la MSK y la cadena CS, que consiste en la concatenación del contenido de la negociación de cipher suite, es decir, la lista de cipher suites enviada por el EAP Authenticator en el EAP Request ID concatenada a la cipher suite seleccionada por el EAP Peer en el EAP Response ID. A continuación, asigno la Master Secret y la Master Salt obtenidas a las variables SECRET y SALT respectivamente, defino un Context ID de OSCORE para el EAP Peer (tiene que ser el mismo que el del EAP Authenticator), defino el mensaje protegido que quiero desproteger

(PROTECTED_MESSAGE), que va a ser el mensaje EAP SUCCESS protegido recibido, y establezco como algoritmos AEAD y HASH a utilizar los elegidos por el EAP Peer en la negociación de cipher suite (son los algoritmos predeterminados).

```
async def render_post(self, request):
    try:
        # Muestro el mensaje POST protegido con OSCORE recibido con el EAP-SUCCESS
        print("Mensaje POST con el EAP-SUCCESS protegido con OSCORE recibido:", request.payload.hex())
        print("")
        # Accedo a la MSK guardada en la clase ThirdResource a traves de root
        MSK = bytes.fromhex(self.root.msk)
        # Accedo al payload del EAP Request ID guardado en la clase FirstResource a traves de root
        first_resource_payload = self.root.first_resource_payload
        # Accedo a la estructura CBOR guardada en la clase FirstResource a traves de root
        cbor_payload = self.root.cbor_payload
        # Concateno el contenido de la negociacion del conjunto de cifrado, es decir, la lista de conjuntos de cifrado
        # enviados por el autenticador EAP en el EAP Request ID concatenada a la opcion seleccionada por el par EAP en el
        # EAP Response ID y lo convierto a hexadecimal
        CS = first_resource_payload[-9:-3].hex() + cbor_payload[2:4].hex()
        # Derivo las claves OSCORE (Master Secret y Master Salt) usando la MSK y el CS
        MASTER_SECRET, MASTER_SALT = derive_oscore_keys(MSK, CS)
        # Asigno a la variable SECRET el valor de la Master Secret y a la variable SALT el valor de la Master Salt obtenidos
        # en la derivacion de las claves OSCORE convertidos a bytes
        SECRET = bytes.fromhex(MASTER_SECRET)
        SALT = bytes.fromhex(MASTER_SALT)
        # Defino el ID del contexto OSCORE del par EAP (tiene que ser el mismo que el del autenticador EAP) y lo convierto a bytes
        ID_CTX = bytes.fromhex("37cbf3210017a2d3")
        # Defino el mensaje protegido que quiero desproteger (el mensaje EAP-SUCCESS protegido), convirtiendolo de hexadecimal a bytes
        PROTECTED_MESSAGE = bytes.fromhex(request.payload.hex())
        # Establezco el algoritmo AEAD y el algoritmo hash seleccionados por el par EAP
        default_algorithm = aiocoap.oscore.algorithms[self.root.algoritmo_AEAD]  # Algoritmo AES_CCM
        default_hashfun = aiocoap.oscore.hashfunctions[self.root.algoritmo_hash]  # Funcion hash SHA-256
```

Figura 6.29.- Método render_post de la clase FourthResource del script peer.py parte 1

Un aspecto importante a destacar es que la implementación del contexto de seguridad OSCORE en ambos extremos de la comunicación, así como la protección, desprotección y envío de los mensajes protegidos, supuso un gran reto técnico en este proyecto. Aunque la biblioteca aiocoap es una de las pocas herramientas en Python que ofrece soporte para OSCORE, su implementación es limitada y carece de ejemplos claros sobre cómo gestionar estos aspectos de forma práctica. La documentación y los ejemplos disponibles no eran suficientes para cubrir las necesidades específicas de esta implementación. Este desafío me obligó a profundizar en el funcionamiento interno de aiocoap, además de investigar detalladamente otros recursos, para adaptar las funciones de protección y desprotección de mensajes a los requisitos del proyecto.

El desarrollo fue un proceso muy laborioso ya que implicó realizar múltiples pruebas y ajustes en el código para asegurar que los mensajes protegidos se generaban y se desprotegían adecuadamente en el cliente y en el servidor, y que cada mensaje cumplía con los estándares de seguridad definidos por OSCORE, lo que incluía manejar correctamente los parámetros del contexto de seguridad como el Sender ID y el Recipient ID, entre otros, para cada extremo. El proceso seguido para crear el contexto de seguridad OSCORE del EAP Peer y desproteger el mensaje EAP SUCCESS se va a explicar a continuación.

En primer lugar defino una clase de contexto de seguridad personalizada llamada OSCORESecurityContext que hereda de la clase CanProtect (permite proteger los mensajes CoAP), de la clase CanUnprotect (maneja la recepción y el procesamiento de mensajes que han sido protegidos, asegurando que se puedan desproteger y validar correctamente) y de la clase SecurityContextUtils (ofrece utilidades relacionadas con el contexto de seguridad como la derivación de la Sender Key, de la Recipient Key y del Common IV), todas ellas desarrolladas en el script OSCORE.py de la biblioteca aiocoap. Esta clase consta de un método constructor que llama al constructor de las clases base para inicializar cualquier configuración o lógica heredada, crea una Replay Window de tamaño 32, lo que significa que puede mantener un registro de hasta 32 mensajes recientes para detectar duplicados, ayudando a prevenir ataques de repetición al rechazar mensajes duplicados que ya se hayan procesado, y no activa la opción de recuperación de eco (echo_recovery) inicialmente. También consta de un método post_seqnoincrease al que se llama al intentar incrementar el número de secuencia, pero no hace nada (no lo voy a utilizar), sólo lo defino puesto que es necesario para que la clase funcione correctamente ya que post_seqnoincrease está definido en CanProtect como un método abstracto, lo que implica que cualquier clase que herede de CanProtect debe implementar este método.

```
class OscoreSecurityContext(CanProtect, CanUnprotect, SecurityContextUtils):
    def __init__(self):
        # Inicializo la clase base
        super().__init__()
        # Establezco una ventana de repeticion con un tamaño de 32
        self.recipient_replay_window = ReplayWindow( size: 32, lambda: None)
        # Inicializo la recuperacion de eco como None
        self.echo_recovery = None

    # Metodo que se llama al intentar incrementar el numero de secuencia, pero no hace nada (es necesario
    # definirlo para que la clase funcione ya que post_seqnoincrease esta definido en CanProtect como un metodo
    # abstracto, lo que implica que cualquier clase que herede de CanProtect debe implementar este metodo)
    def post_seqnoincrease(self):
        pass
```

Figura 6.30.- Método render_post de la clase FourthResource del script peer.py parte 2

Tras esto, creo el contexto de seguridad OSCORE del EAP Peer (secctx) instanciando la clase OSCORESecurityContext y establezco el algoritmo AEAD, la función HASH, el Context ID, el Sender ID (el EAP Peer utiliza el Recipient ID del EAP Authenticator (RID-C) como Sender ID para su Sender Context en OSCORE), el Recipient ID (el EAP Peer usa el Recipient ID del EAP Peer (RID-I) como Recipient ID para su Recipient Context en OSCORE) y el Sender Sequence Number (por ejemplo 20, tiene que ser el mismo que el que se utilice en el EAP Authenticator) que se van a utilizar en este contexto OSCORE del EAP Peer. Además, inicializo la Replay Window como vacía y consigo la Sender Key (sender_key = KDF (master_salt, master_secret, sender_id, "Key")), la Recipient Key (recipient_key = KDF (master_salt, master_secret, recipient_id, "Key")) y el Common IV (common_iv = KDF (master_salt, master_secret, b"", "IV")) llamando a la función derive_keys.

```
# Creo el contexto de seguridad del par EAP instanciando la clase OscoreSecurityContext
secctx = OscoreSecurityContext()
# Establezco el algoritmo AEAD que voy a utilizar en el contexto de seguridad
secctx.alg_aead = default_algorithm
# Establezco la funcion hash que voy a utilizar en el contexto de seguridad
secctx.hashfun = default_hashfun
# El par EAP utiliza el ID de Destinatario del autenticador EAP (RID-C) como ID de Remitente para su contexto de remitente en OSCORE
secctx.sender_id = self.root.rid_c
# El par EAP usa el ID de Destinatario del par EAP (RID-I) como ID de Destinatario para su contexto de destinatario
secctx.recipient_id = self.root.rid_i
# Establezco el ID de contexto
secctx.id_context = ID_CTX
# Llamo a la funcion derive_keys para obtener la clave de remitente, la clave de destinatario y el IV comun a partir
# del algoritmo AEAD, la funcion hash y el ID de contexto ya configurados de antemano, y a partir de la Master Secret
# y de la Master Salt pasadas como parametros
secctx.derive_keys(SALT, SECRET)
# Fijo el numero de secuencia del remitente en 20 (tiene que ser el mismo que el del contexto Oscore del autenticador EAP)
secctx.sender_sequence_number = 20
# Inicializo la ventana de repeticion como vacia
secctx.recipient_replay_window.initialize_empty()
# Imprimo la cadena CS resultante
print("CS:", CS)
```

Figura 6.31.- Método render_post de la clase FourthResource del script peer.py parte 3

Recordemos que para cada extremo, el contexto de seguridad se compone de un "Common Context", un "Sender Context" y un "Recipient Context".

El Common Context del contexto OSCORE del EAP Peer estará formado por el algoritmo AEAD, el algoritmo HKDF (función HASH), la Master Secret, la Master Salt, el

Context ID y el Common IV; el Sender Context del contexto OSCORE del EAP Peer estará formado por el Sender ID, la Sender Key y el Sender Sequence Number; y el Recipient Context del contexto OSCORE del EAP Peer estará formado por el Recipient ID, la Recipient Key y la Replay Window.

```
# Imprimo los parametros del Contexto Comun del contexto de seguridad OSCORE del autenticador EAP
print("Parámetros del Contexto Común del contexto de seguridad OSCORE del autenticador EAP:")
print("Algoritmo AEAD:", self.root.algoritmo_AEAD)
print("Algoritmo HKDF:", self.root.algoritmo_hash)
print("Secreto Maestro (Master Secret):", MASTER_SECRET)
print("Sal Maestra (Master Salt):", MASTER_SALT)
print("ID del Contexto (Context ID):", ID_CTX.hex())
print("IV Común (Common IV):", secctx.common_iv.hex())
# Imprimo los parametros del Contexto de Remitente del contexto de seguridad OSCORE del autenticador EAP
print("Parámetros del Contexto de Remitente del contexto de seguridad OSCORE del autenticador EAP:")
print("ID del Remitente (Sender ID):", secctx.sender_id.hex())
print("Clave del Remitente (Sender Key):", secctx.sender_key.hex())
print("Número de secuencia del Remitente (Sender sequence number):", secctx.sender_sequence_number)
# Imprimo los parametros del Contexto de Destinatario del contexto de seguridad OSCORE del autenticador EAP
print("Parámetros del Contexto de Destinatario del contexto de seguridad OSCORE del autenticador EAP:")
print("ID del Destinatario (Recipient ID):", secctx.recipient_id.hex())
print("Clave del Destinatario (Recipient Key):", secctx.recipient_key.hex())
print("Ventana de repetición (Replay Window): Ventana deslizante anti-repetición de tamaño",
      secctx.recipient_replay_window.get_size(), "mensajes")
print("")
```

Figura 6.32.- Método render_post de la clase FourthResource del script peer.py parte 4

Una vez hecho todo lo anterior, decodifico el mensaje EAP SUCCESS protegido por OSCORE a un objeto de mensaje CoAP de la biblioteca aiocoap al que llamo outer_message, lo desprotejo empleando el contexto de seguridad OSCORE del EAP Peer y lo llamo unprotected_message, asigno el tipo de mensaje (Type), el Message ID, el Token y la opción URI_HOST del mensaje protegido (outer_message) al mensaje desprotegido (unprotected_message) y finalmente codifico el mensaje desprotegido (unprotected_message) a bytes para obtener el mensaje EAP SUCCESS desprotegido, al que llamo encoded_unprotected_message.

```
# Decodifico el mensaje EAP-SUCCESS protegido desde el formato hexadecimal a un objeto de mensaje CoAP de aiocoap
outer_message = aiocoap.Message.decode(PROTECTED_MESSAGE)
# Desprotejo el mensaje protegido usando el contexto de seguridad OSCORE del par EAP
unprotected_message, _ = secctx.unprotect(outer_message)
# Asigno el tipo de mensaje del mensaje protegido al mensaje desprotegido
unprotected_message.mtype = outer_message.mtype
# Asigno el ID de mensaje del mensaje protegido al mensaje desprotegido
unprotected_message.mid = outer_message.mid
# Asigno el token del mensaje protegido al mensaje desprotegido
unprotected_message.token = outer_message.token
# Asigno la opcion URI_HOST del mensaje protegido al mensaje desprotegido
unprotected_message.opt.uri_host = outer_message.opt.uri_host
# Codifico el mensaje desprotegido a bytes
encoded_unprotected_message = unprotected_message.encode()
# Imprimo el mensaje EAP-SUCCESS desprotegido en formato hexadecimal
print("Mensaje EAP-SUCCESS desprotegido:", encoded_unprotected_message.hex())
```

Figura 6.33.- Método render_post de la clase FourthResource del script peer.py parte 5

Seguidamente, con el mensaje EAP SUCCESS ya desprotegido, muestro por consola la información más importante de este mensaje, es decir, el código y el payload. Además, deserializo la estructura CBOR que forma parte del payload del mensaje Success (son los últimos cinco bytes de la carga útil) para obtener el tiempo de vida de la sesión y guardarlo en la variable sesión_lifetime. Si todo es correcto, al imprimir este tiempo de vida se podrá ver que es de 8 horas (es el valor predeterminado) puesto que es el que fijó el EAP Authenticator.

Después, establezco el content format a OCTETSTREAM (debería de ser COAP-EAP al igual que en las respuestas de las clases anteriores). Tras esto, construyo el mensaje de respuesta (resp) con código CHANGED, con un payload que va a ser, por ejemplo, la cadena “FIN” codificada en binario y el content format que acabo de establecer. A esta respuesta le tengo que asignar el Type (en este caso ACK) y el Message ID (será el mismo Message ID que el de la solicitud con el mensaje EAP SUCCESS).

```
print("MENSAJE POST EAP-SUCCESS RECIBIDO (YA DESPROTEGIDO) CON:")
# Imprimo el codigo
print("Código:", request.code)
# Imprimo el payload en binario (son los ultimos 9 bytes del payload ya que el resto son los demas campos del
# mensaje EAP-SUCCESS desprotegidos)
print("Payload:", bytes.fromhex(encoded_unprotected_message.hex())[-9:])
# Imprimo el payload en hexadecimal
print("Payload hexadecimal:", bytes.fromhex(encoded_unprotected_message.hex())[-9:].hex())
# Deserializo la estructura CBOR del payload del EAP-Success para extraer el tiempo de vida de la sesion
deserialized_info = cbor2.loads(bytes.fromhex(encoded_unprotected_message.hex())[-5:])
# Extraigo el tiempo de vida de la sesion y lo asigno a la variable sesion_lifetime
sesion_lifetime = deserialized_info[1]
# Imprimo el tiempo de vida de la sesion extraido
print("Tiempo de vida de la sesión:", sesion_lifetime, "segundos, es decir, ", sesion_lifetime // 3600, "horas")
# Establezco el formato del contenido a OCTETSTREAM (deberia de ser COAP-EAP)
content = ContentFormat.OCTETSTREAM
# El payload de la respuesta sera la palabra "FIN" codificada en binario
payload = "FIN".encode()
# Construyo el mensaje de respuesta con codigo CHANGED y con el payload y el content_format establecidos anteriormente
resp = aiocoap.Message(code=Code.CHANGED, payload=payload, content_format=content)
# Asigno el tipo de mensaje para la respuesta, en este caso lo configuro como ACK
resp.mtype = aiocoap.ACK
# Asigno el ID de mensaje de la solicitud, es decir, del mensaje EAP-SUCCESS desprotegido para asegurar la unicidad
resp.mid = unprotected_message.mid
# Codifico la respuesta a formato hexadecimal
resp_HEX = resp.encode().hex()
```

Figura 6.34.- Método render_post de la clase FourthResource del script peer.py parte 6

Lo que hay que hacer ahora es proteger la respuesta CHANGED con OSCORE. Para ello, decodifico el mensaje de respuesta desde el formato hexadecimal a un objeto de mensaje CoAP de la clase aiocoap al que llamo unprotected, obtengo el Sender ID de la petición (POST con el mensaje Success) que coincidirá con el Recipient ID del contexto de seguridad del EAP Peer, defino el valor del PIV (Partial IV) de la petición, que es un valor que actúa como un identificador de mensaje único en el tiempo usado para evitar la repetición de mensajes y para garantizar la integridad de los mensaje y construyo el Nonce de la petición a partir del PIV y del Sender ID de la petición.

Una vez hecho todo lo anterior, protejo la respuesta empleando el contexto de seguridad OSCORE del EAP Peer y toda la información obtenida anteriormente relacionada con la petición (RequestIdentifiers), entre la que se encuentra el Sender ID, el PIV, el Nonce, si se trata de un mensaje de tipo Confirmable y el método (en este caso POST). A este mensaje de respuesta protegido lo llamo protected_message, le asigno el Type, el Message ID y el Token del mensaje desprotegido (unprotected) al mensaje protegido

(protected_message) y finalmente codifico el mensaje protegido (protected_message) a bytes para obtener el mensaje de respuesta CHANGED protegido, al que llamo encoded_protected_message.

```
unprotected = aiocoap.Message.decode(bytes.fromhex(resp_HEX))
# Obtengo el ID del Remitente de la solicitud, es decir, del mensaje EAP-SUCCESS protegido recibido por el par,
# que coincidira con el ID de Destinatario del contexto de seguridad del par EAP
request_sender_id = secctx.recipient_id
# Defino el valor del PIV de la solicitud
request_piv_short = b"\x14"
# Construyo el nonce de la solicitud usando el PIV y el ID de Remitente de la solicitud
request_nonce = secctx._construct_nonce(request_piv_short, request_sender_id)
# Protejo el mensaje de respuesta usando el contexto de seguridad y la informacion de la solicitud
protected_message, _ = secctx.protect(
    unprotected,  # Mensaje desprotegido que quiero proteger
    aiocoap.oscore.RequestIdentifiers(
        request_sender_id,  # ID del remitente
        request_piv_short,  # Valor corto de PIV
        request_nonce,  # Nonce generado
        can_reuse_nonce: True,  # Indica si es un mensaje de confirmacion
        aiocoap.POST  # Metodo de la solicitud (POST)
    ),
)
# Asigno el ID de mensaje del mensaje desprotegido al mensaje protegido
protected_message.mid = unprotected.mid
# Asigno el token del mensaje desprotegido al mensaje protegido
protected_message.token = unprotected.token
# Asigno el tipo de mensaje del mensaje desprotegido al mensaje protegido
protected_message.mtype = unprotected.mtype
# Codifico el mensaje protegido a bytes
encoded_protected_message = protected_message.encode()
# Imprimo el mensaje CHANGED protegido en formato hexadecimal
print("Mensaje CHANGED protegido por OSCORE:", encoded_protected_message.hex())
```

Figura 6.35.- Método render_post de la clase FourthResource del script peer.py parte 7

Por último, construyo de nuevo el mensaje de respuesta con código CHANGED (resp_prot), con el content format establecido anteriormente (OCTETSTREAM aunque debería ser COAP-EAP) e incluyendo en el payload el mensaje de respuesta completo CHANGED protegido con OSCORE (encoded_protected_message). Lo envío y una vez que el EAP Authenticator reciba este mensaje sabrá que todo el proceso de autenticación se ha completado con éxito y, por tanto, el dispositivo IoT estará correctamente autenticado. Además, en caso de que ocurra un error en la autenticación debido a que no se haya podido desproteger el mensaje EAP SUCCESS como consecuencia de que algún campo del contexto de seguridad OSCORE sea incorrecto o de que la MSK obtenida en el EAP Peer sea diferente a la obtenida en el EAP Authenticator, se indicará que el dispositivo IoT no está autenticado y se construirá un mensaje de respuesta con código UNAUTHORIZED, con un payload que va a ser, por ejemplo, la cadena “Fallo en la autenticación” codificada en

binario y con un formato de contenido que va a ser OCTETSTRING (debería ser CoAP-EAP).

```
    # Construyo un nuevo mensaje de respuesta CoAP con codigo CHANGED enviando en el payload el mensaje CHANGED
    # protegido con OSCORE y con el content_format establecido anteriormente
    resp_prot = aiocoap.Message(code=Code.CHANGED, payload=bytes.fromhex(encoded_protected_message.hex()), content_format=content)
    # Indico que estoy enviando la respuesta al autenticador EAP y que, tras esto, el dispositivo IOT estara autenticado correctamente
    print("Enviando respuesta CHANGED al controlador...")
    print("Dispositivo IOT autenticado con éxito")
    print("")
    # Retorno el mensaje de respuesta
    return resp_prot
except Exception as e:
    # Si ocurre un error en la autenticacion, capturo la excepción y respondo con 4.01 Unauthorized
    # Esto asegura que si hay un fallo, el servidor responde de manera controlada.
    print(f"Error al desproteger o procesar el mensaje, el dispositivo IoT no está autorizado: {e}")
    print("")
    # Retorno el mensaje de respuesta
    return Message(code=Code.UNAUTHORIZED, payload=b"Fallo en la autenticacion", content_format=ContentFormat.OCTETSTREAM)
```

Figura 6.36.- Método render_post de la clase FourthResource del script peer.py parte 8

Lo último que quiero comentar de la clase FourthResource es que también tiene un método render_delete que se encarga de manejar, si la hay, la solicitud CoAP con código DELETE enviada por el EAP Authenticator a este cuarto recurso (auth/eap/4). Sólo tiene este método este último recurso ya que es al que el EAP Authenticator (Controlador) debe enviar la petición DELETE si considera necesario eliminar el "estado" CoAP-EAP del EAP Peer antes de que expire, es decir, si por algún motivo quiere que el dispositivo IoT deje de estar autenticado antes de que caduque el tiempo de vida de la sesión. Esta petición DELETE debe de estar protegida por OSCORE para evitar falsificaciones. Al recibir esta petición, el servidor CoAP (EAP Peer) debe enviar una respuesta al EAP Authenticator con el código DELETED, que también estará protegida por OSCORE.

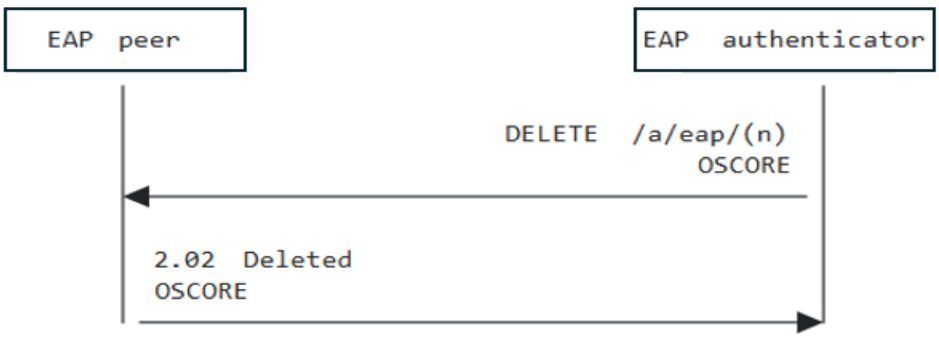


Figura 6.37.- MSC de la eliminación de un cliente del dominio de autenticación

Lo primero que hago en este método es mostrar por consola el mensaje DELETE protegido con OSCORE enviado por el EAP Authenticator y definir las variables que voy a necesitar para posteriormente construir el contexto de seguridad OSCORE del EAP Peer.

Para empezar, accedo a la MSK obtenida en la clase ThirdResource a través de root y la empleo para calcular la Master Secret de tamaño 16 bytes y la Master Salt de tamaño 8 bytes empleando la función derive_OSCORE_keys desarrollada en el script cálculos_OSCORE.py. Esta función recibe como parámetros la MSK y la cadena CS, que consiste en la concatenación del contenido de la negociación de cipher suite, es decir, la lista de cipher suites enviada por el EAP Authenticator en el EAP Request ID concatenada a la cipher suite seleccionada por el EAP Peer en el EAP Response ID. A continuación, asigno la Master Secret y la Master Salt obtenidas a las variables SECRET y SALT respectivamente, defino un Context ID de OSCORE para el EAP Peer (tiene que ser el mismo que el del EAP Authenticator), el mensaje protegido que quiero desproteger (PROTECTED_MESSAGE), que va a ser el mensaje DELETE protegido recibido y establezco como algoritmos AEAD y HASH a utilizar los elegidos por el EAP Peer en la negociación de cipher suite (son los algoritmos predeterminados).

```
async def render_delete(self, request):
    # Muestro el mensaje DELETE protegido con OSCORE recibido
    print("Mensaje DELETE al último recurso protegido con OSCORE recibido:", request.payload.hex())
    # Accedo a la MSK guardada en la clase ThirdResource a traves de root
    MSK = bytes.fromhex(self.root.msk)
    # Accedo al payload del EAP Request ID guardado en la clase FirstResource a traves de root
    first_resource_payload = self.root.first_resource_payload
    # Accedo a la estructura CBOR guardada en la clase FirstResource a traves de root
    cbor_payload = self.root.cbor_payload
    # Concateno el contenido de la negociacion del conjunto de cifrado, es decir, la lista de conjuntos de cifrado
    # enviados por el autenticador EAP en el EAP Request ID concatenada a la opcion seleccionada por el par EAP en el
    # EAP Response ID y lo convierto a hexadecimal
    CS = first_resource_payload[-9:-3].hex() + cbor_payload[2:4].hex()
    # Derivo las claves OSCORE (Master Secret y Master Salt) usando la MSK y el CS
    MASTER_SECRET, MASTER_SALT = derive_oscore_keys(MSK, CS)
    # Asigno a la variable SECRET el valor de la Master Secret y a la variable SALT el valor de la Master Salt obtenidos
    # en la derivacion de las claves OSCORE convertidos a bytes
    SECRET = bytes.fromhex(MASTER_SECRET)
    SALT = bytes.fromhex(MASTER_SALT)
    # Defino el ID del contexto OSCORE del par EAP (tiene que ser el mismo que el del autenticador EAP) y lo convierto a bytes
    ID_CTX = bytes.fromhex("37cbf3210017a2d3")
    # Defino el mensaje protegido que quiero desproteger (el mensaje DELETE protegido), convirtiendolo de hexadecimal a bytes
    PROTECTED_MESSAGE = bytes.fromhex(request.payload.hex())
    # Establezco el algoritmo AEAD y el algoritmo hash seleccionados por el par EAP
    default_algorithm = aiocoap.oscore.algorithms[self.root.algoritmo_AEAD]  # Algoritmo AES_CCM
    default_hashfun = aiocoap.oscore.hashfunctions[self.root.algoritmo_hash]  # Funcion hash SHA-256
```

Figura 6.38.- Método render_delete de la clase FourthResource del script peer.py parte 1

El proceso seguido para crear el contexto de seguridad OSCORE del EAP Peer y desproteger el mensaje DELETE se va a explicar a continuación.

En primer lugar defino una clase de contexto de seguridad personalizada llamada OSCORESecurityContext que hereda de la clase CanProtect, de la clase CanUnprotect y de la clase SecurityContextUtils, todas ellas desarrolladas en el script OSCORE.py de la biblioteca aiocoap. Esta clase consta de un método constructor que llama al constructor de las clases base para inicializar cualquier configuración o lógica heredada, crea una Replay Window (Replay Window) de tamaño 32 y no activa la opción de recuperación de eco (echo_recovery) inicialmente. También consta de un método post_seqnoincrease al que se llama al intentar incrementar el número de secuencia, pero no hace nada (no lo voy a utilizar), sólo lo defino puesto que es necesario para que la clase funcione correctamente ya que post_seqnoincrease está definido en CanProtect como un método abstracto, lo que implica que cualquier clase que herede de CanProtect debe implementar este método.

```
class OscoreSecurityContext(CanProtect, CanUnprotect, SecurityContextUtils):
    def __init__(self):
        # Inicializo la clase base
        super().__init__()
        # Establezco una ventana de repeticion con un tamaño de 32
        self.recipient_replay_window = ReplayWindow( size: 32, lambda: None)
        # Inicializo la recuperacion de eco como None
        self.echo_recovery = None

    # Metodo que se llama al intentar incrementar el numero de secuencia, pero no hace nada (es necesario
    # definirlo para que la clase funcione ya que post_seqnoincrease esta definido en CanProtect como un metodo
    # abstracto, lo que implica que cualquier clase que herede de CanProtect debe implementar este metodo)
    def post_seqnoincrease(self):
        pass
```

Figura 6.39.- Método render_delete de la clase FourthResource del script peer.py parte 2

Tras esto, creo el contexto de seguridad OSCORE del EAP Peer (secctx) instanciando la clase OSCORESecurityContext y establezco el algoritmo AEAD, la función HASH, el Context ID, el Sender ID (el EAP Peer utiliza el Recipient ID del EAP Authenticator (RID-C) como Sender ID para su Sender Context en OSCORE), el Recipient ID (el EAP Peer usa el Recipient ID del EAP Peer (RID-I) como Recipient ID para su Recipient Context en OSCORE) y el Sender Sequence Number (por ejemplo 20, tiene que ser el mismo que el que

utilice en el EAP Authenticator) que se van a utilizar en este contexto OSCORE del EAP Peer.

Además, inicializo la Replay Window como vacía y consigo la Sender Key (**sender_key = KDF (master_salt, master_secret, sender_id, "Key")**), la Recipient Key (**recipient_key = KDF (master_salt, master_secret, recipient_id, "Key")**) y el Common IV (**common_iv = KDF (master_salt, master_secret, b"", "IV")**) llamando a la función derive_keys.

```
# Creo el contexto de seguridad del par EAP instanciando la clase OscoreSecurityContext
secctx = OscoreSecurityContext()
# Establezco el algoritmo AEAD que voy a utilizar en el contexto de seguridad
secctx.alg_aead = default_algorithm
# Establezco la funcion hash que voy a utilizar en el contexto de seguridad
secctx.hashfun = default_hashfun
# El par EAP utiliza el ID de Destinatario del autenticador EAP (RID-C) como ID de Remitente para su contexto de remitente en OSCORE
secctx.sender_id = self.root.rid_c
# El par EAP usa el ID de Destinatario del par EAP (RID-I) como ID de Destinatario para su contexto de destinatario
secctx.recipient_id = self.root.rid_i
# Establezco el ID de contexto
secctx.id_context = ID_CTX
# Llamo a la funcion derive_keys para obtener la clave de remitente, la clave de destinatario y el IV comun a partir
# del algoritmo AEAD, la funcion hash y el ID de contexto ya configurados de antemano, y a partir de la Master Secret
# y de la Master Salt pasadas como parametros
secctx.derive_keys(SALT, SECRET)
# Fijo el numero de secuencia del remitente en 20 (tiene que ser el mismo que el del contexto Oscore del autenticador EAP)
secctx.sender_sequence_number = 20
# Inicializo la ventana de repeticion como vacia
secctx.recipient_replay_window.initialize_empty()
```

Figura 6.40.- Método render_delete de la clase FourthResource del script peer.py parte 3

Una vez hecho todo lo anterior, decodifico el mensaje DELETE protegido por OSCORE a un objeto de mensaje CoAP de la biblioteca aiocoap al que llamo outer_message, lo desprotejo empleando el contexto de seguridad OSCORE del EAP Peer y lo llamo unprotected_message, asigno el Type, el Message ID, el Token y la opción URI_HOST del mensaje protegido (outer_message) al mensaje desprotegido (unprotected_message) y finalmente, codifico el mensaje desprotegido (unprotected_message) a bytes para obtener el mensaje DELETE desprotegido, al que llamo encoded_unprotected_message.

```
# Decodifico el mensaje DELETE protegido desde el formato hexadecimal a un objeto de mensaje CoAP sin proteccion
outer_message = aiocoap.Message.decode(PROTECTED_MESSAGE)
# Desprotejo el mensaje protegido usando el contexto de seguridad OSCORE del par EAP
unprotected_message, _ = secctx.unprotect(outer_message)
# Asigno el tipo de mensaje del mensaje protegido al mensaje desprotegido
unprotected_message.mtype = outer_message.mtype
# Asigno el ID de mensaje del mensaje protegido al mensaje desprotegido
unprotected_message.mid = outer_message.mid
# Asigno el token del mensaje protegido al mensaje desprotegido
unprotected_message.token = outer_message.token
# Asigno el atributo URI_HOST del mensaje protegido al mensaje desprotegido
unprotected_message.opt.uri_host = outer_message.opt.uri_host
# Codifico el mensaje desprotegido a bytes
encoded_unprotected_message = unprotected_message.encode()
# Imprimo el mensaje DELETE desprotegido en formato hexadecimal
print("Mensaje DELETE al último recurso desprotegido:", encoded_unprotected_message.hex())
```

Figura 6.41.- Método render_delete de la clase FourthResource del script peer.py parte 4

Seguidamente, con el mensaje DELETE ya desprotegido, muestro por consola la información más importante de este mensaje, es decir, el código y el payload, y establezco el content format a OCTETSTREAM (debería de ser COAP-EAP al igual que en las respuestas de las clases anteriores). Tras esto, construyo el mensaje de respuesta (resp) con código DELETED, con un payload que va a ser, por ejemplo, la cadena "MENSAJE DELETE RECIBIDO" codificada en binario y el content format que acabo de establecer. A esta respuesta le tengo que asignar el Type (en este caso ACK) y el Message ID (será el mismo Message ID que el de la solicitud DELETE).

```
# Muestro la informacion del mensaje DELETE recibido
print("MENSAJE DELETE AL ÚLTIMO RECURSO RECIBIDO (YA DESPROTEGIDO) CON:")
# Imprimo el codigo
print("Código:", request.code)
# Imprimo el payload en binario (son los ultimos 25 bytes del payload ya que el resto son los demas campos del
# mensaje DELETE desprotegidos)
print("Payload:", bytes.fromhex(encoded_unprotected_message.hex())[-25:])
# Imprimo el payload en hexadecimal
print("Payload hexadecimal:", bytes.fromhex(encoded_unprotected_message.hex())[-25:].hex())
# Establezco el formato del contenido a OCTETSTREAM (deberia de ser COAP-EAP)
content = ContentFormat.OCTETSTREAM
# El payload de la respuesta sera, por ejemplo, la frase "MENSAJE DELETE RECIBIDO" codificada en binario
payload = "MENSAJE DELETE RECIBIDO".encode()
# Construyo el mensaje de respuesta con codigo DELETED y con el payload y el content_format establecidos anteriormente
resp = aiocoap.Message(code=Code.DELETED, payload=payload, content_format=content)
# Asigno el tipo de mensaje para la respuesta, en este caso lo configuro como ACK
resp.mtype = aiocoap.ACK
# Asigno el ID de mensaje de la solicitud, es decir, del mensaje DELETE desprotegido para asegurar la unicidad
resp.mid = unprotected_message.mid
# Codifico la respuesta a formato hexadecimal
resp_HEX = resp.encode().hex()
```

Figura 6.42.- Método render_delete de la clase FourthResource del script peer.py parte 5

Lo que hay que hacer ahora es proteger la respuesta DELETED con OSCORE. Para ello, decodifico el mensaje de respuesta desde el formato hexadecimal a un objeto de mensaje CoAP de la clase aiocoap al que llamo unprotected, obtengo el Sender ID de la petición (petición DELETE) que coincidirá con el Recipient ID del contexto de seguridad del EAP Peer, defino el valor del PIV (Partial IV) de la petición y construyo el Nonce de la petición a partir del PIV y del Sender ID de la petición.

Una vez hecho todo lo anterior, protejo la respuesta empleando el contexto de seguridad OSCORE del EAP Peer y toda la información obtenida anteriormente relacionada con la petición (RequestIdentifiers), entre la que se encuentra el Sender ID, el PIV, el Nonce, si se trata de un mensaje de tipo Confirmable y el método (en este caso DELETE). A este mensaje de respuesta protegido lo llamo protected_message, le asigno el Type, el Message ID y el Token del mensaje desprotegido (unprotected) al mensaje protegido (protected_message) y finalmente, codifico el mensaje protegido (protected_message) a bytes para obtener el mensaje de respuesta DELETED protegido, al que llamo encoded_protected_message.

```
unprotected = aiocoap.Message.decode(bytes.fromhex(resp_HEX))
# Obtengo el ID del Remitente de la solicitud, es decir, del mensaje DELETE protegido recibido por el par,
# que coincidira con el ID de Destinatario del contexto de seguridad del par EAP
request_sender_id = secctx.recipient_id
# Defino un valor corto de PIV (Pseudorandom IV)
request_piv_short = b"\x14"
# Construyo un nonce para proteger el mensaje de respuesta usando el PIV y el ID de Remitente de la solicitud
request_nonce = secctx._construct_nonce(request_piv_short, request_sender_id)
# Protejo el mensaje usando el contexto de seguridad y la informacion de la solicitud
protected_message, _ = secctx.protect(
    unprotected,  # Mensaje desprotegido que quiero proteger
    aiocoap.oscore.RequestIdentifiers(
        request_sender_id,  # ID del remitente
        request_piv_short,  # Valor corto de PIV
        request_nonce,  # Nonce generado
        can_reuse_nonce: True,  # Indica si es un mensaje de confirmacion
        aiocoap.DELETE  # Metodo de la solicitud (DELETE)
    ),
)
# Asigno el ID de mensaje del mensaje desprotegido al mensaje protegido
protected_message.mid = unprotected.mid
# Asigno el token del mensaje desprotegido al mensaje protegido
protected_message.token = unprotected.token
# Asigno el tipo de mensaje del mensaje desprotegido al mensaje protegido
protected_message.mtype = unprotected.mtype
# Codifico el mensaje protegido a bytes
encoded_protected_message = protected_message.encode()
# Imprimo el mensaje DELETED protegido en formato hexadecimal
print("Mensaje DELETED protegido por OSCORE:", encoded_protected_message.hex())
```

Figura 6.43.- Método render_delete de la clase FourthResource del script peer.py parte 6

Finalmente, construyo de nuevo el mensaje de respuesta con código DELETED (resp_prot), con el content format establecido anteriormente (OCTETSTREAM aunque debería ser COAP-EAP) e incluyendo en el payload el mensaje completo DELETED protegido con OSCORE. Una vez que el EAP Authenticator reciba este mensaje sabrá que el dispositivo IoT dejará de estar autenticado. Por último, envío el mensaje de respuesta con el código DELETED y elimino el recurso actual (auth/eap/4).

```
# Construyo un nuevo mensaje de respuesta CoAP con codigo DELETED enviando en el payload el mensaje DELETED
# protegido con OSCORE y con el content_format establecido anteriormente
resp_prot = aiocoap.Message(code=Code.DELETED, payload=bytes.fromhex(encoded_protected_message.hex()), content_format=content)
# Indico que estoy enviando la respuesta al autenticador EAP y que, tras esto, el dispositivo IOT dejara de estar autenticado
print("Enviando respuesta DELETED al controlador...")
print("Dispositivo IOT eliminado del dominio de autenticación con éxito")
# Divido la URI que se necesita eliminar en partes separadas por el caracter '/'
uri_to_remove_partes = self.uri_to_remove.split('/')
# Indico que se va a eliminar el recurso FourthResource una vez que envio la respuesta
print(f"Eliminando recurso con URI: {uri_to_remove_partes[0] + '/' + uri_to_remove_partes[1] + '/' + uri_to_remove_partes[2]}")
# Elimino el recurso con la uri asociada a este cuarto recurso (FourthResource)
self.root.remove_resource([uri_to_remove_partes[0], uri_to_remove_partes[1], uri_to_remove_partes[2]])
print("")
# Retorno el mensaje de respuesta
return resp_prot
```

Figura 6.44.- Método render_delete de la clase FourthResource del script peer.py parte 7

6.3.2.- EAP Authenticator

La clase WellKnownResource del script controller.py maneja el mensaje de activación (mensaje POST a la uri /.well-known/coap-eap) enviado por el EAP Peer y genera la respuesta a ese mensaje. Como ya comenté en la explicación del código del EAP Peer, no debería ser necesario responderlo, pero la implementación de la biblioteca aiocoap lo requiere para que no haya error. Esta clase consta de un método constructor que llama al constructor de la clase base (resource.Resource) para asegurar que cualquier inicialización que se deba realizar en la clase base se ejecute correctamente, e inicializa el atributo de instancia llamado ‘p’, que se utilizará para almacenar el payload del mensaje recibido, como una cadena de bytes vacía.

```python
class WellKnownResource(resource.Resource):
    # Defino el metodo constructor __init__, que inicializa una nueva instancia de la clase
    def __init__(self):
        # Llamo al constructor de la clase base usando super() para asegurar que cualquier inicializacion de
        # la clase padre se ejecute correctamente
        super().__init__()
        # Inicializo el atributo p como una cadena de bytes vacia para almacenar el payload del mensaje recibido
        self.p = b""
```

Figura 6.45.- Constructor de la clase WellKnownResource del script controller.py

La clase WellKnownResource también consta de un método render_post que se encarga de manejar la solicitud CoAP con código POST enviada por el EAP Peer al recurso .well-known/coap-eap. Lo que hago en este método es mostrar por consola la información más importante (código y payload) relativa al mensaje de activación enviado por el EAP Peer, establecer el content format del mensaje de respuesta a OCTETSTREAM (debería de ser COAP-EAP, pero como Wireshark aún no lo reconoce, utilizo este para que se pueda ver la carga útil en hexadecimal como ya expliqué anteriormente), construir el mensaje de respuesta con código CONTINUE (pretendo confirmar que se recibió correctamente el mensaje de activación y que se puede continuar con el resto del flujo de operación del proceso de autenticación) con un payload que va a ser, por ejemplo, la cadena “Mensaje de activación recibido correctamente” codificada en binario y con el content format que acabo de establecer y, finalmente, enviar el mensaje de respuesta CONTINUE al EAP Peer.

```python
# Funcion que maneja los mensajes POST enviados a este recurso
async def render_post(self, request):
    # Muestro la informacion del mensaje POST recibido
    print("MENSAJE POST DE ACTIVACIÓN RECIBIDO CON:")
    # Imprimo el codigo
    print("Código:", request.code)
    # Asigno la variable p al payload del mensaje recibido
    self.p = request.payload
    # Imprimo el payload en binario
    print("Payload:", request.payload)
    # Imprimo el payload en hexadecimal
    print("Payload hexadecimal:", request.payload.hex())
    # Indico que recibi el mensaje POST well-known/coap-eap
    print("Confirmo recepción del mensaje de activación...")
    # Establezco el formato del contenido a OCTETSTREAM (deberia de ser COAP-EAP)
    content = ContentFormat.OCTETSTREAM
    # El payload de la respuesta sera la frase "Mensaje de activacion recibido correctamente" en binario
    payload = b"Mensaje de activacion recibido correctamente"
    # Indico que estoy enviando la respuesta para contentar al par EAP
    print("Enviando respuesta CONTINUE al dispositivo IoT...")
    print("")
    # Construyo el mensaje de respuesta con codigo CONTINUE y con el payload y el content_format establecidos anteriormente
    return aiocoap.Message(code=Code.CONTINUE, payload=payload, content_format=content)
```

Figura 6.46.- Método render_post de la clase WellKnownResource del script controller.py

Lo primero que hago dentro del main del script controller.py es definir como variables globales las uris de los mensajes EAP-PSK-1, EAP-PSK-3 y EAP SUCCESS (uri_eappsk1, uri_eappsk3 y uri_eapsuccess respectivamente). Estas variables las inicialicé a nivel de módulo como cadenas vacías tras los imports y antes de la clase WellKnownResource.

```
from calculos_oscore import derive_oscore_keys
from redes import get_wifi_ip


# Inicializo tres variables globales como cadenas vacías. Al estar definidas fuera de cualquier clase o funcion, estas
# variables pueden ser accedidas y modificadas desde cualquier parte del modulo en el que estan definidas. Esto sera
# util mas adelante, ya que los bucles if del main utilizarán estas variables para la construccion de las uris de los
# mensajes POST
uri_eappsk1 = ""
uri_eappsk3 = ""
uri_eapsuccess = ""


# Clase WellKnownResource que manejara el mensaje POST a la uri well-known/coap-eap y generara la respuesta a ese mensaje.
# Como ya comente, no debería ser necesario responderlo, pero la implementacion de la biblioteca lo requiere para que no haya error
1 usage
class WellKnownResource(resource.Resource):
    # Defino el metodo constructor __init__, que inicializa una nueva instancia de la clase
    def __init__(self):
```

Figura 6.47.- Definición de las variables globales a nivel de módulo en el script controller.py

Al estar definidas fuera de cualquier clase o función, estas variables pueden ser accedidas y modificadas desde cualquier parte del módulo en el que están definidas. En este caso actualizaremos su valor dentro de este método render_post, es decir, dejarán de ser cadenas vacías ya que la URI del mensaje EAP-PSK-1 tomará el valor obtenido tras concatenar los valores de la tupla de la opción Location_Path del del mensaje POST recibido con el EAP Response ID en esta línea de código: uri_eappsk1 = val1 + '/' + val2 + '/' + val3, la uri del mensaje EAP-PSK-3 tomará el valor obtenido tras concatenar los valores de la tupla de la opción Location_Path del mensaje POST recibido con el EAP-PSK-2 en esta línea de código: uri_eappsk3 = val4 + '/' + val5 + '/' + val6 y la uri del mensaje EAP SUCCESS tomará el valor obtenido tras concatenar los valores de la tupla de la opción Location_Path del mensaje POST recibido con el EAP-PSK-4 en esta línea de código: uri_eapsuccess = val7 + '/' + val8 + '/' + val9.

El EAP Authenticator solo actúa como servidor para recibir el mensaje de activación y generar la respuesta correspondiente. Por ello, lo siguiente que hago en el main es crear un contexto de servidor CoAP. Este contexto permanece a la espera de recibir la petición de activación desde el EAP Peer en el puerto 5683 y en la dirección IPv4 del Adaptador de LAN inalámbrica Wi-Fi 2, obtenida mediante la función get_wifi_ip del script redes.py. Además, se le asigna un objeto de tipo Site, llamado root, que funciona como contenedor principal para los recursos del servidor CoAP del EAP Authenticator. A este objeto root le agrego dos recursos. El primer recurso es WKCResource y está asociado a la ruta '.well-known/core', para que cuando el EAP Peer realice una petición a ese recurso, el EAP Authenticator devuelva un encabezado con enlaces a los recursos disponibles. El segundo recurso es WellKnownResource y está asociado a la ruta '.well-known/coap-eap', para que cuando el EAP Peer realice una petición a ese recurso (será el mensaje de activación), el EAP Authenticator use la lógica implementada en la clase WellKnownResource para procesar esa petición y responderla.

```
async def main():
    # Defino las variables globales uri_eappsk1, uri_eappsk3, uri_eapsuccess para almacenar y modificar sus valores
    global uri_eappsk1, uri_eappsk3, uri_eapsuccess
    # Creo un objeto de tipo `Site`, que actua como el contenedor principal para los recursos del servidor CoAP
    root = resource.Site()
    # Creo una instancia de la clase WellKnownResource, que representa un recurso en el servidor CoAP.
    well_known_resource = WellKnownResource()
    # Agrego un recurso al sitio raíz que maneja la URI '.well-known/core', que devuelve un encabezado con enlaces a los recursos disponibles
    root.add_resource( path: ['.well-known', 'core'], resource.WKCResource(root.get_resources_as_linkheader))
    # Agrego un recurso al sitio raíz que maneja la URI '.well-known/coap-eap', que se refiere a la instancia de `well_known_resource`
    root.add_resource( path: ['.well-known', 'coap-eap'], well_known_resource)
    # Creo un contexto de servidor CoAP que está configurando para escuchar por ejemplo en la dirección IPv4 del
    # adaptador de LAN inalambrica WIFI-2 (la direccion se obtiene mediante la funcion get_wifi_ip del script redes.py)
    # y en el puerto 5683.
    # Escuchara las peticiones enviadas por el par EAP (cuando este actua como cliente) al autenticador EAP (cuando este actua como servidor)
    # El par EAP unicamente actuara como servidor para recibir el mensaje a la uri ".well-known/coap-eap" y responder con el mensaje CONTINUE
    wifi_ip = get_wifi_ip()
    server_context = await aiocoap.Context.create_server_context(root, bind=(wifi_ip, 5683))
```

Figura 6.48.- Main del script controller.py parte 1

Una vez que recibo el mensaje de activación, espero de manera asíncrona durante 5 segundos a que la clase WellKnownResource genere y envíe la respuesta CONTINUE al EAP Peer. Tras esto, cierro el contexto de servidor CoAP del EAP Authenticator y espero también de forma asíncrona otros 3 segundos para asegurarme de que el contexto se cierra correctamente, de forma que se liberen los recursos y se finalicen las conexiones.

A partir del momento en que el EAP Authenticator envía la respuesta CONTINUE al EAP Peer, dejará de comportarse como un servidor y comenzará a actuar como un cliente hasta que se complete la autenticación. De modo que lo que hago a continuación es crear un contexto de cliente CoAP, el cual será el encargado de enviar las peticiones oportunas al EAP Peer.

```
# Espero de manera asíncrona durante 5 segundos, permitiendo la generacion y el envio de la respuesta CONTINUE durante ese tiempo
await asyncio.sleep(5)
# Cierro el contexto del servidor CoAP de manera asincrona, liberando recursos y finalizando conexiones
await server_context.shutdown()
# Espero de manera asincrona durante 3 segundos a que el contexto del servidor se cierre correctamente
await asyncio.sleep(3)
# A partir de este momento el controller comenzara a comportarse como un cliente
# Creo un contexto de cliente CoAP que me permitira enviar peticiones CoAP al par EAP
context = await Context.create_client_context()
```

Figura 6.49.- Main del script controller.py parte 2

A continuación, comienzo a obtener todos los parámetros necesarios para construir la petición EAP Request ID. Para ello empiezo definiendo en un diccionario las cinco cipher suites admitidas, donde la clave es un índice y el valor es una lista que contiene el algoritmo de cifrado AEAD y el algoritmo HKDF (función HASH). Esto me sirve para construir la estructura CBOR que forma parte del payload del EAP Request ID, la cual estará formada por dos campos: el primer campo son los índices de las cipher suites admitidas entre las que tendrá que elegir el EAP Peer y el segundo campo es el RID-C (Recipient ID del EAP Authenticator), el cual he decidido definir de forma manual y con tamaño de 1 byte, al igual que hice con el RID-I en la estructura CBOR del payload del EAP Response ID en el EAP Peer y por el mismo motivo que expliqué en esa sección del código.

Una vez que ya tengo serializada la información (coap_eap_info1) en la estructura CBOR (cbor_payload1), establezco el payload de la petición EAP Request ID (payload1) formado por la concatenación de una cadena fija (por ejemplo "010700050183000102", obtenida de un ejemplo de paquete EAP Request ID en una traza de Wireshark) y la estructura CBOR construida (cbor_payload1).

```
conjuntos_cifrado = {
    # Este es el conjunto de cifrado predeterminado, usando AES en modo CCM con SHA-256
    0: ["AES-CCM-16-64-128", "SHA-256"],
    # Este es el segundo conjunto de cifrado, usando A128GCM con SHA-256
    1: ["A128GCM", "SHA-256"],
    # Este es el tercer conjunto de cifrado, usando A256GCM con SHA-384
    2: ["A256GCM", "SHA-384"],
    # Este es el cuarto conjunto de cifrado, usando ChaCha20 con Poly1305 y SHA-256
    3: ["ChaCha20/Poly1305", "SHA-256"],
    # Este es el quinto conjunto de cifrado, usando ChaCha20 con Poly1305 y SHAKE256
    4: ["ChaCha20/Poly1305", "SHAKE256"]
}
# Defino de forma manual y con tamaño de 1 byte el ID de Destinatario del autenticador EAP
RID_C = b'\x01'
# Defino la estructura CoAP-EAP_Info1 como un diccionario que contiene informacion sobre los posibles conjuntos de
# cifrado (lo elegira el peer) en orden decreciente de preferencia y el ID de Destinatario del autenticador EAP
coap_eap_info1 = {
    # Lista de indices de los conjuntos de cifrado habilitados (incluyendo todos los definidos)
    1: [0, 1, 2, 3, 4],
    # RID-C como una cadena binaria
    2: RID_C,
}
# Serializo la estructura CoAP-EAP_Info a formato CBOR para su transporte o almacenamiento
cbor_payload1 = cbor2.dumps(coap_eap_info1)
# El payload del mensaje Request ID sera la cadena hexadecimal fija por defecto "010700050183000102" concatenado con
# la estructura de datos CBOR utilizada para la negociacion de cryptosuite
payload1 = "010700050183000102" + cbor_payload1.hex()
```

Figura 6.50.- Main del script controller.py parte 3

Después, construyo la petición CoAP que contiene el EAP Request ID, a la que llamo RequestID, con código POST, con el content format OCTETSTREAM (aunque debería ser CoAP-EAP), el payload previamente definido (payload1) y con la URI "coap://localhost/" + uri_reqid. Dentro de esta URI, la dirección localhost corresponde a la IP de loopback (normalmente 127.0.0.1 en IPv4), que permite que un equipo se comunique consigo mismo. Se emplea esta dirección para vincular el mensaje con el contexto de servidor del EAP Peer, ya que escucha en dicha dirección. El valor de uri_reqid es el payload del mensaje de activación recibido (recordemos que era "auth/eap/1"). Este valor se obtiene accediendo al atributo p de la instancia well_known_resource de la clase WellKnownResource.

Tras construir la petición EAP Request ID, la envío al EAP Peer y espero a recibir la respuesta con código CREATED, a la que llamo ResponseID. Al recibirla, muestro por consola la información más importante (código y payload) relativa a esa respuesta CREATED enviada por el EAP Peer que contiene el EAP Response ID.

```
# Establezco el formato del contenido a OCTETSTREAM (deberia de ser COAP-EAP)
content_format = ContentFormat.OCTETSTREAM
# Accedo al atributo p (payload del mensaje well-known/coap-eap) de la instancia well_known_resource ya que su contenido
# es la URI a la que tengo que enviar el mensaje EAP Request ID, lo convierto a bytes y luego lo decodifico en una cadena de texto
uri_reqid = bytes.fromhex(well_known_resource.p.hex()).decode('utf-8')
# Imprimo cual va a ser la URI del mensaje Request ID para ver si coincide con la generada en el par EAP
print("URI del mensaje EAP Request ID:", uri_reqid)
# Construyo el mensaje RequestID con codigo POST a la uri indicada y con el payload y el content_format establecidos anteriormente
# La uri indicada contiene la direccion localhost para vincularlo con el contexto del servidor del par EAP ya que escucha en esa direccion
RequestId = aiocoap.Message(code=Code.POST, payload=bytes.fromhex(payload1), uri="coap://localhost/"+uri_reqid, content_format=content_format)
# Indico que estoy enviando la peticion POST con el EAP Request ID al par EAP
print("Enviando una petición POST con el EAP Request ID al dispositivo IoT...")
# Envio una solicitud (Request ID) a través del contexto del cliente CoAP y espero la respuesta (Response ID)
ResponseId = await context.request(RequestId).response
# Muestro la informacion del mensaje CREATED recibido con el Response ID
print("MENSAJE CREATED CON EL EAP RESPONSE ID RECIBIDO CON:")
# Imprimo el codigo
print("Codigo:", ResponseId.code)
# Imprimo el payload en binario
print("Payload:", ResponseId.payload)
# Imprimo el payload en hexadecimal
print("Payload hexadecimal:", ResponseId.payload.hex())
print("")
```

Figura 6.51.- Main del script controller.py parte 4

Seguidamente, deserializo la estructura CBOR del payload del EAP Response ID para extraer la información que necesito para OSCORE, en este caso el algoritmo EAED y la función HASH de la cipher suite elegida por el EAP Peer y el RID-I. Imprimo toda esta información junto con el RID-C por consola y guardo el algoritmo AEAD en la variable algoritmo_aead, la función HASH en la variable algoritmo_hash y el Recipient ID del EAP Authenticator en la variable rid_i. Estas variables las necesitaré más adelante cuando cree el contexto de seguridad OSCORE, ya que el RID-I será el Sender ID, el RID-C será el Recipient ID y la cipher suite elegida por el EAP Peer será el algoritmo AEAD y el algoritmo HKDF utilizados por el contexto de seguridad OSCORE del EAP Authenticator.

```
# Deserializo la estructura CBOR del payload del EAP Response-ID recibido para extraer la informacion que necesito para OSCORE
deserialized_info = cbor2.loads(ResponseId.payload[-7:])
# Extraigo la lista de indices de los conjuntos de cifrado y la asigno a la variable cifrado_indices
cifrado_indices = deserialized_info[1]
# Extraigo el primer indice de la lista de conjuntos de cifrado (en este caso el indice 0)
indice_cifrado = cifrado_indices[0]
# Asigno el nombre del algoritmo de cifrado AEAD correspondiente al indice extraido
algoritmo_AEAD = conjuntos_cifrado[indice_cifrado][0]
# Asigno el nombre del algoritmo hash correspondiente al indice extraido
algoritmo_hash = conjuntos_cifrado[indice_cifrado][1]
# Extraigo el RID_I y lo asigno a la variable rid_i
rid_i = deserialized_info[2]
# Imprimo el nombre del algoritmo de cifrado correspondiente al indice extraido
print("Algoritmo de cifrado AEAD elegido por el par EAP:", algoritmo_AEAD)
# Imprimo el nombre del algoritmo hash correspondiente al índice extraído
print("Algoritmo hash elegido por el par EAP:", algoritmo_hash)
# Imprimo el RID-C extraído (ID de Destinatario del autenticador EAP)
print("RID-C:", RID_C)
# Imprimo el RID-I (ID de Destinatario del par EAP)
print("RID-I:", rid_i)
print("")
```

Figura 6.52.- Main del script controller.py parte 5

Ahora comenzaré a obtener todos los parámetros que necesito para construir la petición con el primer mensaje del protocolo EAP-PSK. Para ello, lo primero que tengo que hacer es obtener el contenido de la opción Location_Path de la respuesta con el mensaje EAP Response ID enviada por el EAP Peer. Este contenido será una tupla, que es un tipo de estructura de datos, similar a una lista, que permite almacenar una secuencia de elementos ordenados de distintos tipos de datos pero que es inmutable, es decir, una vez creada no se puede modificar. La tupla para la respuesta con el mensaje EAP Response ID será (‘auth’, ‘eap’, ‘2’), donde el primer elemento (val1) es la cadena de caracteres ‘auth’, el segundo elemento (val2) es la cadena de caracteres ‘eap’ y el tercer elemento (val3) es la cadena de caracteres ‘2’. Con estos valores puedo construir la uri (uri_eappsk1 = val1 + '/' + val2 + '/' + val3 = auth/eap/2) a la que se enviará la petición POST con el primer mensaje EAP-PSK.

```
# Obtengo el valor de LOCATION_PATH de las opciones del mensaje POST con el EAP Response ID
location_path1 = ResponseId.opt.location_path
# Me aseguro de que location_path1 no es None y que es una tupla ('auth', 'eap', 'counter')
if location_path1 and isinstance(location_path1, tuple):
    # Obtengo el primer elemento de la tupla ('auth') y lo almaceno en val1
    val1 = location_path1[0]
    # Obtengo el segundo elemento de la tupla ('eap') y lo almaceno en val2
    val2 = location_path1[1]
    # Obtengo el tercer elemento de la tupla ('counter') y lo almaceno en val3
    val3 = location_path1[2]
    # Concateno los valores obtenidos con '/' para formar la URI completa
    uri_eappsk1 = val1 + '/' + val2 + '/' + val3
    # Imprimo la URI extraida, que sera la URI del mensaje POST con EAP-PSK-1 que se enviara a continuacion
    print("URI del mensaje EAP-PSK-1:", uri_eappsk1)
    print("")
# Si location_path1 es None o no tiene el formato esperado, imprimo un mensaje de error
else:
    print("No se ha encontrado la opción LOCATION_PATH o el formato no es el esperado")
```

Figura 6.53.- Main del script controller.py parte 6

Tras esto, tengo que hacer las llamadas necesarias al script cliente_radius.py, que implementa un cliente RADIUS para la comunicación con el servidor RADIUS del EAP Server, hasta obtener el primer mensaje EAP-PSK (EAP-PSK-1). El proceso para obtener el mensaje EAP-PSK-1 se va a explicar de forma generalizada a continuación (todo el script cliente_radius.py se va a comentar con detalle en el siguiente subapartado de esta sección, por lo que en esta explicación no voy a entrar en muchos detalles sobre sus funciones, métodos o clases).

En primer lugar, creo una instancia de la clase EAP_Authenticator del script cliente_radius.py llamada eap_auth. Esta clase es la que maneja la autenticación EAP, es decir, el intercambio de mensajes del método EAP-PSK hasta obtener el EAP SUCCESS, ya que se encarga de la generación y del envío de los mensajes del protocolo RADIUS dentro de los cuales van a ir esos mensajes del método EAP-PSK. Tras esto, imprimo por consola las variables del estado inicial de autenticación EAP en el cliente RADIUS de la instancia eap_auth. Estas variables serán: la dirección IP del servidor RADIUS (192.168.217.128), la dirección IP del NAS (127.0.0.1), el puerto de origen (por ejemplo 9906) y de destino (1812), el número de paquetes que se van a enviar en la transacción (1), el tiempo máximo de espera por la respuesta (5 segundos), el identificador de la transacción (0), el Shared Secret (testing123) y una serie de atributos propios de RADIUS entre los que están: el tipo de puerto

del servidor NAS (Wireless), el ID de la estación de llamada (00-00-00-00-00-00), la información de la conexión (CON), el tamaño máximo en bytes de los paquetes que pueden ser enviados (1400) y el mensaje EAP ("0237000a017573657261", que es el payload del EAP Response ID). A partir de estas variables construyo un mensaje RADIUS Access-Request, que envío usando la función sendNextMessageToRADIUS al EAP Server (servidor RADIUS) cuyo atributo EAP-Message será el payload del mensaje EAP Response ID. Debido a la forma en que está implementado el servidor RADIUS, en vez de responder directamente con un Access-Challenge en cuyo atributo EAP-Message estuviese ya el mensaje EAP-PSK-1, responde con un Access-Challenge que insta de nuevo a mandarle la identidad del dispositivo IoT que se quiere autenticar (es una especie de bug). Ahora estoy dentro del estado de autenticación REQ_ID_FIX del cliente RADIUS. Imprimo por consola las variables del estado REQ_ID_FIX de RADIUS de la instancia eap_auth. Estas variables serán: la dirección IP del servidor RADIUS (192.168.217.128), la dirección IP del NAS (127.0.0.1), el puerto de origen (por ejemplo 9906) y de destino (1812), el número de paquetes que se van a enviar en la transacción (1), el tiempo máximo de espera por la respuesta (5 segundos), el identificador de la transacción (1), el Shared Secret (testing123) y una serie de atributos propios de RADIUS entre los que están: el tipo de puerto del servidor NAS (Wireless), el ID de la estación de llamada (00-00-00-00-00-00), la información de la conexión (CON), el tamaño máximo en bytes de los paquetes que pueden ser enviados (1400), el estado (por ejemplo "380389c66a3b3dc75a35f45a6f156344" que es el valor del atributo State del mensaje Access-Challenge anterior) y el mensaje EAP ("0269000a017573657261"). Mediante estas variables construyo, empleando la función genNextRadiusMessageFromState, el siguiente mensaje RADIUS Access-Request que envío usando la función sendNextMessageToRADIUS al EAP Server (servidor RADIUS), cuyo atributo EAP-Message será prácticamente una réplica del payload del mensaje EAP Response ID ya que solo cambia el ID, que debe ser el mismo que el del Access-Challenge que acaba de recibir. Este es el mensaje que se manda para solucionar la especie de bug que presenta la implementación del servidor RADIUS y que hace que el servidor RADIUS responda con un Access-Challenge en cuyo atributo EAP-Message está presente el mensaje EAP-PSK-1, el cual guardo en la variable payload2. Esto puedo hacerlo puesto que al enviar este Access-Request al EAP Server, entro dentro del estado de autenticación EAP_PSK_1 del cliente RADIUS y el código del script cliente_radius.py que implementa ese estado, lo

que hace es extraer el contenido del atributo EAP-Message del mensaje Access-Challenge con el que responde el servidor RADIUS y lo retorna.

```
# Hago las llamadas necesarias al script cliente_radius.py que actua como cliente RADIUS hasta obtener el mensaje EAP-PSK-1
# Creo una instancia de la clase EAP_Authenticator, que gestiona el proceso de autenticacion EAP
eap_auth = EAP_Authenticator()
# Imprimo un mensaje indicando que se va a actualizar el estado
print("Update State")
# Imprimo las variables del estado actual de RADIUS dentro del objeto eap_auth
print(vars(eap_auth._radiusState))
# Envio un mensaje RADIUS cuyo atributo EAP-Message contiene el payload del Response ID al servidor RADIUS a partir del estado actual
eap_auth.sendNextMessageToRADIUS()
# Ahora estaria dentro del estado REQ_ID_FIX
# Imprimo nuevamente las variables del estado de RADIUS despues de enviar el mensaje
print(vars(eap_auth._radiusState))
# Genero el siguiente mensaje RADIUS a partir del estado actual. El atributo EAP-Message de este mensaje es
# practicamente una replica del que acaba de enviar al servidor RADIUS, solo cambia el ID, que debe ser el mismo que
# el del Access-Challenge que recibe como respuesta del servidor RADIUS
# Este es el mensaje que se manda para solucionar la especie de bug que presenta la implementacion del servidor RADIUS
eap_auth.genNextRadiusMessageFromState()
# Envio el mensaje RADIUS generado en el paso anterior al servidor RADIUS
# Ya estoy dentro del estado EAP_PSK_1 y puedo guardar el mensaje EAP-PSK-1 en la variable payload2 ya que es lo que me retorna ese estado
payload2 = eap_auth.sendNextMessageToRADIUS()
# Imprimo el mensaje EAP-PSK-1
print("El payload de la petición CoAP que contiene el mensaje EAP-PSK-1 es este:", payload2)
```

Figura 6.54.- Main del script controller.py parte 7

Después, construyo la petición CoAP que contiene el primer mensaje EAP-PSK, a la que llamo EAP1, con código POST, con el content format OCTETSTREAM (debería de ser CoAP-EAP), con un payload que será el contenido de la variable payload2 (mensaje EAP-PSK-1 recibido del EAP Server) y con la uri "coap://localhost/"+uri_eappsk1". Dentro de esta URI, localhost es la dirección IP de loopback (normalmente 127.0.0.1 en IPv4) que se emplea para vincular este mensaje con el contexto de servidor del EAP Peer, ya que escucha en esa dirección. El valor de uri_eappsk1 es la URI construida a partir de los elementos de la tupla de la opción Location_Path del EAP Response ID (recordemos que uri_eappsk1 = auth/eap/2).

Tras construir la petición con el mensaje EAP-PSK-1, la envío al EAP Peer y espero a recibir la respuesta con código CREATED, a la que llamo REAP1. Al recibirla, muestro por consola la información más importante (código y payload) relativa a esa respuesta CREATED enviada por el EAP Peer que contiene el segundo mensaje EAP-PSK.

```
# Construyo el mensaje EAP-PSK-1 con codigo POST a la uri indicada y con el payload y el content_format establecidos anteriormente
EAP1 = aiocoap.Message(code=Code.POST, payload=bytes.fromhex(payload2), uri="coap://localhost/"+uri_eappsk1, content_format=content_format)
# Indico que estoy enviando la peticion POST con el mensaje EAP-PSK-1 al par EAP
print("Enviando POST con el primer mensaje del protocolo EAP (EAP-PSK-1) al dispositivo IOT...")
# Envio una solicitud (EAP-PSK-1) a traves del contexto del cliente CoAP y espero la respuesta (EAP-PSK-2)
REAP1 = await context.request(EAP1).response
# Muestro la informacion del mensaje CREATED recibido con el EAP-PSK-2
print("MENSAJE CREATED CON EL MENSAJE EAP-PSK-2 RECIBIDO CON:")
# Imprimo el codigo
print("Codigo:", REAP1.code)
# Imprimo el payload en binario
print("Payload:", REAP1.payload)
# Imprimo el payload en hexadecimal
print("Payload hexadecimal:", REAP1.payload.hex())
print("")
```

Figura 6.55.- Main del script controller.py parte 8

Una vez hecho todo lo anterior, comenzaré a obtener todos los parámetros que necesito para construir la petición con el tercer mensaje del protocolo EAP-PSK. Para ello, lo primero que tengo que hacer es obtener el contenido de la opción Location_Path de la respuesta con el segundo mensaje EAP-SPK enviada por el EAP Peer. Este contenido será la tupla (‘auth’, ‘eap’, ‘3’), donde el primer elemento (val4) es la cadena de caracteres ‘auth’, el segundo elemento (val5) es la cadena de caracteres ‘eap’ y el tercer elemento (val6) es la cadena de caracteres ‘3’. Con estos valores puedo construir la uri (uri_eappsk3 = val4 + '/' + val5 + '/' + val6 = auth/eap/3) a la que se enviará la petición POST con el tercer mensaje EAP-PSK.

```
# Obtengo el valor de LOCATION_PATH de las opciones del mensaje POST con el EAP-PSK-2
location_path2 = REAP1.opt.location_path
# Me aseguro de que location_path1 no es None y que es una tupla ('auth', 'eap', 'counter')
if location_path2 and isinstance(location_path2, tuple):
    # Obtengo el primer elemento de la tupla ('auth') y lo almaceno en val4
    val4 = location_path2[0]
    # Obtengo el segundo elemento de la tupla ('eap') y lo almaceno en val5
    val5 = location_path2[1]
    # Obtengo el tercer elemento de la tupla ('counter') y lo almaceno en val6
    val6 = location_path2[2]
    # Concateno los valores obtenidos con '/' para formar la URI completa
    uri_eappsk3 = val4 + '/' + val5 + '/' + val6
    # Imprimo la URI extraida, que sera la URI del mensaje POST con EAP-PSK-3 que se enviara a continuacion
    print("URI del mensaje EAP-PSK-3:", uri_eappsk3)
    print("")
# Si location_path1 es None o no tiene el formato esperado, imprimo un mensaje de error
else:
    print("No se ha encontrado la opción LOCATION_PATH o el formato no es el esperado")
```

Figura 6.56.- Main del script controller.py parte 9

Tras esto, establezco que el mensaje EAP que va a ir en el atributo EAP-Message del siguiente mensaje RADIUS Access-Request que se va a enviar al EAP Server (servidor RADIUS) sea EAP-PSK-2, el cual es el payload de la respuesta CREATED enviada por el EAP Peer (REAP1). También, establezco que la próxima vez que haga una llamada a la función sendNextMessageToRADIUS de la clase EAP_Authenticator del cliente RADIUS se ejecute la parte del código relativa al estado de autenticación EAP_PSK_3.

Seguidamente, hago las llamadas necesarias al script cliente_radius.py, hasta obtener el tercer mensaje EAP-PSK (EAP-PSK-3). El proceso para obtener el mensaje EAP-PSK-3 se va a explicar de forma generalizada a continuación.

En primer lugar, como aún estoy dentro del estado de autenticación EAP_PSK_1, imprimo por consola las variables de este estado RADIUS de la instancia eap_auth. Estas variables serán: la dirección IP del servidor RADIUS (192.168.217.128), la dirección IP del NAS (127.0.0.1), el puerto de origen (por ejemplo 9906) y de destino (1812), el número de paquetes que se van a enviar en la transacción (1), el tiempo máximo de espera por la respuesta (5 segundos), el identificador de la transacción (2), el Shared Secret (testing123) y una serie de atributos propios de RADIUS entre los que están: el tipo de puerto del servidor NAS (Wireless), el ID de la estación de llamada (00-00-00-00-00-00), la información de la conexión (CON), el tamaño máximo en bytes de los paquetes que pueden ser enviados (1400), el estado (por ejemplo “4da89643d3410c1b5b4f49acbbdf3169” que es el valor del atributo State del mensaje Access-Challenge anterior) y el mensaje EAP (“026a003c2f405614e250c7162a99101151ec7e7b5a40039fabba6a21359af97ad9b2477a21 a5628f98af6c1aab35edb1930e790110d0636c69656e74”, que es el mensaje EAP-PSK-2 recibido del EAP Peer). A partir de estas variables construyo, empleando la función genNextRadiusMessageFromState, el siguiente mensaje RADIUS Access-Request que envío usando la función sendNextMessageToRADIUS al EAP Server (servidor RADIUS), cuyo atributo EAP-Message contiene el segundo mensaje EAP-PSK cumpliendo con lo que había fijado previamente. El servidor RADIUS responde a este mensaje con un Access-Challenge en cuyo atributo EAP-Message está el mensaje EAP-PSK-3, el cual guardo en la variable payload3. Esto puedo hacerlo puesto que al enviar este Access-Request al EAP Server, entro dentro del estado de autenticación EAP_PSK_3 y el código del script cliente_radius.py que implementa ese estado, lo que hace es extraer el contenido del atributo

EAP-Message del mensaje Access-Challenge con el que responde el servidor RADIUS y lo retorna.

```
# Fijo que el mensaje EAP que va en el atributo EAP-Message sea EAP-PSK-2 para que la proxima vez que haga una
# llamada al cliente RADIUS, el mensaje RADIUS a enviar al servidor RADIUS se construya con ese contenido
eap_auth.set_eap_message(REAP1.payload.hex())
# Fijo el estado a EAP_PSK_3 para que la proxima vez que haga una llamada al cliente RADIUS se ejecute el codigo de dicho estado
eap_auth.set_state(EAPAuthState.EAP_PSK_3)
# Hago las llamadas necesarias al scrip cliente_radius.py que actua como cliente RADIUS hasta obtener el mensaje EAP-PSK-1
# Imprimo las variables del estado actual de RADIUS dentro del objeto eap_auth
print(vars(eap_auth._radiusState))
# Genero el siguiente mensaje RADIUS a partir del estado actual (EAP_PSK_3). Es decir, ahora estoy generando el mensaje
# RADIUS que contiene el EAP-PSK-2 que acabo de recibir del peer
eap_auth.genNextRadiusMessageFromState()
# Como ya estoy dentro del estado EAP_PSK_3, puedo guardar el mensaje EAP-PSK-3 en la variable payload3 ya que es lo
# que me retorna ese estado
payload3 = eap_auth.sendNextMessageToRADIUS()
# Imprimo el mensaje EAP-PSK-3
print("El payload de la petición CoAP que contiene el mensaje EAP-PSK-3 es este:", payload3)
```

Figura 6.57.- Main del script controller.py parte 10

Después, construyo la petición CoAP que contiene el tercer mensaje EAP-PSK, a la que llamo EAP3, con código POST, con el content format OCTETSTREAM (debería de ser CoAP-EAP), con un payload que será el contenido de la variable payload3 (mensaje EAP-PSK-3 recibido del EAP Server) y con la uri "coap://localhost/"+uri_eappsk3". Dentro de esta URI, localhost es la dirección IP de loopback (normalmente 127.0.0.1 en IPv4) que se emplea para vincular este mensaje con el contexto de servidor del EAP Peer, ya que escucha en esa dirección. El valor de uri_eappsk3 es la URI construida a partir de los elementos de la tupla de la opción Location_Path de la respuesta con el mensaje EAP-PSK-2 (recordemos que uri_eappsk3 = auth/eap/3).

Tras construir la petición con el mensaje EAP-PSK-3, la envío al EAP Peer y espero a recibir la respuesta con código CREATED, a la que llamo REAP2. Al recibirla, muestro por consola la información más importante (código y payload) relativa a esa respuesta CREATED enviada por el EAP Peer que contiene el cuarto mensaje EAP-PSK, y guardo el valor del Message ID de esta respuesta en la variable ID1. Utilizaré este valor más adelante para definir el Message ID de la petición CoAP que contiene el mensaje EAP SUCCESS, ya que como se verá, es necesito definirlo para poder proteger ese mensaje con OSCORE.

```
# Construyo el mensaje EAP-PSK-3 con codigo POST a la uri indicada y con el payload y el content_format establecidos anteriormente
EAP3 = aiocoap.Message(code=Code.POST, payload=bytes.fromhex(payload3), uri="coap://localhost/" + uri_eappsk3, content_format=content_format)
# Indico que estoy enviando la peticion POST con el mensaje EAP-PSK-3 al par EAP
print("Enviando POST con el tercer mensaje del protocolo EAP (EAP-PSK-3) al dispositivo IOT...")
# Envio una solicitud (EAP-PSK-3) a traves del contexto del cliente CoAP y espero la respuesta (EAP-PSK-4)
REAP3 = await context.request(EAP3).response
# Guardo el valor del ID de mensaje del cuarto mensaje del metodo EAP (EAP-PSK-4)
ID1 = REAP3.mid
# Muestro la informacion del mensaje CREATED recibido con el EAP-PSK-2
print("MENSAJE CREATED CON EL MENSAJE EAP-PSK-4 RECIBIDO CON:")
# Imprimo el codigo
print("Codigo:", REAP3.code)
# Imprimo el payload en binario
print("Payload:", REAP3.payload)
# Imprimo el payload en hexadecimal
print("Response payload hexadecimal:", REAP3.payload.hex())
print("")
```

Figura 6.58.- Main del script controller.py parte 11

Lo siguiente que hago es comenzar a obtener todos los parámetros que necesito para construir la petición con el mensaje Success del protocolo EAP-PSK. Para ello, lo primero que tengo que hacer es obtener el contenido de la opción Location_Path de la respuesta con el cuarto mensaje EAP-SPK enviada por el EAP Peer. Este contenido será la tupla (‘auth’, ‘eap’, ‘4’), donde el primer elemento (val7) es la cadena de caracteres ‘auth’, el segundo elemento (val8) es la cadena de caracteres ‘eap’ y el tercer elemento (val9) es la cadena de caracteres ‘4’. Con estos valores puedo construir la uri (uri_eapsuccess = val7 + '/' + val8 + '/' + val9 = auth/eap/4) a la que se enviará la petición POST con el mensaje EAP SUCCESS.

```
# Obtengo el valor de LOCATION_PATH de las opciones del mensaje POST con el EAP-PSK-4
location_path3 = REAP3.opt.location_path
# Me aseguro de que location_path1 no es None y que es una tupla ('auth', 'eap', 'counter')
if location_path3 and isinstance(location_path3, tuple):
    # Obtengo el primer elemento de la tupla ('auth') y lo almaceno en val1
    val7 = location_path3[0]
    # Obtengo el segundo elemento de la tupla ('eap') y lo almaceno en val2
    val8 = location_path3[1]
    # Obtengo el tercer elemento de la tupla ('counter') y lo almaceno en val3
    val9 = location_path3[2]
    # Concateno los valores obtenidos con '/' para formar la URI completa
    uri_eapsuccess = val7 + '/' + val8 + '/' + val9
    # Imprimo la URI extraida, que sera la URI del mensaje POST con EAP-PSK-1 que se enviara a continuacion
    print("URI del mensaje EAP-SUCCESS:", uri_eapsuccess)
    print("")
# Si location_path1 es None o no tiene el formato esperado, imprimo un mensaje de error
else:
    print("No se ha encontrado la opción LOCATION_PATH o el formato no es el esperado")
```

Figura 6.59.- Main del script controller.py parte 12

Tras esto, establezco que el mensaje EAP que va a ir en el atributo EAP-Message del siguiente mensaje RADIUS Access-Request que se va a enviar al EAP Server (servidor RADIUS) sea EAP-PSK-4, el cual es el payload de la respuesta CREATED enviada por el EAP Peer (REAP2). También, establezco que la próxima vez que haga una llamada a la función sendNextMessageToRADIUS de la clase EAP_Authenticator del cliente RADIUS se ejecute la parte del código relativa al estado de autenticación EAP_SUCCESS.

Seguidamente, hago las llamadas necesarias al script cliente_radius.py, hasta obtener el mensaje Success (EAP SUCCESS). El proceso para obtener el mensaje EAP SUCCESS se va a explicar de forma generalizada a continuación.

En primer lugar, como aún estoy dentro del estado de autenticación EAP_PSK_3, imprimo por consola las variables del este estado RADIUS de la instancia eap_auth. Estas variables serán: la dirección IP del servidor RADIUS (192.168.217.128), la dirección IP del NAS (127.0.0.1), el puerto de origen (por ejemplo 9906) y de destino (1812), el número de paquetes que se van a enviar en la transacción (1), el tiempo máximo de espera por la respuesta (5 segundos), el identificador de la transacción (3), el Shared Secret (testing123) y una serie de atributos propios de RADIUS entre los que están: el tipo de puerto del servidor NAS (Wireless), el ID de la estación de llamada (00-00-00-00-00-00), la información de la conexión (CON), el tamaño máximo en bytes de los paquetes que pueden ser enviados (1400), el estado (por ejemplo “e9a30e8e3a011d74c076d8a714fd3f29” que es el valor del atributo State del mensaje Access-Challenge anterior) y el mensaje EAP (“026b002b2fc05614e250c7162a99101151ec7e7b5a400000000153e9e09d48be505ffa3c8169e7d0f5b403”, que es el mensaje EAP-PSK-4 recibido del EAP Peer). A partir de estas variables construyo, empleando la función genNextRadiusMessageFromState, el siguiente mensaje RADIUS Access-Request que envío usando la función sendNextMessageToRADIUS al EAP Server (servidor RADIUS), cuyo atributo EAP-Message contiene el cuarto mensaje EAP-PSK cumpliendo con lo que había fijado previamente. El servidor RADIUS responde a este mensaje con un Access-Accept en cuyo atributo EAP-Message está el mensaje EAP SUCCESS, el cual guardo en la variable payload4. Esto puedo hacerlo puesto que al enviar este Access-Request al EAP Server, entro dentro del último estado de autenticación del cliente RADIUS, que es el estado EAP_SUCCESS. En este estado, el código del script cliente_radius.py extrae el contenido

del atributo EAP-Message del mensaje Access-Accept que responde el servidor RADIUS. Además, retorna dos atributos adicionales del protocolo RADIUS, MS-MPPE-Recv-Key y MS-MPPE-Send-Key, que están relacionados con el cifrado de datos en las sesiones de autenticación y autorización.

El atributo MS-MPPE-Recv-Key contiene la clave de cifrado que el cliente (cliente RADIUS del EAP Authenticator) usará para recibir datos cifrados desde el servidor (servidor RADIUS que actúa como EAP Server). El servidor RADIUS cifra esta clave y la incluye como atributo en el paquete de respuesta Access-Challenge que contiene el mensaje Success, para que el cliente pueda descifrarla y utilizarla en la sesión.

Por su parte, el atributo MS-MPPE-Send-Key contiene la clave que el servidor usará para recibir datos cifrados desde el cliente. Esta clave también se cifra y se envía como un atributo en el paquete de respuesta Access-Challenge que contiene el mensaje Success.

De modo que después de una autenticación exitosa, el servidor RADIUS que actúa como EAP Server envía un mensaje Access-Accept al cliente RADIUS del EAP Authenticator. El Access-Accept contiene estos dos atributos, cifrados típicamente usando MD5 y una clave compartida (RADIUS Shared Secret) para protegerlos y evitar que sean interceptados en texto plano durante la transmisión. El EAP Authenticator deberá seguir un proceso criptográfico ligeramente complejo definido en el RFC2548 [73] para conseguir descifrar las claves de estos dos atributos. Para cubrir todo ese proceso, utilizo una función llamada decrypt_ms_key implementada por mí, que se encarga de descifrar estas claves y que requiere tres parámetros: la clave cifrada (en este caso, MS-MPPE-Recv-Key o MS-MPPE-Send-Key), el campo Authenticator del mensaje Access-Request previo al Access-Accept (obtenido accediendo al atributo req_auth de la instancia eap_auth), y el Shared Secret compartido entre el cliente y el servidor RADIUS (en este caso, “testing123”).

Una vez descifradas, la clave del atributo MS-MPPE-Recv-Key representa la primera mitad de la MSK (32 primeros bytes) y la clave del atributo MS-MPPE-Recv-Key representa la segunda mitad de la MSK (32 últimos bytes). Cuando ambas claves se concatenan (MSK=MS-MPPE-Recv-Key || MS-MPPE-Send-Key), se obtiene la MSK completa de 64 bytes, esencial para establecer un canal seguro entre el EAP Peer y el EAP Authenticator y

la que se deberá emplear para derivar las claves necesarias para crear el contexto de seguridad OSCORE del EAP Authenticator con el que poder proteger el mensaje EAP SUCCESS que se enviará al EAP Peer.

```
# Fijo que el mensaje EAP que va en el atributo EAP-Message sea EAP_PSK_MSG_4 para que la proxima vez que haga una
# llamada al cliente RADIUS, el mensaje RADIUS a enviar al servidor RADIUS se construya con ese contenido
eap_auth.set_eap_message(REAP3.payload.hex())
# Fijo el estado a EAP_SUCCESS para que la proxima vez que haga una llamada al cliente RADIUS se ejecute el codigo de dicho estado
eap_auth.set_state(EAPAuthState.EAP_SUCCESS)
# Imprimo las variables del estado actual de RADIUS dentro del objeto eap_auth
print(vars(eap_auth._radiusState))
# Genero el siguiente mensaje RADIUS a partir del estado actual (EAP_SUCCESS). Es decir, ahora estoy generando el mensaje
# RADIUS que contiene el EAP-PSK-4 que acabo de recibir del par EAP
eap_auth.genNextRadiusMessageFromState()
# Como ya estoy dentro del estado EAP_SUCCESS, puedo guardar el mensaje EAP-SUCCESS en la variable payload4, asi
# como las claves MS-MPPE-Recv-Key y MS-MPPE-Send-Key ya que es lo que me retorna ese estado
payload4, MPPE_REC_KEY, MPPE_SEND_KEY = eap_auth.sendNextMessageToRADIUS()
# Obtengo el campo Authenticator del mensaje Access-Accept previo al Access-Accept. El EAP Authenticator la usara
# para descifrar las claves MS y asi poder calcular la MSK necesaria para crear su contexto de seguridad OSCORE
req_authenticator = bytes.fromhex(eap_auth.req_auth)
# Defino el valor de secret (es el Shared Secret  compartido entre el cliente y el servidor RADIUS)
secret = b"testing123"
# Descifro la MS-MPPE-Recv-Key (seran los primeros 32 bytes de la MSK)
print("")
print("Proceso de descifrado de la MS-MPPE-Recv-Key:")
decrypted_recv_key = decrypt_ms_key(bytes.fromhex(MPPE_REC_KEY), req_authenticator, secret)
# Descifro la MS-MPPE-Send-Key (seran los ultimos 32 bytes de la MSK)
print("Proceso de descifrado de la MS-MPPE-Send-Key:")
decrypted_send_key = decrypt_ms_key(bytes.fromhex(MPPE_SEND_KEY), req_authenticator, secret)
# Calculo la MSK concatendo la MS-MPPE-Recv-Key descifrada con la MS-MPPE-Send-Key descifrada (64 bytes en total)
MSK = bytes.fromhex(decrypted_recv_key.hex() + decrypted_send_key.hex())
# Imprimo su valor en hexadecimal para comprobar que es correcto
print("\nMSK obtenida tras descifrar las claves MS:", MSK.hex())
```

Figura 6.60.- Main del script controller.py parte 13

A continuación, al mensaje de éxito (EAP SUCCESS) recibido procedente del EAP Server le añado una estructura CBOR de un sólo campo: un tiempo de vida de sesión, en segundos (establecí 28800 segundos, es decir, 8 horas que es el tiempo predeterminado), que se representa con un número entero sin signo. Esta estructura CBOR es opcional, pero decidí incorporarla para que el mensaje estuviese completo.

Una vez que ya tengo serializada esta información (coap_eap_info2) en la estructura CBOR (cbor_payload2), establezco el payload de la petición CoAP que contiene el mensaje Success (payload_eap_success). El payload de esta petición estará formado por la concatenación de la variable payload4, que contiene el mensaje EAP SUCCESS recibido del EAP Server y de la estructura CBOR construida (cbor_payload2).

Después, construyo la petición CoAP que contiene el mensaje Success, a la que llamo EAPSUCCESSS, con código POST, con el content format OCTETSTREAM (debería de ser CoAP-EAP), con el payload que acabo de establecer (payload_eap_success) y con la uri "coap://localhost/"+uri_eapsuccess". Dentro de esta URI, localhost es la dirección IP de loopback (normalmente 127.0.0.1 en IPv4) que se emplea para vincular este mensaje con el contexto de servidor del EAP Peer, ya que escucha en esa dirección. El valor de uri_eapsuccess es la URI construida a partir de los elementos de la tupla de la opción Location_Path de la respuesta con el mensaje EAP-PSK-4 (recordemos que uri_eapsuccess = auth/eap/4). A esta petición le tengo que asignar el Type (en este caso el tipo de mensaje es CON, puesto que requiere ser confirmado con una respuesta de tipo ACK, garantizando así una entrega confiable) y el Message ID (será el mismo Message ID que el de la respuesta con el mensaje EAP-PSK-4, el cual está guardado en la variable ID1, incrementado en uno).

```
# La MSK recibida del servidor RADIUS sera la concatenacion de la MS-MPPE-Recv-Key y de la MS-MPPE-Send-Key descifradas
# Imprimo la MSK recibida del servidor RADIUS
# print("MSK recibida del servidor:", MPPE_REC_KEY + MPPE_SEND_KEY)
# Defino la estructura CoAP-EAP_Info2 como un diccionario que contiene informacion sobre el tiempo de vida de la sesion
coap_eap_info2 = {
    # Tiempo de vida de la sesion en segundos (8 horas que es el tiempo predeterminado establecido en el Draft)
    1: 28800
}
# Serializo la estructura CoAP-EAP_Info a formato CBOR para su transporte o almacenamiento
cbor_payload2 = cbor2.dumps(coap_eap_info2)
# Concateno el payload del mensaje EAP-Success recibido del servidor EAP con la estructura CBOR que contiene el tiempo de vida de la sesion
payload_eap_success = payload4 + cbor_payload2.hex()
# Imprimo el mensaje EAP-SUCCESS con la estructura CBOR ya concatenada
print("El payload completo con el mensaje EAP-SUCCESS es este:", payload_eap_success)
print("")
# Construyo el mensaje EAP-SUCCESS con codigo POST a la uri indicada y con el payload y el content_format establecidos anteriormente
EAPSUCCESS = aiocoap.Message(code=Code.POST, payload=bytes.fromhex(payload_eap_success), uri="coap://localhost/" + uri_eapsuccess,
                             content_format=content_format)
# Asigno el tipo de mensaje para EAP-SUCCESS, en este caso lo configuro como Confirmable (CON)
EAPSUCCESS.mtype = aiocoap.CON
# Asigno un ID unico para el mensaje EAP-SUCCESS usando el ID de mensaje del EAP-PSK-4 incrementado en 1
EAPSUCCESS.mid = ID1 + 1
# Codifico el mensaje EAP-SUCCESS en hexadecimal
EAPSUCCESS_HEX = EAPSUCCESS.encode().hex()
```

Figura 6.61.- Main del script controller.py parte 14

El siguiente paso es definir las variables que voy a necesitar para construir el contexto de seguridad OSCORE del EAP Authenticator.

Para empezar, obtengo la Master Secret de tamaño 16 bytes (Master Secret = KDF (MSK, CS | ”COAP-EAP OSCORE Master Secret”, length)) y la Master Salt de tamaño 8

bytes (Master Salt = KDF (MSK, CS | ”COAP-EAP OSCORE Master Salt”, length)) empleando la función derive_OSCORE_keys desarrollada en el script calculos_OSCORE.py. Esta función recibe como parámetros la MSK resultante de concatenar las claves MS descifradas y la cadena CS, que consiste en la concatenación del contenido de la negociación de cipher suite, es decir, la lista de cipher suites enviadas por el EAP Authenticator en el EAP Request ID concatenada a la cipher suite seleccionada por el EAP Peer en el EAP Response ID. A continuación, asigno la Master Secret y la Master Salt obtenidas a la variable SECRET y SALT respectivamente, defino un Context ID de OSCORE para el EAP Authenticator (tiene que ser el mismo que el del EAP Peer) y establezco como algoritmos AEAD y HASH a utilizar los elegidos por el EAP Peer en la negociación de cipher suite (son los algoritmos predeterminados).

```
# Concateno el contenido de la negociacion del conjunto de cifrado, es decir, la lista de conjuntos de cifrado
# enviados por el autenticador EAP en el EAP Request ID concatenada a la opcion seleccionada por el par EAP en el
# EAP Response ID y lo convierto a hexadecimal
CS = cbor_payload1[2:8].hex() + ResponseId.payload[12:14].hex()
# Imprimo la cadena CS resultante
print("CS:", CS)
# Derivo las claves OSCORE (Master Secret y Master Salt) usando la MSK y el CS
MASTER_SECRET, MASTER_SALT = derive_oscore_keys(MSK, CS)
# Asigno a la variable SECRET el valor de la Master Secret y a la variable SALT el valor de la Master Salt obtenidos
# en la derivacion de las claves OSCORE convertidos a bytes
SECRET = bytes.fromhex(MASTER_SECRET)
SALT = bytes.fromhex(MASTER_SALT)
# Defino el ID del contexto OSCORE del autenticador EAP y lo convierto a bytes
ID_CTX = bytes.fromhex("37cbf3210017a2d3")
# Establezco el algoritmo AEAD y el algoritmo hash seleccionados por el par EAP
aead_algorithm = aiocoap.oscore.algorithms[algoritmo_AEAD]  # Algoritmo AES_CCM
fun_hash = aiocoap.oscore.hashfunctions[algoritmo_hash]  # Funcion hash SHA-256
```

Figura 6.62.- Main del script controller.py parte 15

El proceso seguido para crear el contexto de seguridad OSCORE del EAP Authenticator y proteger el mensaje EAP SUCCESS se va a explicar a continuación.

En primer lugar, defino una clase de contexto de seguridad personalizada llamada OSCORESecurityContext que hereda de la clase CanProtect (permite proteger los mensajes CoAP), de la clase CanUnprotect (maneja la recepción y el procesamiento de mensajes que han sido protegidos, asegurando que se puedan desproteger y validar correctamente) y de la

clase SecurityContextUtils (ofrece utilidades relacionadas con el contexto de seguridad como la derivación de la Sender Key, de la Recipient Key y del Common IV), todas ellas desarrolladas en el script OSCORE.py de la biblioteca aiocoap. Esta clase consta de un método post_seqnoincrease al que se llama al intentar incrementar el número de secuencia, pero no hace nada (no lo voy a utilizar), sólo lo defino puesto que es necesario para que la clase funcione correctamente ya que post_seqnoincrease está definido en CanProtect como un método abstracto, lo que implica que cualquier clase que herede de CanProtect debe implementar este método.

```
class OscoreSecurityContext(CanProtect, CanUnprotect, SecurityContextUtils):
    # Metodo que se llama al intentar incrementar el numero de secuencia, pero no hace nada (es necesario
    # definirlo para que la clase funcione ya que post_seqnoincrease esta definido en CanProtect como un metodo
    # abstracto, lo que implica que cualquier clase que herede de CanProtect debe implementar este metodo)
    def post_seqnoincrease(self):
        pass
```

Figura 6.63.- Main del script controller.py parte 16

Tras esto, creo el contexto de seguridad OSCORE del EAP Authenticator (secctx) instanciando la clase OSCORESecurityContext y establezco el algoritmo AEAD, la función HASH, el Context ID, el Sender ID (el EAP Authenticator utiliza el Recipient ID del EAP Peer (RID-I) como Sender ID para su Sender Context en OSCORE), el Recipient ID (el EAP Authenticator usa el Recipient ID del EAP Authenticator (RID-C) como Recipient ID para su Recipient Context) y el Sender Sequence Number (por ejemplo 20, tiene que ser el mismo que el que se utilice en el EAP Peer) que se van a utilizar en este contexto OSCORE del EAP Authenticator. Además, consigo la Sender Key (**sender_key = KDF (master_salt, master_secret, sender_id, "Key")**), la Recipient Key (**recipient_key = KDF (master_salt, master_secret, recipient_id, "Key")**) y el Common IV (**common_iv = KDF (master_salt, master_secret, b"", "IV")**) llamando a la función derive_keys.

```
# Creo el contexto de seguridad del autenticador EAP instanciando la clase OscoreSecurityContext
secctx = OscoreSecurityContext()
# Establezco el algoritmo AEAD que voy a utilizar en el contexto de seguridad
secctx.alg_aead = aead_algorithm
# Establezco la funcion hash que voy a utilizar en el contexto de seguridad
secctx.hashfun = fun_hash
# El autenticador EAP utiliza el ID de Destinatario del par EAP (RID-I) como ID de Remitente para su contexto de remitente en OSCORE
secctx.sender_id = rid_i
# El autenticador EAP usa el ID de Destinatario del autenticador EAP (RID-C) como ID de Destinatario para su contexto de destinatario
secctx.recipient_id = RID_C
# Establezco el ID de contexto
secctx.id_context = ID_CTX
# Llamo a la funcion derive_keys para obtener la clave de remitente, la clave de destinatario y el IV comun a partir
# del algoritmo AEAD, la funcion hash y el ID de contexto ya configurados de antemano, y a partir de la Master Secret
# y de la Master Salt pasadas como parametros
secctx.derive_keys(SALT, SECRET)
# Fijo el numero de secuencia del remitente en 20
secctx.sender_sequence_number = 20
```

Figura 6.64.- Main del script controller.py parte 17

Recordemos que para cada extremo, el contexto de seguridad se compone de un "Common Context", un "Sender Context" y un "Recipient Context".

El Common Context del contexto OSCORE del EAP Authenticator estará formado por el algoritmo AEAD, el algoritmo HKDF (función HASH), la Master Secret, la Master Salt, el Context ID y el Common IV; el Sender Context del contexto OSCORE del EAP Authenticator estará formado por el Sender ID, la Sender Key y el Sender Sequence Number; y el Recipient Context del contexto OSCORE del EAP Authenticator estará formado por el Recipient ID y la Recipient Key.

```
# Imprimo los parametros del Contexto Comun del contexto de seguridad OSCORE del autenticador EAP
print("Parámetros del Contexto Común del contexto de seguridad OSCORE del autenticador EAP:")
print("Algoritmo AEAD:", algoritmo_AEAD)
print("Algoritmo HKDF:", algoritmo_hash)
print("Secreto Maestro (Master Secret):", MASTER_SECRET)
print("Sal Maestra (Master Salt):", MASTER_SALT)
print("ID del Contexto (Context ID):", ID_CTX.hex())
print("IV Común (Common IV):", secctx.common_iv.hex())
# Imprimo los parametros del Contexto de Remitente del contexto de seguridad OSCORE del autenticador EAP
print("Parámetros del Contexto de Remitente del contexto de seguridad OSCORE del autenticador EAP:")
print("ID del Remitente (Sender ID):", secctx.sender_id.hex())
print("Clave del Remitente (Sender Key):", secctx.sender_key.hex())
print("Número de secuencia del Remitente (Sender sequence number):", secctx.sender_sequence_number)
# Imprimo los parametros del Contexto de Destinatario del contexto de seguridad OSCORE del autenticador EAP
print("Parámetros del Contexto de Destinatario del contexto de seguridad OSCORE del autenticador EAP:")
print("ID del Destinatario (Recipient ID):", secctx.recipient_id.hex())
print("Clave del Destinatario (Recipient Key):", secctx.recipient_key.hex())
print("")
```

Figura 6.65.- Main del script controller.py parte 18

Lo que hay que hacer ahora es proteger la petición que contiene el mensaje EAP SUCCESS con OSCORE. Para ello, decodifico la petición desde el formato hexadecimal a un objeto de mensaje CoAP de la clase aiocoap al que llamo unprotected y la protejo empleando el contexto de seguridad OSCORE del EAP Authenticator mediante la función protect. A esta petición protegida la llamo protected_message, le asigno el Type, el Message ID y el Token del mensaje desprotegido (unprotected) al mensaje protegido (protected_message) y, finalmente, codifico este mensaje protegido a bytes para obtener la petición con el mensaje EAP SUCCESS protegida a la que llamo encoded_protected_message.

```
# Decodifico el mensaje EAP-SUCCESS desde el formato hexadecimal a un objeto de mensaje CoAP sin proteccion
unprotected = aiocoap.Message.decode(bytes.fromhex(EAPSUCCESS_HEX))
# Protejo el mensaje sin proteccion usando el contexto de seguridad OSCORE del autenticador EAP
protected_message, _ = secctx.protect(unprotected)
# Asigno el ID de mensaje del mensaje desprotegido al mensaje protegido
protected_message.mid = unprotected.mid
# Asigno el token del mensaje desprotegido al mensaje protegido
protected_message.token = unprotected.token
# Asigno el tipo de mensaje del mensaje desprotegido al mensaje protegido
protected_message.mtype = unprotected.mtype
# Codifico el mensaje protegido a bytes
encoded_protected_message = protected_message.encode()
# Imprimo el mensaje EAP-SUCCESS protegido en formato hexadecimal
print("Mensaje EAP-SUCCESS protegido por OSCORE:", encoded_protected_message.hex())
```

Figura 6.66.- Main del script controller.py parte 19

Por último, construyo de nuevo la petición CoAP que contiene el mensaje Success, a la que llamo EAPSUCCESSS_PROT, con código POST, con el content format OCTETSTREAM (debería de ser CoAP-EAP), incluyendo en el payload la petición completa con el mensaje EAP SUCCESS protegida con OSCORE (encoded_protected_message) y con la uri "coap://localhost/"+uri_eapsuccess". Dentro de esta URI, localhost es la dirección IP de loopback (normalmente 127.0.0.1 en IPv4) que se emplea para vincular este mensaje con el contexto de servidor del EAP Peer, ya que escucha en esa dirección. El valor de uri_eapsuccess es la URI construida a partir de los elementos de la tupla de la opción Location_Path de la respuesta con el mensaje EAP-PSK-4 (uri_eapsuccess = auth/eap/4).

Tras construir la petición con el mensaje EAP SUCCESS protegida con OSCORE, la envío al EAP Peer, espero a recibir la respuesta y guardo el valor del Message ID de esta

respuesta en la variable ID2. Utilizaré este valor más adelante para definir el Message ID de la petición DELETE. Si el código de la respuesta recibida procedente del EAP Peer es CHANGED, significa que la autenticación se ha completado con éxito y muestro por consola el código y el mensaje de respuesta CHANGED protegido por OSCORE. En cambio si el código de la respuesta recibida es UNAUTHORIZED, significa que ha ocurrido un error en la autenticación y muestro por consola el código y el payload relativo a esa respuesta UNAUTHORIZED enviada por el EAP Peer.

```
# Construyo un nuevo mensaje CoAP para enviar en el payload el EAP-SUCCESS protegido con OSCORE a la uri indicada y
# el content_format establecido anteriormente
EAPSUCCESS_PROT = aiocoap.Message(code=Code.POST, payload=bytes.fromhex(encoded_protected_message.hex()),
                                  uri="coap://localhost/" + uri_eapsuccess, content_format=content_format)
# Indico que estoy enviando la peticion POST con el mensaje EAP-SUCCESS al par EAP
print("Enviando POST con el mensaje SUCCESS del protocolo EAP protegido con OSCORE al dispositivo IOT...")
# Envio una solicitud (EAP-SUCCESS) a traves del contexto del cliente CoAP y espero la respuesta final
REAPSUCCESS = await context.request(EAPSUCCESS_PROT).response
# Guardo el valor del ID de mensaje del mensaje CHANGED recibido
ID2 = REAPSUCCESS.mid
if str(REAPSUCCESS.code) == "2.04 Changed":
    # Muestro la informacion del mensaje CHANGED protegido con OSCORE recibido
    print("MENSAJE CHANGED PROTEGIDO CON OSCORE RECIBIDO CON:")
    # Imprimo el codigo
    print("Codigo:", REAPSUCCESS.code)
    # Imprimo el payload en binario
    print("Payload binario que contiene el mensaje CHANGED protegido:", REAPSUCCESS.payload)
    # Imprimo el payload en hexadecimal
    print("Payload hexadecimal que contiene el mensaje CHANGED protegido:", REAPSUCCESS.payload.hex())
    print("")
else:
    # Muestro la informacion del mensaje UNAUTHORIZED recibido
    print("MENSAJE UNAUTHORIZED RECIBIDO CON:")
    # Imprimo el codigo
    print("Codigo:", REAPSUCCESS.code)
    # Imprimo el payload en binario
    print("Payload:", REAPSUCCESS.payload)
    # Imprimo el payload en hexadecimal
    print("Payload hexadecimal:", REAPSUCCESS.payload.hex())
```

Figura 6.67.- Main del script controller.py parte 20

El main del script controller.py también implementa una serie de líneas de código destinadas a verificar el comportamiento del sistema en situaciones donde el EAP Peer recibe una repetición de una solicitud enviada por el EAP Authenticator después de un cierto intervalo de tiempo.

El caso de prueba pretende simular una situación en la cual:

1. El EAP Authenticator envía una solicitud al EAP Peer. Supongamos, por ejemplo, que envía la petición que contiene el mensaje EAP-PSK-1. El EAP Peer responde con el mensaje de respuesta CREATED que contiene el mensaje EAP-PSK-2 y el flujo de autenticación continúa de forma normal.

2. Después de unos instantes, el EAP Authenticator envía la misma solicitud nuevamente, ya que se simula una retransmisión tardía o una solicitud duplicada. En este momento, el EAP Peer ya ha eliminado el recurso correspondiente a la solicitud original, en este caso el recurso SecondResource, ya que el EAP Peer una vez que devuelve la respuesta y asigna un nuevo recurso, elimina el anterior.

Como resultado:

- Al recibir la solicitud duplicada, el EAP Peer al no encontrar el estado correspondiente en su memoria, responde con un código de error ‘4.04 Not Found’.

- Luego, el sistema continúa con el flujo de operación normal, ignorando la solicitud duplicada y evitando cualquier impacto en el proceso de autenticación en curso o en futuros intentos.

En estas líneas de código lo único que hago es construir de nuevo la petición CoAP que contiene el primer mensaje EAP-PSK, a la que llamo EAP_REP, con código POST, con el content format OCTETSTREAM (debería de ser CoAP-EAP), con un payload que será el contenido de la variable payload2 (mensaje EAP-PSK-1) y con la uri "coap://localhost/"+uri_eappsk1", donde uri_eappsk1 es auth/eap/3.

Tras construir la petición duplicada, la envío al EAP Peer, espero a recibir la respuesta NOT FOUND, a la que llamo RES_EAP_REP y, al recibirla, muestro por consola la información más importante (código y payload) relativa a esa respuesta.

```
# Este es un ejemplo de mensaje POST al segundo recurso (SecondResource) para comprobar si los recursos se van eliminando correctamente
# El par EAP me debería responder con un mensaje con codigo 4.04 NOT FOUND
# Construyo de nuevo la peticion CoAP con el mensaje EAP-PSK-1 como si fuera una solicitud duplicada enviada tras un periodo de tiempo
EAP_REP = aiocoap.Message(code=Code.POST, payload=bytes.fromhex(payload2), uri="coap://localhost/" + uri_eappsk1, content_format=content_format)
# Indico que estoy enviando al par EAP la peticion POST al segundo recurso
print("Enviando POST con URI REPETIDA al dispositivo IOT...")
# Envio la solicitud (POST a la uri repetida) a través del contexto del cliente CoAP y espero la respuesta 4.04 NOT FOUND
RES_EAP_REP = await context.request(EAP_REP).response
# Muestro la informacion del mensaje 4.04 NOT FOUND recibido
print("MENSAJE NOT FOUND RECIBIDO CON:")
# Imprimo el codigo
print("Codigo:", RES_EAP_REP.code)
# Imprimo el payload en hexadecimal (vacio)
print("Payload hexadecimal:", RES_EAP_REP.payload.hex())
print("")
```

Figura 6.68.- Main del script controller.py parte 21

Por último, el script controller.py implementa una serie de líneas de código destinadas a verificar el comportamiento del sistema cuando el EAP Authenticator envía una petición DELETE al último recurso si considera necesario eliminar el "estado" CoAP-EAP del EAP Peer antes de que expire, es decir, si por algún motivo quiere que el dispositivo IoT deje de estar autenticado antes de que caduque el tiempo de vida de la sesión. El EAP Peer deberá responder con un mensaje de respuesta ‘2.02 DELETED’. Tanto la solicitud DELETE enviada por el EAP Authenticator como la respuesta DELETED enviada por el EAP Peer deberán ir protegidas por OSCORE.

Lo que hago en estas líneas de código es lo siguiente:

Primero, construyo la petición DELETE al último recurso (FourthResource) a la que llamo PET_DELETE, con código DELETE, con el content format OCTETSTREAM (debería de ser CoAP-EAP), con un payload que va a ser, por ejemplo, la cadena “Mensaje DELETE de prueba” codificada en binario y con la uri "coap://localhost/"+uri_eapsuccess", donde uri_eapsuccess es auth/eap/4. A esta petición le tengo que asignar el Type (en este caso el tipo es CON, puesto que requiere ser confirmado con una respuesta de tipo ACK, garantizando así una entrega confiable) y el Message ID (será el mismo Message ID que el de la respuesta con el mensaje CHANGED, el cual está guardado en la variable ID2, incrementado en dos. Será incrementado en dos y no en uno porque antes de esto hago la prueba del mensaje duplicado).

Lo segundo que hay que hacer es proteger la petición DELETE con OSCORE. Para ello, decodifico la petición desde el formato hexadecimal a un objeto de mensaje CoAP de la clase aiocoap al que llamo unprotected_del y la protejo empleando el contexto de seguridad OSCORE del EAP Authenticator mediante la función protect. A esta petición protegida la llamo protected_del, le asigno el tipo de mensaje, el Message ID y el token del mensaje desprotegido (unprotected_del) al mensaje protegido (protected_del) y, finalmente, codifico este mensaje protegido a bytes para obtener la petición DELETE protegida, a la que llamo encoded_protected_del.

```
# Este es un ejemplo de mensaje DELETE al ultimo recurso para comprobar si el par EAP responde adecuadamente
# El peer me debería responder con un mensaje con codigo 2.02 DELETED
# Construyo el mensaje de prueba con codigo DELETE a la uri indicada, con un payload aleatorio y el content_format establecido anteriormente
PET_DELETE = aiocoap.Message(code=Code.DELETE, payload="Menssaje DELETE de prueba".encode(),
                             uri="coap://localhost/" + uri_eapsuccess, content_format=content_format)
# Asigno el tipo de mensaje para el mensaje DELETE, en este caso lo configuro como Confirmable (CON)
PET_DELETE.mtype = aiocoap.CON
# Asigno un ID unico para el mensaje DELETE usando el ID de mensaje del mensaje CHANGED incrementado en 2
PET_DELETE.mid = ID2 + 2
# Codifico el mensaje DELETE en hexadecimal
PET_DELETE_HEX = PET_DELETE.encode().hex()
# Decodifico el mensaje DELETE desde el formato hexadecimal a un objeto de mensaje CoAP sin proteccion
unprotected_del = aiocoap.Message.decode(bytes.fromhex(PET_DELETE_HEX))
# Protejo el mensaje sin proteccion usando el contexto de seguridad OSCORE del autenticador EAP
protected_del, _ = secctx.protect(unprotected_del)
# Asigno el ID de mensaje del mensaje desprotegido al mensaje protegido
protected_del.mid = unprotected_del.mid
# Asigno el token del mensaje desprotegido al mensaje protegido
protected_del.token = unprotected_del.token
# Asigno el tipo de mensaje del mensaje desprotegido al mensaje protegido
protected_del.mtype = unprotected_del.mtype
# Codifico el mensaje protegido a bytes
encoded_protected_del = protected_del.encode()
# Imprimo el mensaje EAP-SUCCESS protegido en formato hexadecimal
print("Mensaje DELETE protegido por OSCORE:", encoded_protected_del.hex())
```

Figura 6.69.- Main del script controller.py parte 20

Finalmente, construyo de nuevo la petición DELETE, a la que llamo PET_DELETE_PROT, con código DELETE, con el content format OCTETSTREAM (debería de ser CoAP-EAP), incluyendo en el payload la petición DELETE completa protegida con OSCORE (encoded_protected_del) y con la uri "coap://localhost/"+uri_eapsuccess", donde localhost es la dirección IP de loopback (normalmente 127.0.0.1 en IPv4) que se emplea para vincular este mensaje con el contexto de servidor del EAP Peer y donde uri_eapsuccess es auth/eap/4.

Tras construir la petición DELETE protegida con OSCORE, la envío al EAP Peer, espero a recibir la respuesta y, al recibirla, muestro por consola el código y la respuesta DELETED protegida por OSCORE.

```
# Construyo un nuevo mensaje CoAP para enviar en el payload el mensaje DELETE protegido con OSCORE a la uri indicada
# y el content_format establecido anteriormente
PET_DELETE_PROT = aiocoap.Message(code=Code.DELETE, payload=bytes.fromhex(encoded_protected_del.hex()),
                                  uri="coap://localhost/" + uri_eapsuccess, content_format=content_format)
# Indico que estoy enviando al peer la peticion DELETE protegida por OSCORE al ultimo recurso
print("Enviando el mensaje DELETE protegido con OSCORE al dispositivo IOT...")
# Envio la solicitud (DELETE a la ultima uri) a través del contexto del cliente CoAP y espero la respuesta 2.02 DELETED
RES_PET_DELETE = await context.request(PET_DELETE_PROT).response
# Muestro la informacion del mensaje 2.02 DELETED recibido
print("MENSAJE DELETED PROTEGIDO CON OSCORE RECIBIDO CON:")
# Imprimo el codigo
print("Codigo:", RES_PET_DELETE.code)
# Imprimo el payload en binario
print("Payload binario que contiene el mensaje DELETED protegido:", RES_PET_DELETE.payload)
# Imprimo el payload en hexadecimal (vacio)
print("Payload hexadecimal que contiene el mensaje DELETED protegido:", RES_PET_DELETE.payload.hex())
```

Figura 6.70.- Main del script controller.py parte 21

6.3.3.- Cliente RADIUS

Lo primero que hago en el script cliente_radius.py es definir una clase de enumeración, a la que llamo EAPAuthState, que me ayuda a organizar y manejar los distintos estados que puede tener el proceso de autenticación EAP en el cliente RADIUS. Este enfoque me permite asociar nombres significativos con valores numéricos específicos, facilitando la lectura y el mantenimiento del código, además de reducir el riesgo de errores.

Los estados que definí son los siguientes:

1. **REQ_ID_FIX:** Este estado tiene el valor 1 y se utiliza para gestionar un caso específico en el que se envía al EAP Server (servidor RADIUS) el mensaje EAP Response ID en un Access-Request, pero el servidor responde con un mensaje Access-Challenge pidiendo de nuevo la identidad en lugar de enviar el mensaje EAP-PSK-1. Para resolver este bug he creado este estado.

2. **EAP_PSK_1:** Este estado, con valor 2, representa la recepción del mensaje EAP-PSK-1. Este es el primer mensaje del protocolo EAP-PSK y su llegada indica que el

servidor respondió correctamente y que el cliente puede continuar con el flujo de autenticación.

3. **EAP_PSK_3:** Con el valor 3, este estado maneja la recepción del mensaje EAP-PSK-3. Este mensaje es otro paso en el protocolo EAP-PSK y su llegada indica que el servidor respondió correctamente y que el cliente puede continuar con el intercambio de autenticación.

4. **EAP_SUCCESS:** Este último estado, con valor 4, corresponde a la recepción del mensaje de éxito (EAP SUCCESS), lo que indica que la autenticación EAP se ha completado exitosamente.

```
# Defino una clase de enumeracion (Enum) para representar los diferentes estados de autenticacion EAP
9 usages
class EAPAuthState(Enum):
    # Defino el estado REQ_ID_FIX con el valor 1, el cual maneja el bug que se produce cuando se envia al servidor
    # RADIUS el EAP Response ID (el servidor responde con un Access-Challenge solicitando de nuevo la identidad del par
    # EAP en lugar de responder ya con el mensaje EAP-PSK-1, por lo que hay que enviar de nuevo el EAP Response ID)
    REQ_ID_FIX = 1
    # Defino el estado EAP_PSK_1 con el valor 2, el cual maneja la recepcion del mensaje EAP-PSK-1
    EAP_PSK_1 = 2
    # Defino el estado EAP_PSK_3 con el valor 3, el cual maneja la recepcion del mensaje EAP-PSK-3
    EAP_PSK_3 = 3
    # Defino el estado EAP_SUCCESS con el valor 4, el cual maneja la recepcion del mensaje EAP-SUCCESS
    # (representa el exito en la autenticacion EAP)
    EAP_SUCCESS = 4
```

Figura 6.71.- Clase EAPAuthState del script cliente_radius.py

Lo segundo que hago es definir otra clase llamada EAPAuth_AAA_connection, que modela una conexión AAA (Authentication, Authorization, Accounting) en el contexto del protocolo EAP. Esta clase representa la configuración básica de la conexión que el cliente RADIUS usará para comunicarse con el servidor, especificando los elementos esenciales de la conexión, como los puertos y la dirección IP.

Dentro de esta clase, se encuentra el método constructor __init__, que se ejecuta cada vez que se crea una instancia de EAPAuth_AAA_connection. Este método inicializa los siguientes atributos:

- **src_port:** Este atributo representa el puerto de origen desde el que el cliente establece la conexión. Inicialmente, lo configuro en 0 como valor por defecto, permitiendo definir el puerto específico en otro momento del flujo de ejecución si fuera necesario.

- **dst_port:** Este atributo representa el puerto de destino en el servidor RADIUS, al que se enviarán los paquetes de autenticación. Al igual que src_port, lo configuro en 0 de forma inicial.

- **dst_ip:** Este atributo contiene la dirección IP de destino del servidor RADIUS, a la cual se conecta el cliente. De momento, está inicializado como una cadena vacía ("") y tomará el valor de la IP del servidor en la configuración o durante la ejecución del programa.

```
# Defino una clase EAPAuth_AAA_connection que representa una conexion AAA (Authentication, Authorization, Accounting)
# para EAP
1 usage
class EAPAuth_AAA_connection:
    # Defino el metodo constructor __init__, que se ejecuta al crear una instancia de la clase
    def __init__(self):
        # Inicializo el atributo src_port, que representa el puerto de origen de la conexion, en 0
        self.src_port = 0
        # Inicializo el atributo dst_port, que representa el puerto de destino de la conexion, en 0
        self.dst_port = 0
        # Inicializo el atributo dst_ip, que representa la direccion IP de destino, como una cadena vacia
        self.dst_ip = ""
```

Figura 6.72.- Clase EAPAuth_AAA_connection del script cliente_radius.py

Tras esto creo la clase RadiusState, la cual prepara una conexión RADIUS configurando el servidor de autenticación, detalles de la sesión y atributos específicos de autenticación y autorización. Esta configuración incluye elementos estándar (como IPs y puertos) y otros atributos detallados (como el nombre de usuario, Shared Secret y atributos personalizados), que se inicializan a través de un constructor y permiten al cliente enviar una solicitud de autenticación adecuada al servidor. En resumen, los valores definidos en el constructor permiten mantener el estado y la configuración de la conexión y preparan al cliente para manejar solicitudes EAP en el marco de un protocolo RADIUS.

A continuación, detallo cada atributo que inicializo en el constructor __init__:

1. **Herencia de EAPAuth_AAA_connection:** Llamo a super().__init__() para heredar e inicializar los atributos src_port, dst_port y dst_ip desde la clase padre EAPAuth_AAA_connection.

2. **dst_ip:** Sobrescribo la dirección IP de destino (IP del servidor RADIUS) a la que se van a enviar los mensajes de autenticación, estableciéndola como la IP "192.168.217.128", que es la dirección IP de la máquina virtual.

3. **password:** Inicializado como una cadena vacía, este campo almacenará la contraseña del usuario necesaria para validar su identidad en algunos métodos de autenticación. No voy a utilizar este atributo.

4. **username:** Establece el nombre de usuario o NAI del dispositivo IoT como "usera", el cual será enviado al servidor en el proceso de autenticación.

5. **secret:** Este es el Shared Secret (secreto compartido) entre el cliente y el servidor RADIUS. Es fundamental para autenticar mensajes RADIUS y garantizar que provienen de una fuente confiable. Este atributo se inicializa con la cadena “testing123”.

6. **nasipaddr:** Es la dirección IP del NAS, el dispositivo que facilita la conexión entre el cliente y el servidor RADIUS. La defino como "127.0.0.1", lo cual indica que el NAS está en la misma máquina o en la red local.

7. **src_port:** Sobrescribo el puerto de origen de la conexión, estableciéndolo de manera aleatoria en un valor entre 1024 y 65535 para evitar conflictos de puerto.

8. **dst_port:** Sobrescribo el puerto de destino del servidor RADIUS, configurándolo en 1812, que es el puerto estándar para mensajes de autenticación RADIUS.

9. **timeout:** Define el tiempo máximo que esperará el cliente RADIUS por una respuesta del servidor antes de dar por terminada la solicitud o reintentarlo. Lo establezco en 5 segundos.

10. **packet_num:** Inicializado en 1, representa un contador de los paquetes que se envían al servidor RADIUS en cada transacción. Siempre voy a enviar un sólo Access-Request por eso lo establezco en 1.

11. **service_type:** Almacena el tipo de servicio solicitado en la conexión. Lo inicializo como una cadena vacía ya que no lo voy a utilizar.

```
# Defino la clase RadiusState para configurar los parametros de una conexion RADIUS y los atributos de una solicitud de
# autenticacion, permitiendo mantener el estado. Es decir, a traves de esta clase puedo almacenar toda la informacion
# relevante para la conexion y enviar los mensajes de autenticacion adecuados
1 usage
class RadiusState (EAPAuth_AAA_connection):
    # Defino el metodo constructor __init__, que inicializa los atributos necesarios para la conexion RADIUS
    def __init__(self):
        # Esto inicializa src_port, dst_port, y dst_ip desde la clase padre (EAPAuth_AAA_connection)
        super().__init__()
        # Inicializo el atributo host con la direccion IP del servidor RADIUS
        self.dst_ip = "192.168.217.128"
        # Inicializo el atributo password como una cadena vacia, que almacenara la contraseña del usuario
        self.password = ""
        # Inicializo el atributo username con el nombre de usuario (peer)
        self.username = "usera"
        # Inicializo el atributo secret como una cadena vacia, que almacenara el secreto compartido
        self.secret = "testing123"
        # Inicializo el atributo nasipaddr con la direccion IP del NAS (Network Access Server)
        self.nasipaddr = "127.0.0.1"
        # Inicializo el atributo src_port con un puerto de origen aleatorio entre 1024 y 65535
        self.src_port = random.randint( a: 1024,  b: 65535)
        # Inicializo el atributo dst_port con el puerto de destino por defecto para RADIUS (1812)
        self.dst_port = 1812
        # Inicializo el atributo timeout en 5 segundos, que es el maximo tiempo de espera para respuestas
        self.timeout = 5
        # Inicializo el atributo packet_num en 1, que cuenta el numero de paquetes enviados
        self.packet_num = 1
        # Inicializo el atributo service_type como una cadena vacia (define el tipo de servicio)
        self.service_type = ""
```

Figura 6.73.- Constructor de la clase RadiusState del script cliente_radius.py parte 1

12. **nas_port_type:** Este atributo identifica el tipo de puerto del NAS. Al configurarse como "wireless", sugiere que el cliente se conectará mediante una red inalámbrica.

13. **calling_station_id y called_station_id:** Los atributos Calling-Station-Id y Called-Station-Id en RADIUS son utilizados para proporcionar información sobre las estaciones que inician y reciben la solicitud de conexión, respectivamente. El atributo Calling-Station-Id se refiere a la dirección MAC o identificador único de la estación (o dispositivo) que está iniciando la solicitud de acceso a la red (lo defino como "00-00-00-00-00-00" que es la MAC por defecto) y el atributo Called-Station-Id

identifica la estación o punto de acceso al que el usuario está intentando conectarse (lo defino como una cadena vacía, no lo voy a utilizar).

14. **vsa_id, vsa_type, vsa_value, vsa_coding y vsa_trim:** Estos atributos permiten agregar atributos específicos del proveedor (Vendor-Specific Attributes) al mensaje RADIUS. Un VSA puede incluir cualquier dato personalizado por el proveedor, codificado como una cadena o en otro formato. vsa_coding especifica cómo debe interpretarse el valor (string en este caso). El resto de estos atributos los inicializo como cadenas vacías ya que no los voy a utilizar.

15. **framed_ip:** Atributo utilizado para asignar una dirección IP dinámica a un usuario que se conecta a una red, generalmente en el contexto de autenticación de acceso remoto o conexiones VPN. Está inicializado como una cadena vacía porque no lo voy a usar.

16. **radiusid:** Este atributo es el ID del mensaje RADIUS y se utiliza para identificar la transacción. Lo inicializo en 0 y cada vez que se envíe un nuevo mensaje Access-Request se incrementará en uno.

17. **state:** Es un atributo que se utiliza para mantener el estado de una sesión durante el proceso de autenticación y autorización. Principalmente se emplea en los protocolos de autenticación que involucran múltiples etapas o desafíos. Funciona de la siguiente manera:
    - El cliente envía una solicitud Access-Request al servidor RADIUS. El servidor responde con un Access-Challenge que contiene un atributo State y que proporciona algún dato relevante para la continuación del proceso de autenticación (como un desafío de autenticación).
    - El cliente responde al servidor, incluyendo un atributo State (con el mismo valor que el atributo State del Access-Challenge recibido) en su siguiente solicitud Access-Request. El servidor utiliza esa información para continuar con el proceso de autenticación o autorización.

    Está inicializado como una cadena vacía que se actualizará durante el proceso de autenticación.

18. **eap:** Este atributo almacena un mensaje EAP en formato hexadecimal. En esta implementación, se inicializa con un valor específico "0237000a017573657261", que representa el EAP Response ID (sin incluir la estructura CBOR), puesto que es el primer mensaje hay que enviar al EAP Server ya que es el que contiene la identificación del dispositivo IoT (username).

```
# Inicializo el atributo nas_port_type con "wireless", indicando el tipo de puerto del NAS
self.nas_port_type = "wireless"
# Inicializo el atributo called_station_id como una cadena vacia (almacenar el ID de la estacion que llama)
self.called_station_id = ""
# Inicializo el atributo calling_station_id con una direccion MAC por defecto (alamcena el ID de la estacion llar
self.calling_station_id = "00-00-00-00-00-00"
# Inicializo el atributo vsa_id como una cadena vacia (almacena el identificador de un atributo especifico)
self.vsa_id = ""
# Inicializo el atributo vsa_type como una cadena vacia (define el tipo del atributo VSA)
self.vsa_type = ""
# Inicializo el atributo vsa_value como una cadena vacia (almacena el valor del atributo VSA)
self.vsa_value = ""
# Inicializo el atributo vsa_coding con "string" (define la codificacion del valor VSA)
self.vsa_coding = "string"
# Inicializo el atributo vsa_trim como una cadena vacia (define si se debe recortar el valor VSA)
self.vsa_trim = ""
# Inicializo el atributo framed_ip como una cadena vacia (almacena una direccion IP enmarcada)
self.framed_ip = ""
# Inicializo el atributo radiusid en 0 (identifica la transaccion RADIUS)
self.radiusid = 0
# Inicializo el atributo state como una cadena vacia (almacenar el estado de la conexion)
self.state = ""
# Inicializo el atributo eap con un valor el hexadecimal del mensaje EAP-Response ID (representa un mensaje EAP)
self.eap = "0237000a017573657261"
```

Figura 6.74.- Constructor de la clase RadiusState del script cliente_radius.py parte 2

La clase RadiusState también consta de un método llamado genRADIUSMessageFromState, que tiene como objetivo generar un mensaje RADIUS de autenticación a partir del estado actual de la conexión y de los parámetros previamente configurados en el constructor de la clase. Este mensaje se utiliza para enviar una solicitud de autenticación (Access-Request) al servidor RADIUS.

Para ello, el método genRADIUSMessageFromState recopila los parámetros de conexión (IPs, puertos, identificador de transacción…) y los atributos RADIUS (username, eap, Calling_station_id…) definidos en el constructor de la clase y se los pasa como parámetros a la función Radius_Build_Request, implementada por mí en el script RADIUSMsgGenerator.py. A esta función también se le pasa como parámetro una cadena de 16 bytes (authenticator), formada por números y caracteres, calculada usando la función

Generate_Authenticator del script radiusext.py. Esta cadena será el contenido del campo Authenticator del mensaje RADIUS que se va a construir.

La función Radius_Build_Request construye un paquete RADIUS de tipo Access-Request completo (Header + Payload), incorporando todos los atributos pasados como parámetros y añadiendo otros tres atributos clave. El primero es Framed-MTU, que define el tamaño máximo de los paquetes que pueden ser transmitidos a través de la conexión y se establece en 1400 bytes. El segundo, Connect-Info, proporciona información adicional sobre la conexión del cliente, como detalles de la red, la tecnología o el medio de conexión, y en este caso, se le asigna el valor "CON", indicando una conexión activa entre cliente y servidor sin especificar su tipo. Por último, se incluye el atributo Message-Authenticator, esencial para garantizar la integridad y autenticidad del mensaje, especialmente en solicitudes y respuestas que emplean métodos de autenticación seguros como EAP. Este atributo asegura que los datos no hayan sido alterados durante el tránsito y se calcula utilizando la función CalcRADIUS_MessageAuthenticator, también desarrollada por mí en el script RADIUSMsgGenerator.py.

```
# Esta funcion construye un paquete RADIUS para la autenticacion
2 usages
def Radius_Build_Request(dst_ip, dst_port, src_port, password, username, authenticator, secret, radius_id, nasipaddr,
                         service_type, nas_port_type, calling_station_id, called_station_id, eap, vsa_id, vsa_type,
                         vsa_value, vsa_coding, vsa_trim, framed_ip, state):

    # Creo el atributo de usuario con el nombre de usuario codificado en bytes
    avp1 = RadiusAttr(type="User-Name", value=username.encode())
    # Creo el atributo de direccion IP NAS
    avp3 = RadiusAttr(type="NAS-IP-Address", value=socket.inet_aton(nasipaddr))
    # Combino los atributos en un solo paquete
    avp = bytes(avp1) + bytes(avp3)
    # Si hay una ID de estacion de llamada, la agrego como atributo
    if calling_station_id:
        avp4 = RadiusAttr(type="Calling-Station-Id", value=calling_station_id.encode())
        avp += bytes(avp4)
    # Agrego el atributo de Framed-MTU con un tamaño de 1400
    avp += bytes(RadiusAttr(type="Framed-MTU", value=socket.inet_aton("1400")))
    # Dependiendo del tipo de puerto NAS, agrego el atributo correspondiente
    if nas_port_type == "ethernet":
        avp4 = RadiusAttr(type="NAS-Port-Type", value=socket.inet_aton("15"))  # Ethernet
        avp += bytes(avp4)
    elif nas_port_type == "wireless":
        avp4 = RadiusAttr(type="NAS-Port-Type", value=socket.inet_aton("19"))  # Wireless
        avp += bytes(avp4)
    elif nas_port_type == "virtual":
        avp4 = RadiusAttr(type="NAS-Port-Type", value=socket.inet_aton("5"))  # Virtual
        avp += bytes(avp4)
```

Figura 6.75.- Función Radius_Build_Request del script RADIUSMsgGenerator.py parte 1

```
# Agrego el atributo de informacion de conexion codificado en bytes
avp += bytes(RadiusAttr(type="Connect-Info", value="CON".encode()))
# Agrego el mensaje EAP como atributo en bytes
avp5 = RadiusAttr(type="EAP-Message", value=stringHEXtoHEX(eap))
avp += bytes(avp5)
# Imprimo el estado
print("STATE:", state)
# Si hay un estado, lo agrego como atributo en bytes
if state:
    avp5 = RadiusAttr(type="State", value=stringHEXtoHEX(state))
    avp += bytes(avp5)
# Imprimo el ID del mensaje Radius
print("RADIUS ID:", radius_id)
# Creo el paquete RADIUS con el codigo de solicitud de acceso, el autenticador y el ID
RadiusPacket = RadiusExt(code="Access-Request", authenticator=authenticator, id=radius_id)
# Guardo los atributos en una variable
avp2 = avp
# Calculo el autenticador del mensaje RADIUS
auth = CalcRADIUS_MessageAuthenticator(RadiusPacket, avp2)
# Imprimo el Message-Autehticator calculado
print("Message-Authenticator calculado:", auth)
print("------")
# Agrego el atributo de autenticador de mensaje
avp6 = RadiusAttr(type="Message-Authenticator", value=stringHEXtoHEX(auth))
avp += bytes(avp6)
# Muestro el paquete RADIUS que se va a enviar
print("Enviando Paquete Radius.......")
print((RadiusPacket / avp).show())
print("............................")
# Retorno el paquete RADIUS combinado con los atributos
return RadiusPacket / avp
```

Figura 6.76.- Función Radius_Build_Request del script RADIUSMsgGenerator.py parte 2

Una vez que la función Radius_Build_Request ha construido el paquete RADIUS Access-Request con todos los atributos necesarios (incluido el Message-Authenticator), este paquete es retornado por genRADIUSMessageFromState y estará listo para ser enviado al servidor RADIUS para que lo procese.

```python
# Defino el metodo genRADIUSMessageFromState, que genera un mensaje RADIUS basado en el estado actual
1 usage
def genRADIUSMessageFromState(self):
    # Retorno un mensaje RADIUS creado con los atributos de la clase usando la funcion Radius_Build_Request
    return Radius_Build_Request(
        self.dst_ip,  # Direccion IP del servidor RADIUS
        self.dst_port,  # Puerto de destino para el mensaje
        self.src_port,  # Puerto de origen para el mensaje
        self.password,  # Contrasena del usuario
        self.username,  # Nombre de usuario
        RadiusExt.Generate_Authenticator(),  # Genero un autenticador usando una funcion de RadiusExt
        self.secret,  # Secreto compartido
        self.radiusid,  # Identificador de la transaccion RADIUS
        self.nasipaddr,  # Direccion IP del NAS
        self.service_type,  # Tipo de servicio
        self.nas_port_type,  # Tipo de puerto del NAS
        self.calling_station_id,  # ID de la estacion que llama
        self.called_station_id,  # ID de la estacion llamada
        self.eap,  # Mensaje EAP
        self.vsa_id,  # Identificador de atributo VSA
        self.vsa_type,  # Tipo de atributo VSA
        self.vsa_value,  # Valor del atributo VSA
        self.vsa_coding,  # Codificacion del atributo VSA
        self.vsa_trim,  # Indica si el valor VSA debe ser recortado
        self.framed_ip,  # Direccion IP enmarcada
        self.state  # Estado de la conexion
    )
```

Figura 6.77.- Método genRADIUSMessageFromState de la clase RadiusState del script cliente_radius.py

Tras esto creo la última clase que voy a necesitar, a la que llamo EAP_Authenticator, que tiene como objetivo principal gestionar el proceso de autenticación EAP a través de la creación, del envío y de la recepción de mensajes RADIUS. Proporciona un marco para automatizar el flujo de autenticación EAP, almacenando y administrando el estado de cada paso en la secuencia de autenticación.

Esta clase consta de un método constructor __init__ en el que inicializo los atributos necesarios para el proceso de autenticación EAP:

- Establezco el primer estado en REQ_ID_FIX, que es el estado inicial de autenticación.
- Defino dos listas, receivedPackets y sentPackets, que almacenarán los mensajes RADIUS recibidos y enviados respectivamente durante la autenticación.
- El atributo currentRADIUSPacket se utiliza para almacenar el mensaje RADIUS más reciente generado. Como aún no hay ninguno, lo inicializo como None.

- Creo una instancia de la clase RadiusState, llamada _radiusState.
- Llamo al método genNextRadiusMessageFromState para generar el primer mensaje RADIUS en base a los parámetros de conexión y los atributos del estado inicial.
- Inicializo un contador (contador) en cero, el cual voy a usar para llevar un registro del número de mensajes recibidos del servidor.
- Inicializo otro contador (contador2) en cero, el cual voy a usar para llevar un registro del número de mensajes enviados al servidor.
- Inicializo una variable req_auth como una cadena vacía. La voy a utilizar para almacenar el valor del campo Authenticator del mensaje Access-Request previo al Access-Accept.

```
# Defino la clase EAP_Authenticator, que maneja la autenticacion EAP y la extraccion, generacion y envio de mensajes RADIUS
2 usages
class EAP_Authenticator:
    # Defino el metodo constructor __init__, que inicializa los atributos necesarios para el autenticador EAP
    def __init__(self):
        # Inicializo el estado en REQ_ID_FIX_1, que es el primer estado de autenticacion
        self._state = EAPAuthState.REQ_ID_FIX
        # Inicializo una lista para almacenar los paquetes recibidos
        self.receivedPackets = []
        # Inicializo una lista para almacenar los paquetes enviados
        self.sentPackets = []
        # Inicializo el paquete RADIUS actual como None
        self.currentRADIUSPacket = None
        # Inicializo el estado RADIUS usando la clase RadiusState
        self._radiusState = RadiusState()
        # Genero el siguiente mensaje RADIUS basado en el estado actual
        self.genNextRadiusMessageFromState()
        # Inicializo un contador en 0
        self.contador = 0
        # Inicializo otro contador en 0
        self.contador2 = 0
        # Inicializo una variable req_auth como una cadena vacia
        self.req_auth = ""
```

Figura 6.78.- Constructor de la clase EAP_Authenticator del script cliente_radius.py

El método genNextRadiusMessageFromState de la clase EAP_Authenticator se encarga de crear el siguiente mensaje RADIUS basándose en el estado actual de la autenticación. Para ello, llama al método genRADIUSMessageFromState de la clase RadiusState, que genera un nuevo mensaje RADIUS Access-Request utilizando los valores y configuraciones actuales de sus atributos (como el identificador de usuario, la dirección IP del servidor y otros detalles del mensaje). El mensaje generado se asigna a la variable interna currentRADIUSPacket, de modo que esté listo para ser enviado.

```
# Defino el metodo genNextRadiusMessageFromState, que genera el siguiente mensaje RADIUS
4 usages
def genNextRadiusMessageFromState(self):
    # Genero un nuevo paquete RADIUS usando el estado actual
    self.currentRADIUSPacket = self._radiusState.genRADIUSMessageFromState()
```

Figura 6.79.- Método genNextRadiusMessageFromState de la clase EAP_Authenticator del script cliente_radius.py

El método extractEAPMessage de la clase EAP_Authenticator extrae el mensaje EAP contenido dentro de un mensaje RADIUS (es decir, el mensaje contenido dentro del atributo EAP-Message), localizando su posición dentro del mensaje, calculando su longitud y devolviendo el segmento específico correspondiente al mensaje EAP.

El proceso seguido para conseguirlo es el siguiente: Tomo los bytes 4 a 8 del mensaje EAP (raw_eap_msg[4:8]), donde se encuentra la longitud del mensaje EAP en formato hexadecimal y convierto este valor hexadecimal a un entero (EAP_len), que representa la longitud en bytes del mensaje EAP. Tras esto, encuentro la posición inicial del mensaje EAP dentro del mensaje RADIUS (radius_msg) utilizando find, que localiza la primera aparición de raw_eap_msg (mensaje EAP) en radius_msg (mensaje RADIUS) y guardo esta posición en la variable init_possition. Después, calculo la posición final del mensaje EAP en el mensaje RADIUS (radius_msg), sumando a la posición inicial (init_possition) el doble de la longitud del mensaje (EAP_len * 2) y guardo esta posición en la variable final_possition. Finalmente, extraigo el segmento correspondiente al mensaje EAP que comprende desde la posición init_possition hasta la posición final_possition del mensaje RADIUS (radius_msg), lo almaceno en la variable EAP_msg y lo retorno.

```python
# Defino el metodo extractEAPMessage, que extrae el mensaje EAP de un mensaje RADIUS
3 usages
def extractEAPMessage(self, radius_msg, raw_eap_msg):
    # Extraigo la longitud del mensaje EAP del mensaje en crudo, tomando los bytes 4 a 8
    EAP_len_str = raw_eap_msg[4:8]
    # Convierto la longitud EAP de hexadecimal a entero
    EAP_len = int(EAP_len_str, 16)
    # Encuentro la posicion inicial del mensaje EAP en el mensaje RADIUS
    init_possition = radius_msg.find(raw_eap_msg)
    # Calculo la posicion final del mensaje EAP en el mensaje RADIUS
    final_possition = init_possition + (EAP_len * 2)
    # Extraigo el mensaje EAP del mensaje RADIUS
    EAP_msg = radius_msg[init_possition:final_possition]
    # Retorno el mensaje EAP extraido
    return EAP_msg
```

Figura 6.80.- Método extractEAPMessage de la clase EAP_Authenticator del script cliente_radius.py

El método set_state de la clase EAP_Authenticator permite fijar o actualizar el estado actual de la autenticación EAP dentro de la clase. Esta clase recibe un parámetro new_state, que representa el nuevo estado que se desea establecer y asigna este nuevo estado a la variable interna _state.

```python
# Defino el metodo set_state, que permite fijar el estado de autenticacion EAP
2 usages
def set_state(self, new_state):
    """Fijar el estado de autenticacion EAP."""
    self._state = new_state
```

Figura 6.81.- Método set_state de la clase EAP_Authenticator del script cliente_radius.py

El método set_eap_message de la clase EAP_Authenticator permite establecer el mensaje EAP que se incluirá en el próximo mensaje RADIUS que se envíe. Este método recibe un parámetro eap_message, que representa el nuevo mensaje EAP que se quiere fijar y actualiza el atributo eap del objeto _radiusState con ese nuevo mensaje EAP, lo cual asegura que el mensaje EAP configurado esté listo para ser añadido en la próxima petición RADIUS generada por el método genRADIUSMessageFromState.

```
# Defino el metodo set_eap_message, que permite fijar el mensaje EAP que se va a enviar
2 usages
def set_eap_message(self, eap_message):
    """Fijar el mensaje EAP que se va a enviar."""
    self._radiusState.eap = eap_message
```

Figura 6.82.- Método set_eap_message de la clase EAP_Authenticator del script cliente_radius.py

Por último, la clase EAP_Authenticator también consta de un método sendNextMessageToRADIUS que tiene como objetivo enviar el mensaje RADIUS Access-Request construido con los datos del estado actual de la autenticación al servidor RADIUS y recibir y procesar la respuesta a ese mensaje.

Lo primero que hago dentro de este método es definir como variables globales el mensaje de respuesta RADIUS en bytes y el estado (atributo State) de la segunda y de la tercera respuesta RADIUS enviada por el servidor (datos_radiusResponsePacket_bytes, radiusResponsePacket_state_2 y radiusResponsePacket_state_3 respectivamente). Estas variables las inicialicé a nivel de módulo como cadenas vacías tras los imports y antes de la clase EAPAuthState.

```
# Importo el modulo Enum para definir enumeraciones
from enum import Enum

# Inicializo tres variables globales como cadenas vacías. Al estar definidas fuera de cualquier clase o funcion, estas
# variables pueden ser accedidas y modificadas desde cualquier parte del modulo en el que estan definidas. Esto sera
# util mas adelante, ya que en los bucles if del metodo sendNextMessageToRADIUS de la clase EAP_Authenticator se
# utilizaran estas variables para identificar el estado radius o para obtener las claves MS-MPPE
# para la construccion de las uris de los mensajes POST
datos_radiusResponsePacket_bytes = ""
radiusResponsePacket_state_2 = ""
radiusResponsePacket_state_3 = ""


# Establezco el nivel de logging para los mensajes de runtime de Scapy en ERROR, lo que significa que solo se
# registraran errores y no mensajes de depuracion o advertencias
logging.getLogger("scapy.runtime").setLevel(logging.ERROR)


# Defino una clase de enumeracion (Enum) para representar los diferentes estados de autenticacion EAP
9 usages
class EAPAuthState(Enum):
```

Figura 6.83.- Definición de las variables globales a nivel de módulo en el script cliente_radius.py

Al estar definidas fuera de cualquier clase o función, estas variables pueden ser accedidas y modificadas desde cualquier parte del módulo en el que están definidas. En este caso actualizaremos su valor dentro de este método sendNextMessageToRADIUS, es decir, dejarán de ser cadenas vacías.

Tras esto, creo un socket UDP al que llamo clientSocket, que será el canal de comunicación con el servidor RADIUS, asegurándome de configurarlo con un tiempo de espera adecuado (por ejemplo de 1 segundo) para no bloquear el flujo de ejecución si la respuesta tarda demasiado. A continuación, enlazo el socket a todas las interfaces locales en el puerto de origen especificado por self._radiusState.src_port, lo cual permite que el cliente reciba la respuesta en el mismo puerto desde el que envió el mensaje. También configuro la dirección del servidor RADIUS, especificando su IP (192.168.217.128) y el puerto en el que está escuchando (self._radiusState.dst_port).

Para enviar el mensaje, convierto el paquete RADIUS Access-Request con los datos del estado actual (self.currentRADIUSPacket) a un formato base64, luego lo decodifico a bytes haciéndolo apto para la transmisión y lo guardo en la variable message. Antes de transmitirlo, si ya he enviado tres peticiones Access-Request, extraigo el valor del campo Authenticator correspondiente a la cuarta petición. Este valor, en formato hexadecimal, lo asigno a la variable interna req_auth. La importancia de esta variable radica en que será utilizada por el EAP Authenticator para descifrar las claves MS y calcular la MSK necesaria para establecer su contexto de seguridad OSCORE. Después de esto, imprimo el valor del Request Authenticator para verificar que ha sido correctamente extraído, envío el mensaje (message) al servidor RADIUS utilizando el método sendto del socket, y actualizo el contador que registra el número de peticiones enviadas, es decir, lo incremento en uno.

```python
# Defino el metodo sendNextMessageToRADIUS, que envia el mensaje RADIUS y recibe la respuesta
4 usages
def sendNextMessageToRADIUS(self):
    # Declaro que las variables datos_radiusResponsePacket_bin, radiusResponsePacket_state_2 y
    # radiusResponsePacket_state_3 son globales,lo que permite acceder a ellas y modificarlas dentro de esta funcion
    global datos_radiusResponsePacket_bytes, radiusResponsePacket_state_2, radiusResponsePacket_state_3
    # Creo un socket UDP para la comunicacion con el servidor RADIUS
    clientSocket = socket(AF_INET, SOCK_DGRAM)
    # Establezco un tiempo de espera de 1 segundo para el socket
    clientSocket.settimeout(1)
    # Enlazo el socket a todas las interfaces en el puerto de origen
    clientSocket.bind(('0.0.0.0', self._radiusState.src_port))
    # Defino la direccion del servidor RADIUS al que voy a enviar el mensaje
    addr = ("192.168.124.128", self._radiusState.dst_port)
    # Inicio el cronometro para medir el tiempo de envio
    start = time.time()
    # Conviero el paquete RADIUS a un mensaje en formato base64
    b64 = codecs.encode(self.currentRADIUSPacket.build(), encoding: 'base64').decode().strip()
    # Decodifico el mensaje base64 a bytes
    message = base64.b64decode(b64)
    # Si ya he enviado tres peticiones Access-Request
    if self.contador2 == 3:
        # Obtengo el campo Authenticator de la cuarta peticion Access-Request y lo asigno a la varibale interna
        # req_auth. Esta variable la empleara el EAP Authenticator para descifrar las claves MS y poder obtener la MSK
        self.req_auth = message.hex()[8:40]
        # Imprimo su valor para comprobar que es correcto
        print("Request Authenticator previo al Access_Accept:", self.req_auth)
    # Envio el mensaje a la direccion del servidor RADIUS
    clientSocket.sendto(message, addr)
    # Incremento el contador2 en 1 para llevar un registro de cuantas peticiones he enviado
    self.contador2 += 1
```

Figura 6.84.- Método sendNextMessageToRADIUS de la clase EAP_Authenticator del script cliente_radius.py parte 1

A continuación, preparo la recepción de la respuesta inicializando en None varias variables que almacenarán la información relevante del paquete de respuesta: el paquete completo (radiusResponsePacket), el atributo State del paquete (response_state), el atributo EAP-Message (response_eap_msg) y el mensaje EAP sin procesar (raw_eap_msg).

Continuo el proceso intentando recibir una respuesta del servidor RADIUS a través del socket previamente configurado. Al recibir los datos de la respuesta (data), procedo a imprimirlos empleando la función hexdump de la biblioteca Scapy, la cual muestra los datos en formato hexadecimal en la parte izquierda y el equivalente en ASCII en la parte derecha, facilitando la lectura de los datos en bruto y el reconocimiento de los caracteres legibles. Luego, creo un objeto RADIUS de la biblioteca Scapy (es una poderosa biblioteca de Python que permite la manipulación y el análisis de paquetes de red de bajo nivel) a partir de estos

datos, al que llamo radiusResponsePacket, permitiendo procesar el mensaje como un paquete RADIUS estructurado y visualizarlo en un formato más legible mediante el método show.

Además, convierto radiusResponsePacket en bytes a través del método build, lo almaceno en la variable datos_radiusResponsePacket_bytes, lo transformo en formato hexadecimal, lo cual me facilita el análisis y la verificación de datos específicos y lo muestro por consola. Finalmente, obtengo la lista de los atributos del paquete RADIUS.

```
# Inicializo la variable radiusResponsePacket como None para almacenar la respuesta del servidor RADIUS
radiusResponsePacket = None
# Inicializo la variable response_state como None para almacenar el estado de la respuesta
response_state = None
# Inicializo la variable response_eap_msg como None para almacenar el mensaje EAP de la respuesta
response_eap_msg = None
# Inicializo la variable raw_eap_msg como None para almacenar el mensaje EAP en su formato crudo
raw_eap_msg = None
# Intento recibir un paquete de respuesta del servidor RADIUS
try:
    # Recibo los datos y la direccion del servidor usando el socket
    data, server = clientSocket.recvfrom(self._radiusState.src_port)
    # Indico que estoy procesando el mensaje recibido
    print("Procesando el mensaje recibido")
    # Muestro el contenido del paquete recibido en formato hexadecimal
    print(hexdump(data))
    # Creo un objeto Radius a partir de los bytes recibidos
    radiusResponsePacket = Radius(bytes(data))
    # Muestro el contenido del paquete RADIUS en formato legible
    print(radiusResponsePacket.show())
    # Construyo el paquete RADIUS de nuevo en formato bytes
    datos_radiusResponsePacket_bytes = radiusResponsePacket.build()
    # Convierto el mensaje RADIUS recibido a hexadecimal y lo decodifico a cadena
    radiusResponsePacketHex = binascii.hexlify(datos_radiusResponsePacket_bytes).decode('utf-8')
    # Muestro el mensaje RADIUS recibido en hexadecimal
    print("MENSAJE RADIUS RECIBIDO HEXADECIMAL:", radiusResponsePacketHex)
    print("Ahora los atributos del paquete recibido:")
    # Obtengo la lista de atributos del paquete RADIUS
    attlist = radiusResponsePacket.attributes
```

Figura 6.85.- Método sendNextMessageToRADIUS de la clase EAP_Authenticator del script cliente_radius.py parte 2

Tras obtener la lista con los atributos del paquete RADIUS recibido, la recorro y para cada atributo, comienzo mostrando su contenido de manera legible con el método show, seguido de un separador visual para facilitar la lectura. Posteriormente, visualizo el tipo de cada atributo, lo que me ayuda a identificar el propósito de cada uno dentro del paquete.

Si el tipo del atributo es 24, lo identifico como el estado (atributo State) del mensaje de respuesta, y almaceno este valor en la variable response_state después de convertirlo a formato hexadecimal para una manipulación más sencilla en pasos posteriores. Por otro lado, si el tipo es 79, esto indica que contiene un mensaje EAP (atributo EAP-Message), así que guardo el valor del atributo en la variable response_eap_msg y también conservo el atributo EAP-Message completo tal como se recibe del servidor RADIUS en la variable raw_eap_msg.

En caso de que el servidor no responda dentro del tiempo configurado, capturo una excepción e informo de que se ha agotado el tiempo de espera, lo que indica un fallo en la recepción de la respuesta del servidor RADIUS.

```
    # Recorro cada atributo de la lista
    for att in attlist:
        # Muestro cada atributo en formato legible
        print(att.show())
        # Separador
        print("--")
        # Muestro el tipo del atributo
        print(att.type)
        # Si el tipo del atributo es 24, indica el estado
        if att.type == 24:
            # Almaceno el valor del atributo convertido a hexadecimal y decodificado a
            # cadena como el estado de respuesta
            response_state = hexlify(att.value).decode('utf-8')
            # Muestro el valor del estado del primer mensaje de respuesta
            print("Este es el estado (state): ", response_state)
        # Si el tipo del atributo es 79, indica un mensaje EAP
        if att.type == 79:
            # Almaceno el mensaje EAP
            response_eap_msg = att.value
            # Almaceno el atributo crudo
            raw_eap_msg = att
# Capturo la excepcion de tiempo de espera si no se recibe respuesta
except timeout:
    # Indico que se ha agotado el tiempo de espera para la solicitud
    print("SE AGOTÓ EL TIEMPO DE ESPERA PARA LA SOLICITUD")
```

Figura 6.86.- Método sendNextMessageToRADIUS de la clase EAP_Authenticator del script cliente_radius.py parte 3

Una vez hecho todo lo anterior, incremento la variable interna contador en uno. Esto lo hago para llevar un registro del número de respuestas que he procesado. Dependiendo de cuántas haya procesado, la forma en que guardo el estado (valor del atributo State) de la respuesta cambia:

- Si es la primera respuesta, guardo directamente en el atributo State de la variable interna _radiusState (self._radiusState.state) el valor previamente almacenado en la variable response_state.

- Si es la segunda respuesta, a partir del paquete RADIUS recibido en formato bytes (datos_radiusResponsePacket_bytes), extraigo el fragmento que contiene la información de estado (del byte 53 al byte 69). Lo convierto a formato hexadecimal y lo almaceno en el atributo State de la variable interna _radiusState (self._radiusState.state). Esto lo hago así porque no soy capaz, por algún motivo de la lógica interna de la implementación, de obtener directamente el valor del atributo State de esta segunda respuesta, como sí podía hacerlo con la primera.

- De manera similar, si es la tercera respuesta, a partir del paquete RADIUS recibido en formato bytes (datos_radiusResponsePacket_bytes), extraigo el fragmento que contiene la información de estado (del byte 83 al byte 99). Lo convierto a formato hexadecimal y lo almaceno en el atributo State de la variable interna _radiusState (self._radiusState.state). Esto lo hago así porque no soy capaz, por algún motivo de la lógica interna de la implementación, de obtener directamente el valor del atributo State de esta tercera respuesta, como sí podía hacerlo con la primera.

En cada paso, también muestro por consola el estado que he almacenado para asegurarme de que estoy registrando correctamente los valores.

```
# Incremento el contador en 1 para llevar un registro de cuantas respuestas he procesado
self.contador += 1
# Si es la primera respuesta, almaceno el estado recuperado directamente
if self.contador == 1:
    # Almaceno el estado recuperado
    self._radiusState.state = response_state
    # Muestro el estado recuperado
    print("El estado (state) recuperado es:", response_state)
# Si es la segunda respuesta, almaceno el estado desde el paquete RADIUS
if self.contador == 2:
    # Extraigo el estado de la segunda respuesta RADIUS desde el paquete en bytes, tomando los bytes
    # desde la posición 53 hasta la 69
    radiusResponsePacket_state_2 = datos_radiusResponsePacket_bytes[53:69]
    # Indico que ahora voy a mostrar los atributos del paquete RADIUS recibido
    # Almaceno el estado de la variable radiusResponsePacket_state_2 en hexadecimal
    self._radiusState.state = hexlify(radiusResponsePacket_state_2).decode('utf-8')
    # Muestro el estado en hexadecimal
    print("Este es el estado (state):", hexlify(radiusResponsePacket_state_2).decode('utf-8'))
    # Muestro el estado recuperado
    print("El estado (state) recuperado es:", hexlify(radiusResponsePacket_state_2).decode('utf-8'))
# Si es la tercera respuesta, almaceno el estado desde el paquete RADIUS
if self.contador == 3:
    # Extraigo el estado de la tercera respuesta RADIUS desde el paquete en bytes, tomando los bytes
    # desde la posición 83 hasta la 99
    radiusResponsePacket_state_3 = datos_radiusResponsePacket_bytes[83:99]
    # Almaceno el estado de la variable radiusResponsePacket_state_3 en hexadecimal
    self._radiusState.state = hexlify(radiusResponsePacket_state_3).decode('utf-8')
    # Muestro el estado en hexadecimal
    print("Este es el estado (state):", hexlify(radiusResponsePacket_state_3).decode('utf-8'))
    # Muestro el estado recuperado
    print("El estado (state) recuperado es:", hexlify(radiusResponsePacket_state_3).decode('utf-8'))
```

Figura 6.87.- Método sendNextMessageToRADIUS de la clase EAP_Authenticator del script cliente_radius.py parte 4

Lo siguiente que hago es incrementar el identificador de mensaje RADIUS o identificador de transacción (radiusid) para actualizar su valor en la próxima solicitud Access-Request que se vaya a enviar al servidor. Luego, creo un objeto EAP de la biblioteca Scapy, llamado radiusEAPPacket, utilizando el contenido del atributo EAP-Message del mensaje de respuesta recibido, guardado previamente en la variable response_eap_msg. Después, muestro el contenido de este paquete EAP de forma legible mediante el método show.

A continuación, verifico si el estado actual de la autenticación es REQ_ID_FIX, lo que indica que debo realizar unas acciones específicas. Si este es el caso, convierto el paquete

EAP (radiusEAPPacket) a formato hexadecimal, lo guardo en la variable paq_hex y lo imprimo para ver su contenido. Luego, realizo los cambios necesarios a este mensaje EAP hexadecimal (paq_hex) para construir el atributo EAP-Message del nuevo mensaje Access-Request que se enviará al servidor. Para ello, cambio el código (en vez de ser 1, será 2 ya que es una respuesta), cambio también la longitud (será 10 en vez de 5 puesto que es la longitud del EAP Response ID) y le añado al final la identidad del dispositivo IoT (usera, que es la cadena hexadecimal "7573657261"). El paquete EAP hexadecimal con todos estos cambios lo guardo en la variable paq_hex_mod y lo asigno al atributo EA-Message de la variable interna _radiusState (self._radiusState.eap).

Finalmente, actualizo el estado de autenticación a EAP_PSK_1 (self._state = EAPAuthState.EAP_PSK_1), lo que indica que el proceso de autenticación ha avanzado a una nueva fase. Con todas las acciones realizadas en este estado soluciono el problema del bug.

```
# Incremento el identificador RADIUS para la siguiente solicitud
self._radiusState.radiusid += 1
# Creo un paquete EAP utilizando el mensaje EAP de respuesta
radiusEAPPacket = EAP(response_eap_msg)
# Muestro el contenido del paquete EAP
radiusEAPPacket.show()
# Verifico si el estado actual es REQ_ID_FIX
if self._state == EAPAuthState.REQ_ID_FIX:
    # Convierto el paquete EAP recibido en hexadecimal
    paq_hex = str(radiusEAPPacket.build().hex())
    # Imprimo el paquete en formato hexadecimal
    print(paq_hex)
    # Defino el codigo (2 ya que es una respuesta) y la longitud (se que la longitud del EAP Response ID es 10)
    # que deben ser reemplazados ya que tengo que "replicar" el primer mensaje RADIUS que contenia el
    # EAP-Response ID pero con el ID del mensaje (Access-Challenge con codigo 1 y longitud 5) recibido
    cod = "02"  # Reemplazo la parte correspondiente a "01"
    lon = "000a"  # Reemplazo la parte correspondiente a "0005"
    # Realizo las modificaciones al paquete hexadecimal
    # El paquete hexadecimal a enviar estara formado por el codigo (2), el ID de la respuesta, la longitud (10),
    # el type (Identity (1)) y la cadena hexadecimal "7573657261" ("usera") que es el ID del par EAP
    paq_hex_mod = cod + paq_hex[2:4] + lon + paq_hex[8:] + "7573657261"
    # Muestro el paquete modificado en formato hexadecimal
    print(paq_hex_mod)
    # Almaceno el paquete EAP modificado en el atributo EAP de la clase RadiusState
    self._radiusState.eap = paq_hex_mod
    # Cambio el estado de autenticacion a EAP_PSK_1
    self._state = EAPAuthState.EAP_PSK_1
```

Figura 6.88.- Método sendNextMessageToRADIUS de la clase EAP_Authenticator del script cliente_radius.py parte 5

Si el estado actual de la autenticación es EAP_PSK_1, debo realizar otras acciones específicas diferentes. Primero, imprimo el mensaje EAP que he recibido (raw_eap_msg).

Si el mensaje es de tipo bytes, lo muestro directamente en hexadecimal; si no, lo convierto a bytes antes de imprimirlo en hexadecimal.

Luego, construyo el paquete de respuesta EAP en bytes con el método build a partir del objeto radiusEAPPacket, lo codifico a hexadecimal para verlo claramente y lo guardo en la variable radius_eap_packet_hex. Lo mismo hago con el paquete RADIUS de respuesta (radiusResponsePacket), construyéndolo en bytes, codificándolo a hexadecimal y luego guardándolo en la variable radius_response_packet_hex.

A continuación, llamo al método extractEAPMessage de la clase EAP_Authenticator, que toma el contenido hexadecimal de ambos paquetes (EAP y RADIUS) para extraer el primer mensaje EAP-PSK. Este mensaje lo guardo en la variable EAP_PSK_MSG_1 y será el valor del atributo EAP-Message del segundo mensaje de respuesta recibido, que en este caso es un Access-Challenge. Finalmente, imprimo el mensaje extraído para asegurarme de que es correcto y lo retorno, avanzando así en el proceso de autenticación.

```
# Verifico si el estado actual es EAP_PSK_1
if self._state == EAPAuthState.EAP_PSK_1:
    # Imprimo el mensaje EAP recibido indicando que se esta mostrando el mensaje EAP en crudo
    print("Mensaje EAP en crudo (RAW)")
    # Me aseguro de que raw_eap_msg sea un objeto que pueda ser convertido a bytes
    if isinstance(raw_eap_msg, bytes):
        # Si raw_eap_msg es de tipo bytes, lo imprimo en hexadecimal
        print(raw_eap_msg.hex())
    else:
        # Si raw_eap_msg no es de tipo bytes, lo convierto a bytes y luego imprimo en hexadecimal
        print(str(raw_eap_msg).encode('utf-8').hex())
    # Imprimo una línea vacia para separar el contenido
    print("")
    # Construyo el paquete EAP de respuesta en bytes utilizando el metodo build() del objeto radiusEAPPacket y
    # lo codifico a hexadecimal
    radius_eap_packet_hex = radiusEAPPacket.build().hex()
    # Imprimo un mensaje indicando que se esta mostrando el contenido del metodo EAP (el paquete EAP en formato
    # hexadecimal)
    print("Contenido del metodo EAP:", radius_eap_packet_hex)
    # Construyo el paquete RADIUS de respuesta en bytes utilizando el metodo build() del objeto
    # radiusResponsePacket y lo codifico a hexadecimal
    radius_response_packet_bytes = radiusResponsePacket.build().hex()
    # Extraigo el mensaje EAP-PSK-1 utilizando el metodo extractEAPMessage
    EAP_PSK_MSG_1 = self.extractEAPMessage(radius_response_packet_bytes, radius_eap_packet_hex)
    # Imprimo el mensaje EAP-PSK-1 extraido
    print("Mensaje EAP-PSK-1:", EAP_PSK_MSG_1)
    # Retorno el mensaje EAP-PSK-1
    return EAP_PSK_MSG_1
```

Figura 6.89.- Método sendNextMessageToRADIUS de la clase EAP_Authenticator del script cliente_radius.py parte 6

En caso de que el estado actual de la autenticación sea EAP_PSK_3, debo realizar unas acciones similares a las del estado EAP_PSK_1. Primero, imprimo el mensaje EAP que he recibido (raw_eap_msg). Esto me permite verificar el mensaje original tal como llega. Si el mensaje es de tipo bytes, lo muestro directamente en hexadecimal; si no, lo convierto a bytes antes de imprimirlo en hexadecimal, asegurándome de ver la estructura en el formato correcto.

Luego, construyo el paquete de respuesta EAP en bytes con el método build a partir del objeto radiusEAPPacket, lo codifico a hexadecimal para verlo claramente y lo guardo en la variable radius_eap_packet_hex. Lo mismo hago con el paquete RADIUS de respuesta (radiusResponsePacket), construyéndolo en bytes, codificándolo a hexadecimal y luego guardándolo en la variable radius_response_packet_hex.

A continuación, llamo al método extractEAPMessage de la clase EAP_Authenticator, que toma el contenido hexadecimal de ambos paquetes (EAP y RADIUS) para extraer el tercer mensaje EAP-PSK. Este mensaje lo guardo en la variable EAP_PSK_MSG_3 y será el valor del atributo EAP-Message del tercer mensaje de respuesta recibido, que en este caso es un Access-Challenge. Finalmente, imprimo el mensaje extraído para asegurarme de que es correcto y lo retorno, avanzando así en el proceso de autenticación.

```python
# Verifico si el estado actual es EAP_PSK_3
if self._state == EAPAuthState.EAP_PSK_3:
    # Imprimo el mensaje EAP recibido indicando que se esta mostrando el mensaje EAP en crudo
    print("Mensaje EAP en crudo (RAW)")
    # Me aseguro de que raw_eap_msg sea un objeto que pueda ser convertido a bytes
    if isinstance(raw_eap_msg, bytes):
        # Si raw_eap_msg es de tipo bytes, lo imprimo en hexadecimal
        print(raw_eap_msg.hex())
    else:
        # Si raw_eap_msg no es de tipo bytes, lo convierto a bytes y luego imprimo en hexadecimal
        print(str(raw_eap_msg).encode('utf-8').hex())
    # Imprimo una línea vacia para separar el contenido
    print("")
    # Construyo el paquete EAP de respuesta en bytes utilizando el metodo build() del objeto radiusEAPPacket y
    # lo codifico a hexadecimal
    radius_eap_packet_hex = radiusEAPPacket.build().hex()
    # Imprimo un mensaje indicando que se esta mostrando el contenido del metodo EAP (el paquete EAP en formato
    # hexadecimal)
    print("Contenido del metodo EAP:", radius_eap_packet_hex)
    # Construyo el paquete RADIUS de respuesta en bytes utilizando el método build() del objeto
    # radiusResponsePacket y lo codifico a hexadecimal
    radius_response_packet_hex = radiusResponsePacket.build().hex()
    # Extraigo el mensaje EAP-PSK-3 utilizando el metodo extractEAPMessage
    EAP_PSK_MSG_3 = self.extractEAPMessage(radius_response_packet_hex, radius_eap_packet_hex)
    # Imprimo el mensaje EAP-PSK-3 extraido
    print("Mensaje EAP-PSK-3:", EAP_PSK_MSG_3)
    # Retorno el mensaje EAP-PSK-3
    return EAP_PSK_MSG_3
```

Figura 6.90.- Método sendNextMessageToRADIUS de la clase EAP_Authenticator del script cliente_radius.py parte 7

Por último, en caso de que el estado actual de la autenticación sea EAP_SUCCESS, debo realizar unas acciones similares a las del estado EAP_PSK_1 y EAP-PSK-3. Primero, imprimo el mensaje EAP que he recibido (raw_eap_msg). Esto me permite verificar el mensaje original tal como llega. Si el mensaje es de tipo bytes, lo muestro directamente en hexadecimal; si no, lo convierto a bytes antes de imprimirlo en ese formato, asegurándome de ver la estructura en el formato correcto.

Luego, construyo el paquete de respuesta EAP en bytes con el método build a partir del objeto radiusEAPPacket, lo codifico a hexadecimal para verlo claramente y lo guardo en la variable radius_eap_packet_hex. Lo mismo hago con el paquete RADIUS de respuesta (radiusResponsePacket), construyéndolo en bytes, codificándolo a hexadecimal y luego guardándolo en la variable radius_response_packet_hex.

A continuación, llamo al método extractEAPMessage de la clase EAP_Authenticator, que toma el contenido hexadecimal de ambos paquetes (EAP y RADIUS) para extraer el mensaje de éxito. Este mensaje lo guardo en la variable EAP_PSK_MSG_SUCCESS y será el valor del atributo EAP-Message del cuarto mensaje de respuesta recibido, que en este caso es un Access-Accept. Tras esto, imprimo el mensaje extraído para asegurarme de que es correcto.

```
# Verifico si el estado actual es EAP_SUCCESS
if self._state == EAPAuthState.EAP_SUCCESS:
    # Imprimo el mensaje EAP recibido indicando que se esta mostrando el mensaje EAP en crudo
    print("Mensaje EAP en crudo (RAW)")
    # Me aseguro de que raw_eap_msg sea un objeto que pueda ser convertido a bytes
    if isinstance(raw_eap_msg, bytes):
        # Si raw_eap_msg es de tipo bytes, lo imprimo en hexadecimal
        print(raw_eap_msg.hex())
    else:
        # Si raw_eap_msg no es de tipo bytes, lo convierto a bytes y luego imprimo en hexadecimal
        print(str(raw_eap_msg).encode('utf-8').hex())
    # Imprimo una línea vacia para separar el contenido
    print("")
    # Construyo el paquete EAP de respuesta en bytes utilizando el metodo build() del objeto radiusEAPPacket y
    # lo codifico a hexadecimal
    radius_eap_packet_hex = radiusEAPPacket.build().hex()
    # Imprimo un mensaje indicando que se esta mostrando el contenido del metodo EAP (el paquete EAP en formato
    # hexadecimal)
    print("Contenido del metodo EAP:", radius_eap_packet_hex)
    # Construyo el paquete RADIUS de respuesta en bytes utilizando el método build() del objeto
    # radiusResponsePacket y lo codifico a hexadecimal
    radius_response_packet_hex = radiusResponsePacket.build().hex()
    # Extraigo el mensaje EAP-SUCCESS utilizando el metodo extractEAPMessage
    EAP_PSK_MSG_SUCCESS = self.extractEAPMessage(radius_response_packet_hex, radius_eap_packet_hex)
    # Imprimo el mensaje EAP-SUCCESS extraido
    print("Mensaje EAP-SUCCESS:", EAP_PSK_MSG_SUCCESS)
```

Figura 6.91.- Método sendNextMessageToRADIUS de la clase EAP_Authenticator del script cliente_radius.py parte 8

Una vez hecho lo anterior, a partir del paquete RADIUS recibido en formato bytes (datos_radiusResponsePacket_bytes), extraigo el fragmento (del byte 28 al byte 78) que contiene la clave MS-MPPE-Recv-Key. Esta clave está contenida en un atributo específico de Microsoft y se identifica con el Type 17 dentro del atributo Vendor-Specific con Vendor ID de Microsoft. Convierto esta clave a formato hexadecimal, la imprimo para confirmar visualmente su contenido cifrado y la guardo en la variable MPPE_RecKey_hex.

Después, extraigo el fragmento (del byte 86 al byte 136) que contiene la clave MS-MPPE-Send-Key, que está contenida en un atributo específico de Microsoft y se identifica con el Type 16 dentro del atributo Vendor-Specific con Vendor ID de Microsoft. Convierto esta clave a formato hexadecimal, la imprimo para confirmar visualmente su contenido cifrado y la guardo en la variable MPPE_SendKey_hex.

Finalmente, retorno el mensaje EAP SUCCESS junto con ambas claves en formato hexadecimal.

```
# Extraigo la clave MS-MPPE-Recv-Key que comprende de los bytes 28 al 78 del paquete RADIUS
radiusResponsePacket_MPPE_RecKey = datos_radiusResponsePacket_bytes[28:78]
# Convierto la clave MS-MPPE-Recv-Key a formato hexadecimal
MPPE_RecKey_hex = binascii.hexlify(radiusResponsePacket_MPPE_RecKey).decode('utf-8')
# Imprimo la clave MS-MPPE-Recv-Key en formato hexadecimal
print("MPPE_RecKey cifrada:", MPPE_RecKey_hex)
# Extraigo la clave MS-MPPE-Send-Key que comprende de los bytes 86 al 136 del paquete RADIUS
radiusResponsePacket_MPPE_SendKey = datos_radiusResponsePacket_bytes[86:136]
# Convierto la clave MS-MPPE-Send-Key a formato hexadecimal
MPPE_SendKey_hex = binascii.hexlify(radiusResponsePacket_MPPE_SendKey).decode('utf-8')
# Imprimo la clave MS-MPPE-Send-Key en formato hexadecimal
print("MPPE_SendKey cifrada:", MPPE_SendKey_hex)
# Retorno el mensaje de exito EAP-PSK y las claves MS-MPPE-Recv-Key y MS-MPPE-Send-Key
# en formato hexadecimal
return EAP_PSK_MSG_SUCCESS, MPPE_RecKey_hex, MPPE_SendKey_hex
```

Figura 6.92.- Método sendNextMessageToRADIUS de la clase EAP_Authenticator del script cliente_radius.py parte 9

6.3.4.- EAP Server

El EAP Server ya me vino dado en una máquina virtual a la que llamé “TFG Daniel”. Esta máquina virtual ya tiene implementado en C un servidor RADIUS (FreeRadius), con soporte para el método EAP-PSK, que permite la comunicación con el cliente RADIUS que está dentro del EAP Authenticator. En primer lugar el EAP Server se encargará, a partir del servidor RADIUS, de construir el primer mensaje EAP-PSK y enviarlo al EAP Authenticator. Más tarde, recibirá el segundo mensaje EAP-PSK, construido por el EAP Peer y enviado por el EAP Authenticator a través del cliente RADIUS, lo procesará, realizará las comprobaciones oportunas (cabecera correcta, Flag correcto...), construirá el tercer

mensaje EAP-PSK y lo enviará al EAP Authenticator. Por último, recibirá el cuarto mensaje EAP-PSK construido por el EAP Peer y enviado por el EAP Authenticator a través del cliente RADIUS, lo procesará, realizará las comprobaciones oportunas (cabecera correcta, Flag correcto, canal protegido bien calculado, MAC_S bien calculada...) y si todo es correcto, construirá y enviará el mensaje EAP SUCCESS al EAP Authenticator. Además, esta máquina virtual permite ejecutar un ejemplo que usa PANATIKI, introduciendo una serie de comandos en el terminal, que me sirvió para entender mejor el protocolo RADIUS, el método EAP-PSK y ver qué atributos tenían que llevar cada uno de los mensajes RADIUS utilizados para el intercambio de mensajes EAP-PSK.

Otra cosa que me resulto de utilidad fue ver el código del script eap_psk.c del servidor RADIUS implementado en C, ya que pude añadir prints para saber, entre otras cosas, qué parámetros había que utilizar para construir el canal protegido. También me sirvió como referencia para implementar las funciones del script mensajes_EAP_PSK.py y el código del EAP Peer donde construyo el segundo y el cuarto mensaje EAP-PSK.

```
wpa_hexdump(MSG_INFO, "EAP-PSK: RECEIVED MAC_S", psk->mac_s, EAP_PSK_MAC_LEN);

if (eap_psk_derive_keys(data->kdk, data->rand_p, data->tek, data->msk,
            data->emsk))
    goto fail;
wpa_hexdump_key(MSG_INFO, "EAP-PSK: TEK", data->tek, EAP_PSK_TEK_LEN);
wpa_printf(MSG_INFO, "EAP-PSK: TEK");
printf_hex(data->tek, EAP_PSK_TEK_LEN);
wpa_hexdump_key(MSG_INFO, "EAP-PSK: MSK", data->msk, EAP_MSK_LEN);
printf_hex(data->msk, EAP_MSK_LEN);
wpa_hexdump_key(MSG_INFO, "EAP-PSK: EMSK", data->emsk, EAP_EMSK_LEN);
printf_hex(data->emsk, EAP_EMSK_LEN);

os_memset(nonce, 0, sizeof(nonce));
pchannel = wpabuf_put(req, 4 + 16 + 1);
os_memcpy(pchannel, nonce + 12, 4);
os_memset(pchannel + 4, 0, 16); /* Tag */
pchannel[4 + 16] = EAP_PSK_R_FLAG_DONE_SUCCESS << 6;
wpa_hexdump(MSG_INFO, "EAP-PSK: PCHANNEL (plaintext)",
        pchannel, 4 + 16 + 1);

// Imprimir el nonce antes de la llamada a aes_128_eax_encrypt
printf_hex(nonce, sizeof(nonce));

// Imprimir el header (wpabuf_head(req))
printf_hex(wpabuf_head(req), 22);

// Imprimir el data (pchannel + 4 + 16, 1)
printf_hex(pchannel + 4 + 16, 1);

if (aes_128_eax_encrypt(data->tek, nonce, sizeof(nonce),
            wpabuf_head(req), 22,
            pchannel + 4 + 16, 1, pchannel + 4))
    goto fail;

// Imprimir el tag después de la llamada a aes_128_eax_encrypt
printf_hex(pchannel + 4, 16);

wpa_hexdump(MSG_INFO, "EAP-PSK: PCHANNEL (encrypted)",
        pchannel, 4 + 16 + 1);

return req;
```

Figura 6.93.- Fragmento del código del script eap_psk.c con los prints para ver los campos del canal protegido del tercer mensaje EAP-PSK

6.4.- EVALUACIÓN DE LA SOLUCIÓN PROPUESTA

La longitud de los mensajes y el número total de intercambios son factores determinantes para evaluar el rendimiento de los protocolos de comunicación en entornos restringidos, como los típicos del IoT. En esta sección, se realiza una comparación entre la implementación original de CoAP-EAP y la implementación actual (v12), descrita en el apartado 6.3 de este documento. El análisis se centra exclusivamente en la capa CoAP-EAP, excluyendo las interacciones con la infraestructura AAA subyacente (mensajes RADIUS).

El propósito principal de esta comparación es identificar las diferencias entre ambas versiones en términos de la cantidad de mensajes necesarios para el proceso de bootstrapping y el tamaño total de los datos transmitidos. Mientras que la implementación original de CoAP-EAP se centraba en minimizar el tamaño de los mensajes individuales, la versión más reciente del Draft incorpora mejoras destinadas a aumentar la seguridad, la flexibilidad y la interoperabilidad, lo que podría tener implicaciones en la longitud de los mensajes y en la carga de comunicación global.

Esta comparación se llevará a cabo mediante la tabla 6.1, en la que se detalla la longitud (en bytes) de cada mensaje en ambas implementaciones desde dos perspectivas diferentes: una representa la longitud de la EAP lower layer, excluyendo el tamaño del mensaje EAP (columna LL), y la otra incluye la longitud total, sumando el mensaje EAP (columna LL+EAP). En el caso de CoAP-EAP, la columna LL incluye la longitud de la cabecera CoAP (4 bytes), la lista de opciones CoAP (de longitud variable) y la carga útil, excluyendo el tamaño del mensaje EAP. La columna LL+EAP, por su parte, suma la longitud del mensaje EAP a los valores mostrados en la columna LL. Además, la tabla proporcionará información adicional sobre el número total de mensajes intercambiados, la suma total de bytes enviados, y el porcentaje de variación en el tamaño de los mensajes entre las dos versiones, indicando si el tamaño ha aumentado o disminuido.

	CoAP-EAP versión Original			CoAP-EAP última versión			% Incremento	
	Mensaje	**LL**	**LL+EAP**	**Mensaje**	**LL**	**LL+EAP**	**LL**	**LL+EAP**
	POST	13	13	POST	44	44		
	POST (nonce-c)	18	18	ACK	53	53		
	ACK (nonce-s)	20	20	(Continue) *				
	POST (Req/Id)	17	22	POST (Req/Id)	31	51		
	ACK (Res/Id)	9	20	ACK (Res/Id)	21	38		
	POST (EAP-PSK-1)	17	46	POST (EAP-PSK-1)	31	60		
	ACK (EAP-PSK-2)	9	69	ACK (EAP-PSK-2)	21	81		
	POST (EAP-PSK-3)	17	76	POST (EAP-PSK-3)	31	90		
	ACK (EAP-PSK-4)	9	52	ACK (EAP-PSK-4)	21	64		
	POST (EAP Success)	36	40	POST (EAP Success)	31	92		
	ACK	27	27	ACK (Changed)	30	30		
TOTAL	**11 mensajes**	**192**	**403**	**10 mensajes**	**314**	**603**	**38.9%**	**33.2%**

Tabla 6.1.- Comparativa del tamaño de los mensajes entre la versión original y la última versión implementada de CoAP-EAP

En su versión original, la implementación de CoAP-EAP utilizaba un total de 11 mensajes para completar el flujo de autenticación, con una cantidad total de 192 bytes de datos enviados solo considerando CoAP (LL) y 403 bytes considerando también los datos de EAP (LL+EAP). En cambio, la última versión del borrador reduce el número de mensajes a 10, alcanzando una suma total de 314 bytes de datos en LL y 603 bytes en LL+EAP. Esto implica un incremento del 38,9 % en la capa base y del 33,2 % cuando se incluye EAP, lo que refleja las mejoras y ajustes realizados durante el proceso de estandarización.

La evolución del protocolo responde a la necesidad de equilibrar eficiencia e interoperabilidad, un compromiso clave en el contexto del IETF. Para hacer el sistema estandarizable y adoptable globalmente, se ha mejorado el diseño y se han añadido elementos que incrementan el tamaño de los mensajes, pero que facilitan la interoperabilidad, la seguridad y la flexibilidad del protocolo, logrando así un equilibrio entre robustez y tamaño.

Al observar los mensajes específicos, destacan los siguientes puntos:

- **POST inicial (mensaje de activación):** En la implementación original, el mensaje POST inicial se enviaba a la URI '/boot'. Sin embargo, en la última versión, esta URI se ha reemplazado por '/.well-known/coap-eap', siguiendo las recomendaciones del IETF. No obstante, este cambio aumenta el número de opciones Uri-Path en el mensaje, ya que cada segmento de la ruta (por ejemplo, .well-known y coap-eap) se codifica como una opción independiente. Además, otra cosa en la que difiere este mensaje respecto al de la versión original, es que incluye las opciones Content-Format (application/link-format para este primer mensaje) y No-Response, junto con un payload que indica al EAP Authenticator el nombre del recurso al que debe enviar el primer mensaje EAP Request ID. Todo esto contribuye a incrementar notablemente el tamaño de este primer mensaje.

- **ACK con código Continue:** Este mensaje está marcado con un asterisco porque no es necesario en el flujo teórico de esta última versión del Draft. Sin embargo, fue necesariamente añadido debido a cómo está implementada la biblioteca aiocoap, ya que el EAP Peer requiere esta confirmación para que el intercambio continue funcionando correctamente, como ya se explicó anteriormente. Tener que incluir este mensaje también incrementa el tamaño total de datos enviados.

- **EAP Request/Response ID:** La negociación de cipher suite en la versión original era limitada y no permitía una selección dinámica entre las partes. Esta negociación se realizaba mediante valores predefinidos y carecía de flexibilidad para adaptarse a diferentes contextos. En la última versión, se introduce un enfoque mejorado mediante la codificación CBOR. Este mecanismo permite que el EAP Peer y el EAP Authenticator intercambien un array con las cipher suites soportadas, logrando una negociación dinámica y extensible. Este diseño no solo incrementa la interoperabilidad al admitir una mayor variedad de algoritmos, sino que también facilita la incorporación de nuevas cipher suites en el futuro sin necesidad de modificar la estructura del protocolo. Ese es el principal motivo por el que el tamaño de estos dos mensajes ha aumentado sustancialmente.

  Asimismo, estos mensajes también incorporan la opción Content-Format y, como los nombres de recursos son más largos (ahora deben cumplir con el formato

path/eap/counter), se incrementa la cantidad de las opciones Uri-Path en el EAP Request ID y de las opciones Location-Path en el EAP Response ID.

- **POST con los mensajes del método EAP-PSK:** El ligero incremento del tamaño que presentan estos mensajes respecto a la versión original es también debido a la opción Content-Format y al nuevo diseño de los nombres de los recursos. Este diseño hace que los nombres sean más descriptivos y mejoran la claridad y la gestión del flujo, pero también incrementa el número de opciones Uri-Path en los mensajes POST y el número de opciones Location-Path en las respuestas ACK, puesto que cada segmento de la ruta se codifica como una opción independiente.

- **POST EAP-Success y ACK con código CHANGED:** El mayor tamaño de estos mensajes se debe a la incorporación de OSCORE. En la implementación original se empleaba AUTH, que era una opción diseñada específicamente para CoAP que contenía una MAC (Message Authentication Code) para proteger la integridad de los mensajes. Esta opción Auth se utilizaba como un marcador simple para indicar la autenticación exitosa en los mensajes EAP-Success. Si bien cumplía con su propósito, no ofrecía garantías avanzadas de seguridad, dejando vulnerabilidades en cuanto a la integridad y a la confidencialidad de los mensajes. En la última versión, la opción AUTH ha sido reemplazada por OSCORE, que proporciona un cifrado robusto para el contenido del mensaje y una autenticación basada en AEAD. Esto garantiza tanto la confidencialidad como la integridad de los mensajes, representando una mejora significativa al proteger los datos intercambiados frente a ataques de intermediarios y otras amenazas.

Por último, es interesante señalar que aunque la última versión ha aumentado el tamaño de los mensajes, el total de bytes enviados (603 bytes) sigue siendo muy similar a los 596 bytes de la implementación con PANATIKI , que es la opción más eficiente entre las soluciones basadas en PANA. Esto demuestra que las mejoras introducidas no sacrifican la eficiencia global del protocolo. Además, la reducción en el número de mensajes a 10 (e incluso 9 si no fuese necesario el ACK con código Continue) representa una ventaja crucial en redes con alta latencia o ancho de banda limitado, reforzando su idoneidad para escenarios IoT.

7. Ejecución y pruebas.

En esta sección se detallan los pasos que hay que seguir para poder ejecutar la implementación del Draft y así poder visualizar los resultados. Se van a probar cuatro escenarios diferentes con el fin de comprobar que todo funciona correctamente y que se cumplen los requisitos detallados en la sección 5.1. También cabe destacar que, durante el desarrollo, se realizaron pruebas unitarias sobre partes pequeñas del código de forma rigurosa y minuciosa, con el objetivo de detectar fallos con mayor facilidad y corregirlos. Esto permitió evitar errores al integrar todo en la implementación final.

7.1.- PASOS A SEGUIR PARA LA EJECUCIÓN

A continuación, se detallan los pasos a seguir para ejecutar la implementación:

1. Abrir el programa VMWare Workstation y seleccionar la máquina virtual llamada “TFG Daniel”.

2. Ejecutar la máquina virtual, introducir la contraseña (student) y esperar a que cargue completamente.

3. Abrir un terminal en la máquina virtual para lanzar FreeRadius, que es un servidor de autenticación de código abierto ampliamente utilizado que implementa el protocolo RADIUS.

4. Ejecutar los siguientes comandos:

   - ✓ **cd /home/student/freeradius-psk/sbin:** Este comando lleva al directorio /sbin dentro de la instalación de FreeRADIUS, que está ubicado en /home/student/freeradius-psk.

   - ✓ **Export LD_PRELOAD = /home/student/coap-eap-tfg/freeradius-2.0.2-psk/hostapd/eap_example/libeap.so:** Este comando fuerza la carga de la

biblioteca libeap.so, que contiene funcionalidades específicas relacionadas con el protocolo EAP, antes de que FreeRADIUS se inicie. Esto es necesario para que FreeRADIUS utilice esas funciones y soporte la autenticación mediante EAP-PSK de la implementación.

- ✓ **./radiusd -X:** Este comando inicia el servidor FreeRADIUS en modo depuración, mostrando información detallada sobre su funcionamiento. Es útil para verificar que FreeRADIUS está funcionando correctamente, ver información detallada sobre el proceso de arranque, observar los intercambios de mensajes RADIUS y solucionar problemas o errores en la configuración o en la implementación.

5. Abrir el programa Wireshark en el ordenador, seleccionar las interfaces de red en las que queremos capturar el tráfico y aplicar los filtros de captura oportunos, en este caso, introducimos este filtro: **coap or radius**. Esto permitirá ver los mensajes CoAP intercambiados entre el EAP Peer y el EAP Authenticator y los mensajes RADIUS intercambiados entre el cliente RADIUS presente en el EAP Authenticator y el EAP Server.

6. Abrir PyCharm en el equipo, buscar y cargar el proyecto en el que se han desarrollado todos los scripts necesarios para la implementación.

7. Seleccionar el script Controller.py como archivo actual del proyecto y ejecutarlo haciendo click en la opción Run. Se ejecuta primero este archivo porque al principio es el EAP Authenticator el que va a actuar como servidor.

8. Seleccionar el archivo Peer.py como archivo actual del proyecto y ejecutarlo haciendo click en la opción Run. Es el EAP Peer el que actúa como cliente ya que lanza la primera petición POST a la URI .well-known/coap-eap, es por eso por lo que se ejecuta después del script Controller.py.
   El intercambio de roles está gestionado automáticamente dentro de ambos scripts.

Una vez realizados estos pasos, ya se pueden visualizar los resultados de la implementación en tres lugares diferentes:

- **En PyCharm:** En la consola se pueden ver, entre otras cosas, los códigos y los payloads de cada uno de los mensajes CoAP enviados y recibidos, los resultados de parsear y construir los mensajes EAP-PSK, los parámetros del contexto de seguridad OSCORE de ambas entidades, la cipher suite elegida, el tiempo de vida de la sesión...
- **En Wireshark:** Se pueden observar los detalles de cada paquete del flujo de operación de CoAP-EAP:
    - Paquetes RADIUS: Cabecera de los mensajes y el payload con los atributos (e.g., EAP-Message, User-Name).
    - Paquetes CoAP: Se pueden ver los campos de los mensajes del protocolo CoAP y los campos de los mensajes EAP-PSK encapsulados en ellos.
- **En la terminal de FreeRADIUS:** Se pueden visualizar los detalles de las claves (PSK, claves de sesión y claves de larga duración), de la creación del canal protegido y de los campos de los mensajes EAP-PSK. También se pueden ver el tipo de los mensajes RADIUS intercambiados entre el cliente y el servidor y sus atributos.

Gracias a esto se puede asegurar que se intercambian todos los mensajes, en el orden adecuado y que su contenido es el correcto.

7.2.- ESCENARIOS DE PRUEBA

En este apartado se analizarán cuatro escenarios de prueba fundamentales para evaluar la implementación del servicio de autenticación basado en EAP para CoAP. Cada uno de estos escenarios representa una situación típica en el proceso de autenticación y tiene como objetivo verificar que los resultados visualizados en Wireshark concuerden con el flujo de operación esperado. Los escenarios que se van a examinar son los siguientes:

1. **Autenticación exitosa ideal**: Prueba básica donde se evalúa el flujo completo de autenticación sin interrupciones, garantizando que los mensajes intercambiados correspondan con una autenticación exitosa.

2. **Autenticación exitosa con petición duplicada o retransmisión tardía**: Este escenario simula condiciones en las que se produce una duplicación o retraso en la transmisión de paquetes. El objetivo es comprobar que la implementación maneje correctamente estas anomalías sin afectar al resultado final de la autenticación.

3. **Autenticación fallida**: Este escenario prueba una autenticación que no cumple los requisitos de seguridad, lo que debería resultar en una denegación de acceso. Aquí, se evaluará que los paquetes reflejen adecuadamente el proceso de rechazo de autenticación.

4. **Eliminación del estado CoAP-EAP tras autenticación exitosa**: En este caso, se examina el comportamiento del sistema cuando, después de una autenticación exitosa, el cliente pierde el estado autenticado. La prueba permite verificar que el sistema finalice correctamente la sesión y restrinja el acceso según lo esperado.

Para cada escenario, se describirá de manera resumida el contenido de los paquetes capturados, permitiendo una comprensión clara de los mensajes y su flujo en cada caso. Esta sección asegura que la implementación del sistema cumple con los comportamientos previstos en cada tipo de situación de autenticación.

7.2.1.- Autenticación exitosa ideal

En este escenario, se analizará el flujo de operación ideal en el que el proceso de autenticación se completa de manera exitosa, sin interferencias ni errores. Este flujo sigue paso a paso el intercambio de mensajes entre el EAP Peer (dispositivo IoT), el EAP Authenticator y el EAP Server, asegurando que cada entidad cumpla con su rol en el proceso de autenticación.

En primer lugar, se presenta un diagrama de secuencia (MSC) que detalla el flujo teórico de mensajes esperados en un proceso de autenticación exitoso. Este diagrama permitirá visualizar claramente cada uno de los paquetes que deben intercambiarse, incluyendo los códigos específicos y los valores de control necesarios para confirmar la

autenticación. El objetivo de este diagrama es proporcionar una referencia visual del flujo de operación ideal del Draft.

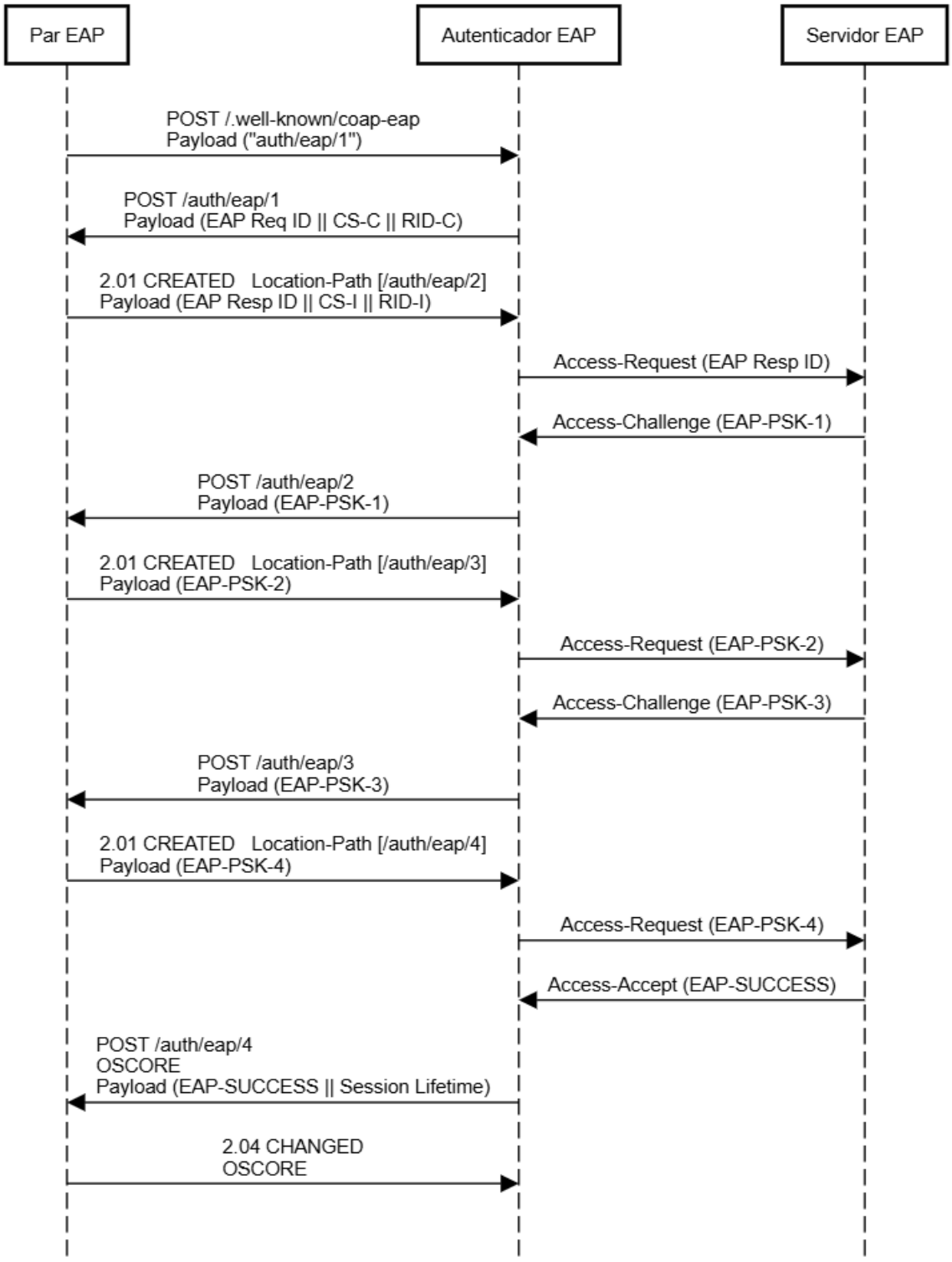


Figura 7.1.- Diagrama MSC del escenario de autenticación exitosa ideal

A continuación, se muestra la captura de tráfico obtenida en Wireshark al ejecutar este escenario con la implementación explicada en la sección 6.3. A partir de esta captura, se comentarán los paquetes reales observados en comparación con el flujo teórico del diagrama MSC. Se verificará que cada mensaje esperado esté presente en la captura y que cada uno cuente con los códigos y datos correctos según el estándar, confirmando así que la implementación sigue el flujo adecuado para una autenticación exitosa.

coap or radius

No.	Time	Source	Destination	Protocol	Length	Info
325	11.537539	192.168.0.3	192.168.0.3	CoAP	76	CON, MID:17235, POST, TKN:5f 39, /.well-known/coap-eap
326	11.538664	192.168.0.3	192.168.0.3	CoAP	85	ACK, MID:17235, 2.31 Continue, TKN:5f 39, /.well-known/coap-eap
496	17.321690	::1	::1	CoAP	103	CON, MID:41634, POST, TKN:1d dc, coap://localhost/auth/eap/61
497	17.322750	::1	::1	CoAP	90	ACK, MID:41634, 2.01 Created, TKN:1d dc, coap://localhost/auth/eap/61
483	17.326143	192.168.217.1	192.168.217.128	RADIUS	141	Access-Request id=0
484	17.326540	192.168.217.128	192.168.217.1	RADIUS	105	Access-Challenge id=0
485	17.329971	192.168.217.1	192.168.217.128	RADIUS	159	Access-Request id=1
486	17.330600	192.168.217.128	192.168.217.1	RADIUS	129	Access-Challenge id=1
498	17.333033	::1	::1	CoAP	112	CON, MID:41635, POST, TKN:1d dd, coap://localhost/auth/eap/62
499	17.334464	::1	::1	CoAP	133	ACK, MID:41635, 2.01 Created, TKN:1d dd, coap://localhost/auth/eap/62
487	17.336143	192.168.217.1	192.168.217.128	RADIUS	209	Access-Request id=2
488	17.336592	192.168.217.128	192.168.217.1	RADIUS	159	Access-Challenge id=2
500	17.340037	::1	::1	CoAP	142	CON, MID:41636, POST, TKN:1d de, coap://localhost/auth/eap/63
501	17.342192	::1	::1	CoAP	116	ACK, MID:41636, 2.01 Created, TKN:1d de, coap://localhost/auth/eap/63
489	17.344141	192.168.217.1	192.168.217.128	RADIUS	192	Access-Request id=3
490	17.344604	192.168.217.128	192.168.217.1	RADIUS	209	Access-Accept id=3
502	17.349375	::1	::1	CoAP	144	CON, MID:41637, POST, TKN:1d df, coap://localhost/auth/eap/64
503	17.350917	::1	::1	CoAP	82	ACK, MID:41637, 2.04 Changed, TKN:1d df, coap://localhost/auth/eap/64

Figura 7.2.- Captura Wireshark del escenario de autenticación exitosa ideal

Como se puede observar en la captura de Wireshark, todos los paquetes del protocolo CoAP tienen un identificador Message ID (MID) y un Token que permiten garantizar la correcta asociación entre las solicitudes y las respuestas. El MID es un número único de 16 bits generado por el cliente o el servidor que asegura la correlación entre mensajes a nivel del protocolo. Por ejemplo, en el caso de una solicitud enviada por el cliente con MID 17235, el servidor responde con el mismo MID en su respuesta, lo que indica que esa respuesta corresponde a esa solicitud. Además, el Token, que es un identificador de aplicación generado por el cliente, también se incluye en los mensajes. Este Token tiene como propósito identificar las solicitudes activas cuando hay múltiples intercambios simultáneos. En la captura, se puede ver que si el mensaje de solicitud lleva el Token 5f39, la respuesta del servidor incluye ese mismo Token, permitiendo que el cliente reconozca a qué solicitud corresponde.

Por otro lado, los mensajes del protocolo RADIUS utilizan un Packet Identifier (id), que cumple una función similar a la del MID en CoAP. Este identificador, que es un número

entero de 1 byte generado por el cliente, asegura que las respuestas estén correctamente asociadas con las solicitudes originales. Como se muestra en la captura de Wireshark, cuando el cliente envía una solicitud RADIUS, por ejemplo, un Access-Request, incluye un Packet Identifier único, como el valor 1. El servidor responde con un mensaje, como un Access-Accept o Access-Challenge, en el que reutiliza exactamente el mismo Packet Identifier. Esto permite al cliente saber que esa respuesta corresponde a su solicitud.

En ambos casos, tanto en CoAP como en RADIUS, los identificadores se utilizan para mantener el orden y garantizar la coherencia en la comunicación. De esta forma, se puede verificar que las respuestas recibidas corresponden a las solicitudes enviadas, lo que resulta esencial para el correcto funcionamiento de estos protocolos, especialmente en redes con múltiples intercambios de mensajes simultáneos.

El primer paquete capturado (paquete número 325) corresponde al mensaje de activación. Se trata de un paquete de solicitud CoAP de tipo CON (Confirmable) con código POST, enviado desde el EAP Peer, que actúa como cliente, hacia el EAP Authenticator, que actúa como servidor en la dirección 192.168.0.3 (dirección del adaptador de LAN Inalámbrica WI-FI 2) y utilizando el puerto CoAP por defecto (5683). Este mensaje consta de las siguientes opciones:

- **Uri-Path:** La opción Uri-Path es esencial para indicar el recurso específico dentro del servidor CoAP al que se dirige la solicitud. Cada segmento de la ruta del recurso se representa como una opción independiente de tipo URI-Path. Estas opciones juntas forman la URI completa del recurso solicitado (‘.well-known/coap-eap’).
- **Content-Format:** application/link-format, indicando que el payload sigue el formato estándar de enlaces (links) definido en CoAP.
- **No-Response:** 1, sugiriendo que el cliente no espera una respuesta explícita del servidor para este mensaje.

La carga útil (payload) contiene información adicional en el formato application/link-format. Se puede observar en texto plano que el contenido incluye la URI ‘auth/eap/61’, que sirve para indicar a qué recurso debe enviar el autenticador la petición con el mensaje EAP Request ID. Esta URI, como se puede apreciar, cumple con el formato path/eap/counter

recomendado por el estándar, donde "path" es la ruta del dispositivo IoT elegida para hacer que la URI sea única, "eap" es el nombre que indica que la URI es para el EAP Peer y "counter" es un número único que se incrementa con cada nueva solicitud EAP enviada por el EAP Authenticator.

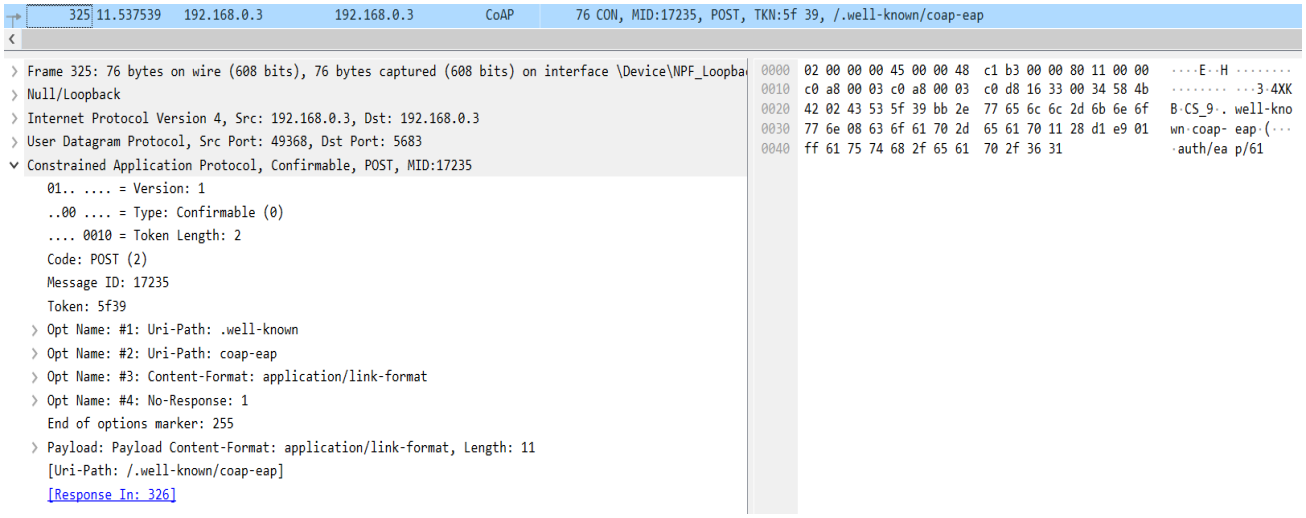


Figura 7.3.- Contenido del primer paquete de la autenticación exitosa ideal visto en Wireshark

El segundo paquete capturado (paquete número 326) corresponde a una respuesta CoAP enviada por el autenticador al EAP Peer de tipo ACK (Acknowledgement) para confirmar la recepción del paquete de solicitud anterior. El código de esta respuesta es 2.31 Continue para indicar que el proceso de autenticación ha comenzado y que el autenticador puede continuar enviando el siguiente mensaje del flujo.

La única opción que presenta es el Content-Format, que será application/octet-stream para indicar que la carga útil contiene un flujo de bytes. En realidad, el Content-Format tanto de este paquete como de todos los paquetes posteriores debería de ser application/coap-eap pero, como ya se comentó en la sección de implementación, Wireshark no lo reconoce y por eso se emplea octet-stream.

La carga útil de esta respuesta, como se puede ver en texto plano, incluye la cadena “Mensaje de activación recibido correctamente”.

Este paquete no aparece en el flujo de operación teórico puesto que no es necesario que se responda al mensaje de activación pero está presente porque, como se explicó en la sección 6.3, la forma en la que está diseñada la biblioteca aiocoap requiere que se responda a este mensaje obligatoriamente para el correcto funcionamiento de la implementación.

Otro aspecto a destacar es que los únicos mensajes en los que el EAP Peer es el cliente CoAP y el EAP Authenticator es el servidor CoAP son los dos primeros.

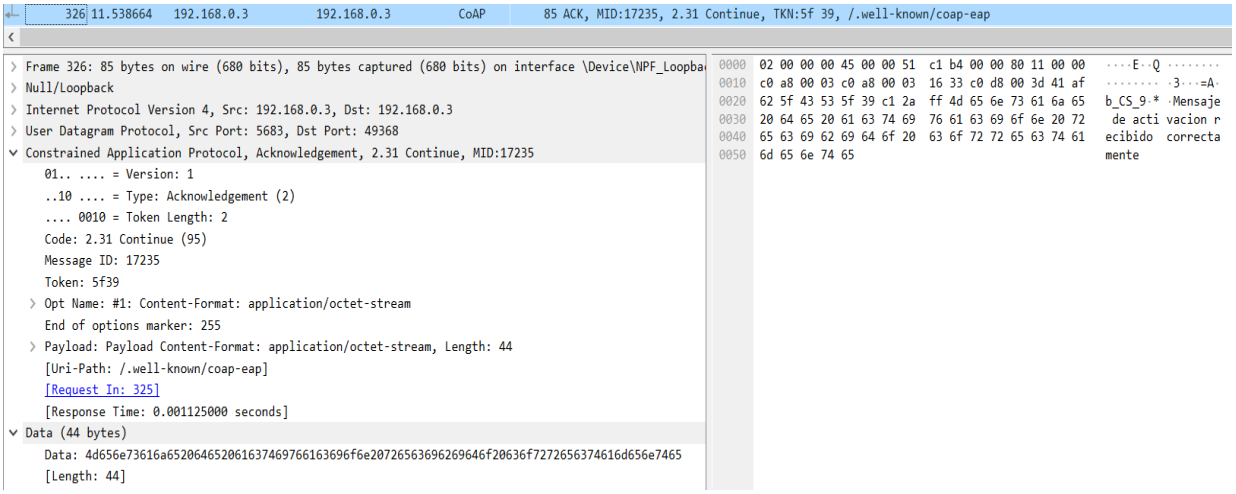


Figura 7.4.- Contenido del segundo paquete de la autenticación exitosa ideal visto en Wireshark

El tercer paquete capturado (paquete número 496) se trata de una solicitud CoAP de tipo CON (Confirmable) con código POST, enviada desde el EAP Authenticator, que actúa como cliente, hacia el EAP Peer, que actúa como servidor en la dirección 127.0.0.1 (en IPv6 se pone ::1 para hacer referencia a la dirección de loopback, también llamada localhost) y utilizando el puerto CoAP por defecto (5683). Este paquete consta de las siguientes opciones:

- **Uri-Host:** Esta opción es utilizada para indicar el destino específico de la solicitud cuando un cliente CoAP se comunica con un servidor. En este mensaje, la opción Uri-Host tiene el valor localhost. Esto significa que el cliente CoAP se está dirigiendo a un servidor identificado como localhost, que hace referencia a la propia máquina en la que está ejecutándose el cliente, es decir, la dirección de loopback.

- **Uri-Path:** La URI del recurso solicitado está compuesta por tres segmentos, cada uno de los cuales se representa como una opción independiente de tipo URI-Path. Estas opciones juntas forman la URI completa del recurso solicitado (‘auth/eap/61’). Esta URI es el contenido de la carga útil del primer mensaje (mensaje de activación) por lo que se está cumpliendo con lo establecido en el Draft.
- **Content-Format:** application/octet-stream, indicando que el contenido del payload es un flujo de bytes.

La carga útil (payload) mostrada en hexadecimal en el campo Data tiene un tamaño de 20 bytes y contiene el primer mensaje EAP, al que se le llama EAP Request Identity. Los primeros nueve bytes (“010700050183000102”) constituyen una cadena fija que puede incluir campos específicos del protocolo o valores iniciales necesarios para la comunicación. Los once bytes restantes (“a2018500010203040241 01”) corresponden a una estructura de datos codificada en CBOR (Concise Binary Object Representation). CBOR es un formato de codificación binaria que permite representar estructuras de datos complejas en un formato compacto y eficiente. En este caso, la estructura CBOR incluye:

- **Una lista de cipher suites:** Representa los posibles algoritmos o conjuntos de cifrado que el EAP Authenticator (Extensible Authentication Protocol) sugiere o admite para la negociación de Cipher suite. Esto es esencial para establecer la seguridad de la comunicación, ya que permite acordar una cipher suite compatible entre ambas partes.

- **El Recipient ID del EAP Authenticator (RID-C):** El RID-C ayuda a distinguir al EAP Authenticator en las comunicaciones, asegurando que los mensajes se entreguen a la entidad correcta. Este ID se emplea en la configuración del contexto de seguridad OSCORE.

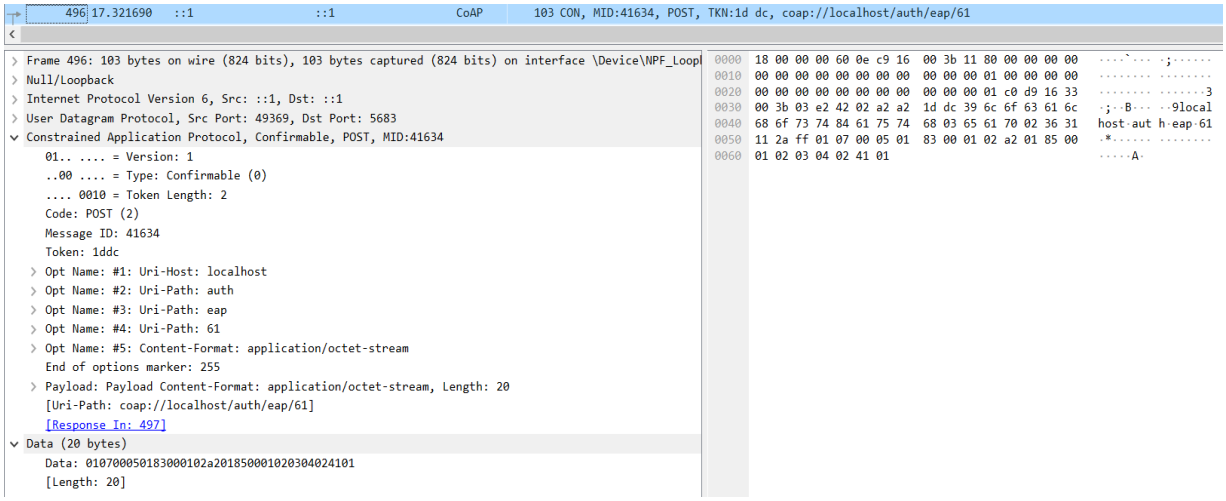


Figura 7.5.- Contenido del tercer paquete de la autenticación exitosa ideal visto en Wireshark

Se puede comprobar que la URI del recurso, la estructura CBOR y el payload de este tercer paquete son correctos revisando la consola del script controller.py:

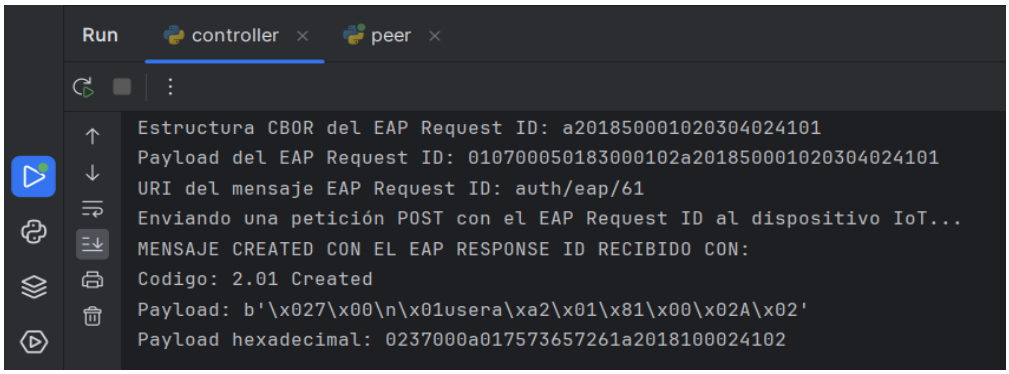


Figura 7.6.- Información del tercer paquete visto en la consola del script controller.py

El cuarto paquete capturado (paquete número 497) corresponde a una respuesta CoAP enviada por el autenticador al EAP Peer de tipo ACK (Acknowledgement) para confirmar la recepción del paquete de solicitud anterior. Este paquete de respuesta consta de las siguientes opciones:

- **Location-Path:** Esta opción es utilizada en el protocolo CoAP por el servidor para indicar al cliente la URI del recurso que ha sido creado como resultado de una solicitud POST exitosa. Cada opción Location-Path representa un segmento de la URI, y el cliente las concatenará en el orden recibido para formar la URI completa.

En el caso de este mensaje, la URI incluida en las opciones Location-Path es 'auth/eap/62', que sirve para indicar a qué recurso debe enviar el autenticador la petición con el primer mensaje del método EAP-PSK (EAP-PSK-1).

- **Content-Format:** application/octet-stream, indicando que el contenido del payload se presenta como un flujo de bytes.

El código de esta respuesta es 2.01 Created para confirmar que la solicitud es válida y que el recurso solicitado fue efectivamente creado o configurado en la URI indicada por las opciones de Location-Path ('auth/eap/62' en este caso). Es decir, el mensaje de respuesta 2.01 Created asegura al cliente (EAP Authenticator) que ahora existe un recurso en la URI especificada (/auth/eap/62) y que puede acceder o interactuar con él en el futuro usando esa ubicación.

La carga útil (payload) de esta respuesta, mostrada en hexadecimal en el campo Data, tiene un tamaño de 17 bytes y contiene el segundo mensaje EAP, al que se le llama EAP Response Identity. Los primeros cinco bytes ("0237000a01") constituyen una cadena fija que actúa como un identificador o prefijo que ayuda a interpretar el tipo de información contenida en el mensaje. Los siguientes cinco bytes ("7573657261") representan al NAI (Network Access Identifier) del dispositivo IoT, que en este caso es "usera" como se puede ver en texto plano y los siete bytes restantes ("a2018100024102") corresponden a una estructura de datos codificada en CBOR que incluye:

- **Una lista de cipher suites:** Representa el conjunto de cifrado elegido por el EAP Peer.

- **El Recipient ID del EAP Peer (RID-I):** El RID-I ayuda a distinguir al EAP Peer en las comunicaciones, asegurando que los mensajes se entreguen a la entidad correcta. Este ID también se emplea en la configuración del contexto de seguridad OSCORE.

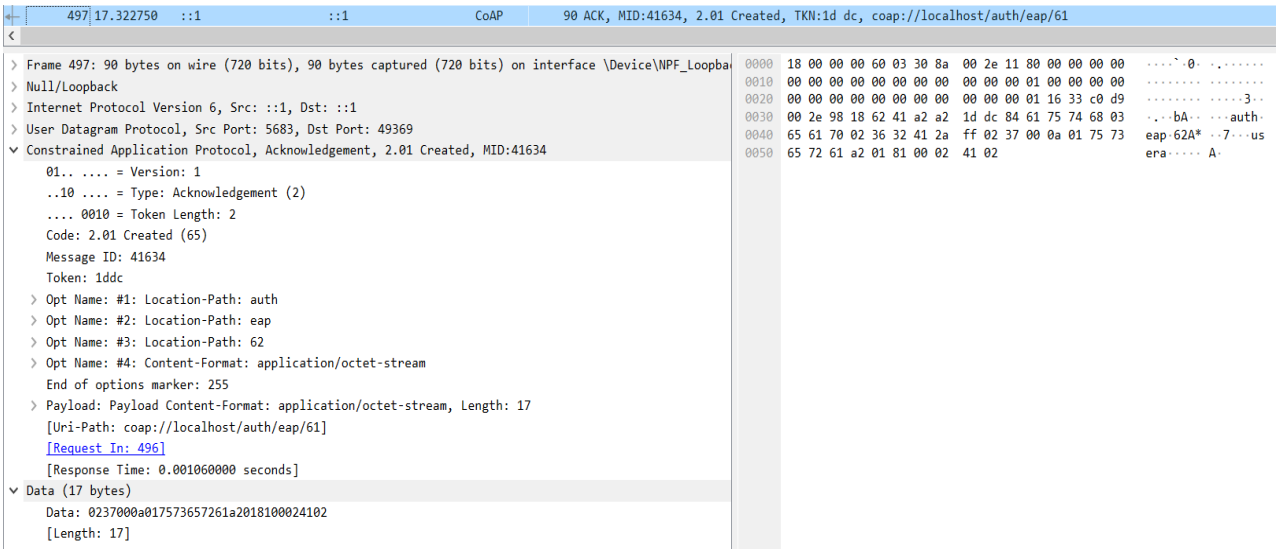


Figura 7.7.- Contenido del cuarto paquete de la autenticación exitosa ideal visto en Wireshark

Se puede comprobar que la estructura CBOR y el payload de este cuarto paquete son correctos revisando la consola del script peer.py. Además, en esta consola también se puede ver la cipher suite elegida, el RID-C y el RID-I tras deserializar las estructuras CBOR del EAP Request Identity y del EAP Response Identity:

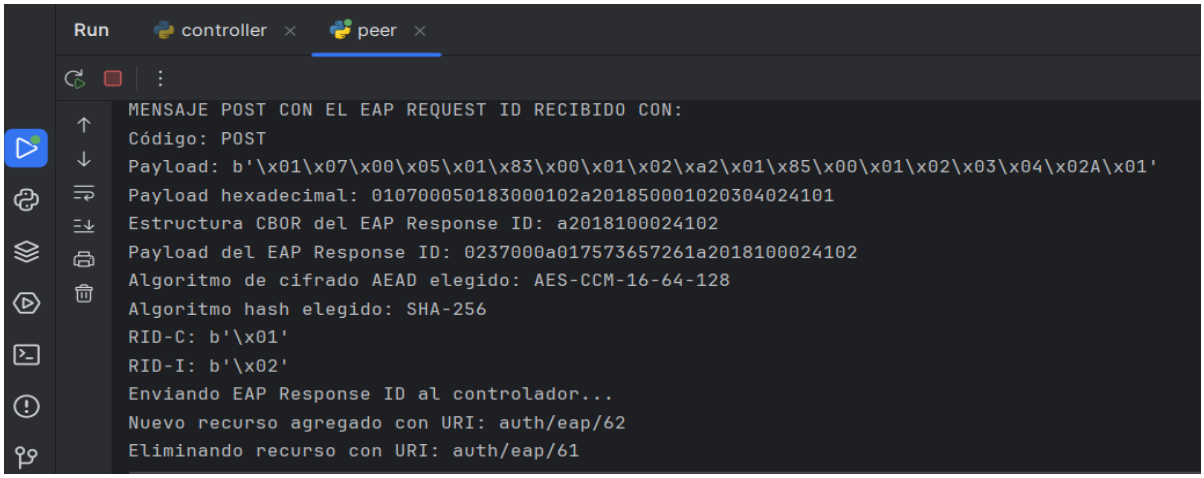


Figura 7.8.- Información del cuarto paquete visto en la consola del script peer.py

El quinto paquete capturado (paquete número 483) se trata de un Access-Request enviado desde el cliente RADIUS del EAP Authenticator (dirección IP 192.168.217.1) al servidor RADIUS que actúa como EAP Server, el cual escucha en la dirección IP

192.168.217.128, utilizando el protocolo UDP con el puerto 1812 como destino. Este Access-Request consta de los siguientes atributos:

- **User-Name:** usera, que identifica el nombre de usuario o el NAI (Network Access Identifier) del dispositivo IoT solicitante.
- **NAS-IP-Address:** 127.0.0.1, que indica la dirección IP del NAS.
- **Framed-MTU:** 1400, indicando el tamaño máximo de la unidad de transmisión.
- **Calling-Station-Id:** 00-00-00-00-00-00, que representa la dirección MAC del dispositivo solicitante.
- **NAS-Port-Type:** Wireless-802.11, que especifica el tipo de conexión del NAS, en este caso, una red inalámbrica.
- **Connect-Info:** CON, que indica que hay una conexión activa entre el cliente y el servidor pero sin especificar el tipo de esa conexión.
- **EAP-Message:** Esta es la parte más relevante del mensaje ya que es donde se envía el mensaje EAP. En este caso, el contenido es "0237000a017573657261", que representa el mensaje EAP Response ID sin la estructura CBOR. Dentro de este atributo se proporciona información sobre el mensaje EAP contenido en él. En esta ocasión se detallan cada uno de los campos del mensaje EAP Response Identity:
  - Code: 2 porque es una respuesta.
  - ID: 55
  - Length: 10
  - Type: Identity (1), se utiliza para que el cliente se identifique ante el servidor proporcionando un nombre de usuario.
  - Identity: usera, confirmando el NAI o identidad del dispositivo IoT.
- **Message-Authenticator:** "4ac08361c153fc94d8cffc90dfd15eb0", es un código hash que asegura la integridad del mensaje, detectando cualquier alteración en el mensaje durante el transporte. El valor de este atributo va a ser diferente para cada mensaje RADIUS independientemente de si es Access-Request, Access-Challenge o Access-Accept.

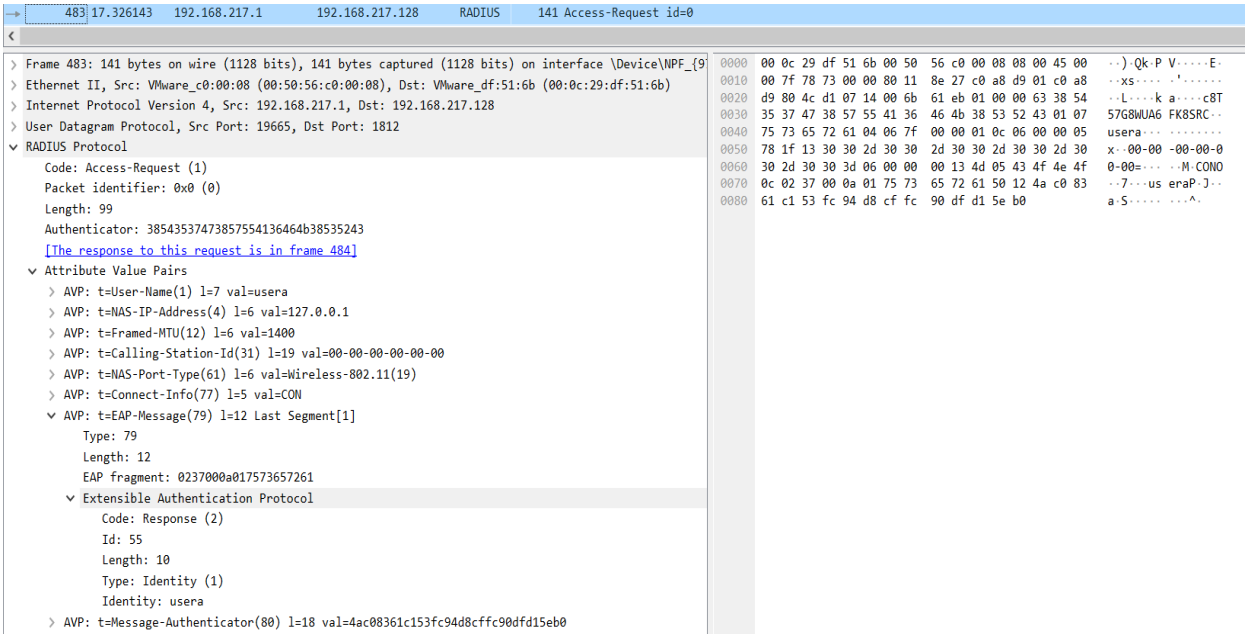


Figura 7.9.- Contenido del quinto paquete de la autenticación exitosa ideal visto en Wireshark

El sexto mensaje capturado (paquete número 484) es un Access-Challenge enviado por el servidor al cliente RADIUS en respuesta al Access-Request anterior. Este Access-Challenge tiene los siguientes atributos:

- **EAP-Message:** En este caso, el contenido es “01c6000501”, un mensaje EAP que insta de nuevo al cliente a proporcionar el NAI al servidor RADIUS. Los campos de este mensaje detallados son:
  - Code: 1 porque es una solicitud.
  - ID: 198
  - Length: 5
  - Type: Identity (1), se utiliza para el cliente proporcione de nuevo su identidad.
- **State:** Este campo se usa para mantener el estado entre múltiples intercambios de Access-Request y Access-Challenge, permitiendo al servidor RADIUS asociar las solicitudes y respuestas que forman parte de la misma sesión de autenticación. En este caso su valor es “d52092276ad95afbded48fc5d0fc5c0d”.
- **Message-Authenticator:** Su valor en este mensaje es “60ff2bbb35dc4e8a2457ce229fcee897”.

Este paquete no aparece en el flujo de operación teórico puesto que el servidor RADIUS debería responder directamente con un Access-Challenge que contenga el primer mensaje del método EAP-PSK pero, como ya se explicó en la sección 6.3, la implementación del servidor RADIUS presenta un bug. Este bug hace que responda al Access-Request que contiene el EAP Response Identity con un Access-Challenge que obliga al cliente a indicar de nuevo la identidad del dispositivo IoT.

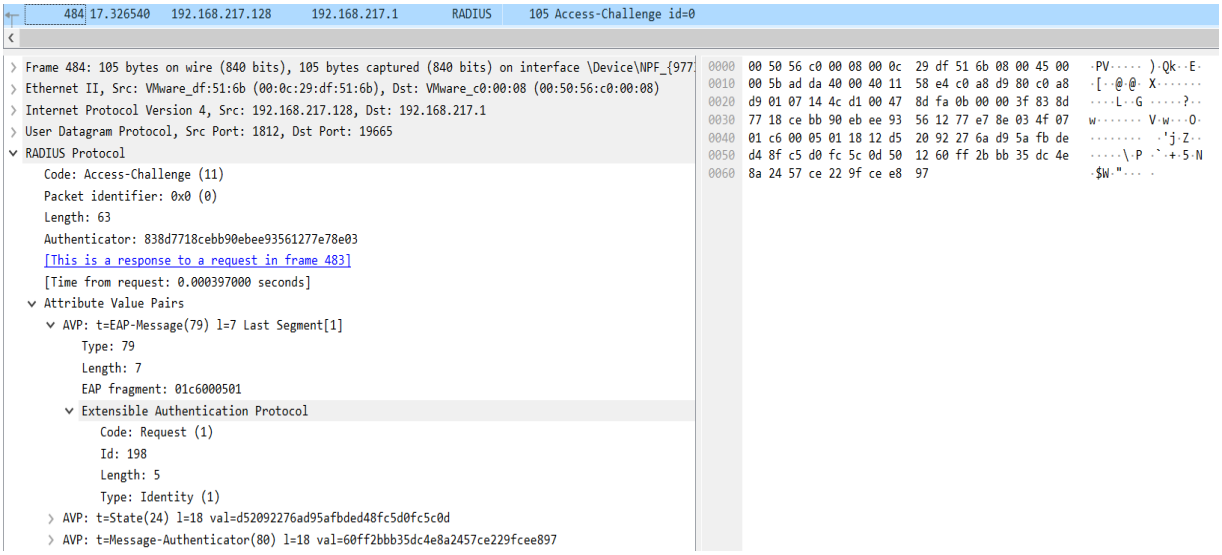


Figura 7.10.- Contenido del sexto paquete de la autenticación exitosa ideal visto en Wireshark

El séptimo paquete capturado (paquete número 485) es el último paquete que no está en el flujo de operación teórico. Se trata de un Access-Request enviado desde el cliente RADIUS hacia el servidor RADIUS que proporciona de nuevo la identidad del dispositivo IoT para solucionar el problema del bug del servidor. Es prácticamente una réplica del Access-Request previo que posee los siguientes atributos:

- **User-Name:** usera
- **NAS-IP-Address:** 127.0.0.1
- **Framed-MTU:** 1400
- **Calling-Station-Id:** 00-00-00-00-00-00
- **NAS-Port-Type:** Wireless-802.11
- **Connect-Info:** CON

- **EAP-Message:** En este caso, el contenido es "02c6000a017573657261", que representa el mensaje EAP Response ID sin la estructura CBOR pero con el ID del mensaje EAP contenido en el atributo EAP-Message del paquete Access-Challenge anterior. Los campos de este mensaje detallados son:
  - Code: 2 porque es una respuesta.
  - ID: 198, para asociar este mensaje de respuesta al de solicitud del Access-Challenge anterior.
  - Length: 10 (longitud del EAP Response ID sin la estructura CBOR).
  - Type: Identity (1)
  - Identity: usera
  - State: El valor de este atributo es el mismo que el del atributo State del Access-Challenge anterior ("d52092276ad95afbded48fc5d0fc5c0d").
- **Message-Authenticator:** Su valor en este mensaje es "54f2b5355b02f97eaada5c9ce4af163c".

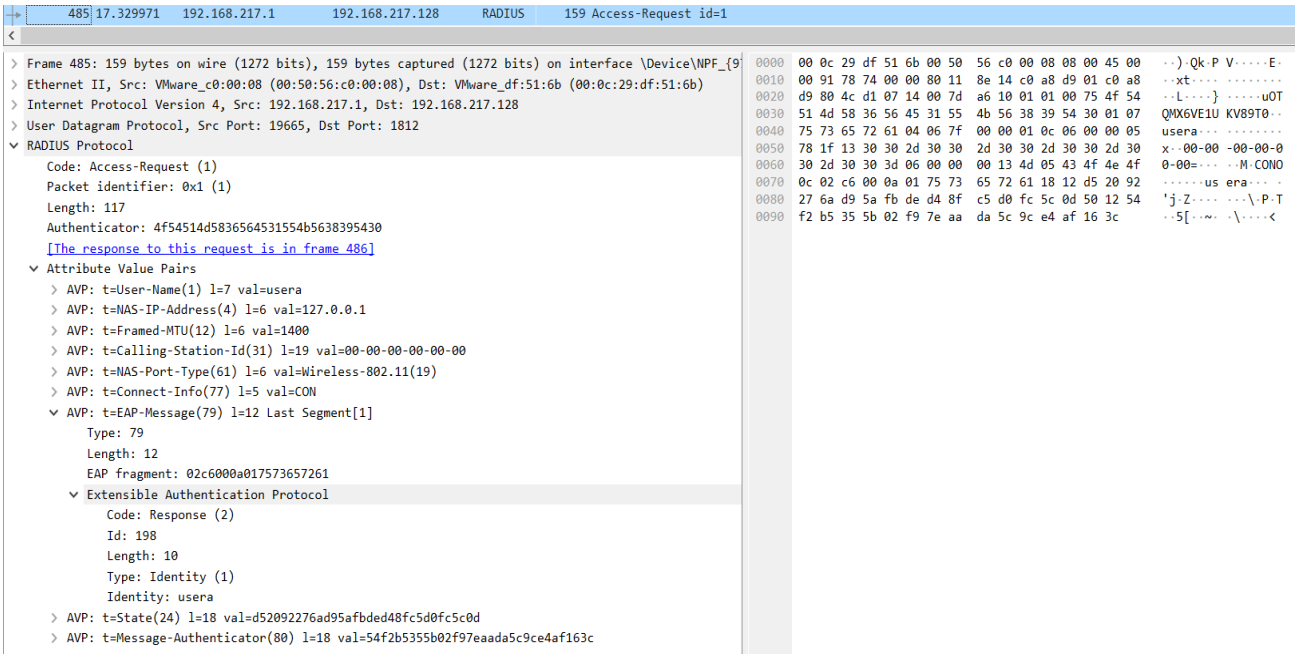


Figura 7.11.- Contenido del séptimo paquete de la autenticación exitosa ideal visto en Wireshark

El octavo mensaje capturado (paquete número 486) es un Access-Challenge enviado por el servidor RADIUS al cliente RADIUS del EAP Authenticator en respuesta al Access-Request anterior. Este Access-Challenge tiene los siguientes atributos:

- **EAP-Message:** En este caso, el contenido es "01c7001d2f0041e5a7769135685f09a50b20f48e0975686f7374617064", que representa el primer mensaje del método EAP-PSK (EAP-PSK-1) construido por el EAP Server (servidor RADIUS). Los campos de este mensaje detallados son:
    - Code: 1 porque es una solicitud.
    - ID: 199
    - Length: 29
    - Type: Pre-Shared Key EAP (EAP-PSK) (47). Indica que el método de autenticación es EAP-PSK.
    - EAP-PSK Flags: 0x00. Configuración de los Flags específicos del primer mensaje EAP-PSK.
    - EAP-PSK RAND_S: 41e5a7769135685f09a50b20f48e0975". Valor aleatorio generado por el servidor (RAND_S), usado en el protocolo EAP-PSK para evitar ataques de repetición y para calcular la MAC_P.
    - EAP-PSK ID_S: Identificador del servidor (ID_S), en este caso establecido como "hostapd".
- State: Al tratarse de un nuevo Access-Challenge el valor de este atributo cambia. En este caso su valor es "7b2e5ae09a97c0d503b47577a2f31792".
- Message-Authenticator: Su valor en este mensaje es "31856bebc5a93e4ee65675e7cad3a584".

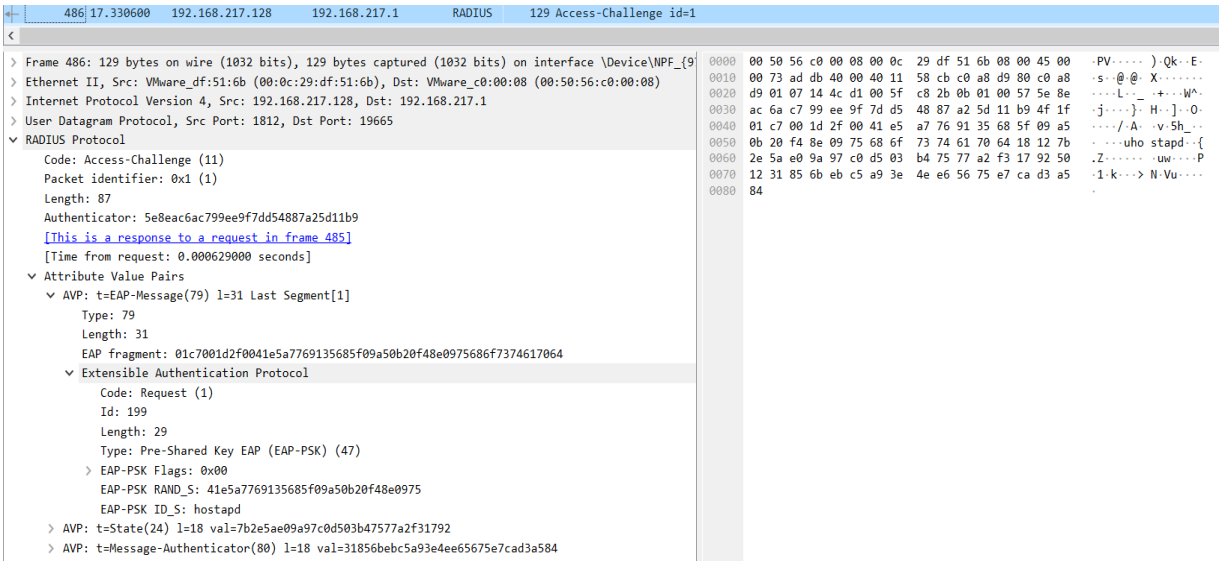


Figura 7.12.- Contenido del octavo paquete de la autenticación exitosa ideal visto en Wireshark

El noveno paquete capturado (paquete número 498) se trata de una solicitud CoAP de tipo CON (Confirmable) con código POST, enviada desde el EAP Authenticator hacia el EAP Peer. Este paquete consta de las siguientes opciones:

- **Uri-Host:** En este mensaje, la opción Uri-Host tiene el valor localhost. Esto significa que el cliente CoAP se está dirigiendo a un servidor identificado como localhost, que hace referencia a la propia máquina en la que está ejecutándose el cliente, es decir, la dirección de loopback.
- **Uri-Path:** La URI del recurso solicitado está compuesta por tres segmentos, cada uno de los cuales se representa como una opción independiente de tipo URI-Path. Estas opciones juntas forman la URI completa del recurso solicitado (‘auth/eap/62’). Esta URI es el contenido de la opción Location-Path del cuarto paquete (solicitud CoAP cuyo payload es el EAP Response Identity) por lo que se está cumpliendo con lo establecido en el Draft.
- **Content-Format:** application/octet-stream, indicando que el contenido del payload es un flujo de bytes.

La carga útil (payload) de este paquete, mostrada en hexadecimal en el campo Data, tiene un tamaño de 29 bytes y su contenido es “01c7001d2f0041e5a7769135685f09a50b20f48e0975686f7374617064”, que representa el primer mensaje del método EAP-PSK (EAP-PSK-1) recibido del EAP Server.

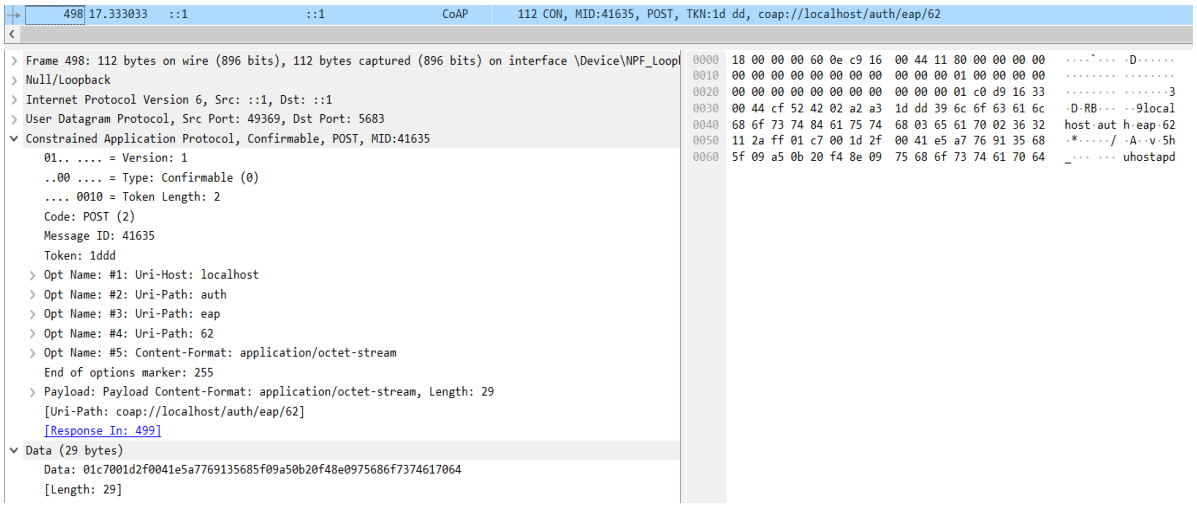


Figura 7.13.- Contenido del noveno paquete de la autenticación exitosa ideal visto en Wireshark

Se puede comprobar que la URI del recurso y el payload de este noveno paquete son correctos revisando la consola del script controller.py:

Figura 7.14.- Información del noveno paquete visto en la consola del script controller.py

El décimo paquete capturado (paquete número 499) corresponde a una respuesta CoAP enviada por el autenticador al EAP Peer de tipo ACK (Acknowledgement) para confirmar la recepción del paquete de solicitud anterior. Este paquete de respuesta consta de las siguientes opciones:

- **Location-Path:** En el caso de este mensaje, la URI del recurso que ha sido creado como resultado de una solicitud POST exitosa consta de tres segmentos, cada uno de los cuales se representa como una opción independiente de tipo Location-Path. El cliente concatenará esas opciones en el orden recibido para formar la URI completa (‘auth/eap/63’), que sirve para indicar a qué recurso debe enviar el autenticador la petición con el tercer mensaje del método EAP-PSK (EAP-PSK-3).
- **Content-Format:** application/octet-stream, indicando que el contenido del payload se presenta como un flujo de bytes.

El código de esta respuesta es 2.01 Created para confirmar que la solicitud es válida y que el recurso solicitado fue efectivamente creado o configurado en la URI indicada por las opciones de Location-Path (‘auth/eap/63’ en este caso). Es decir, el mensaje de respuesta 2.01 Created asegura al cliente (EAP Authenticator) que ahora existe un recurso en la URI especificada (/auth/eap/63) y que puede acceder o interactuar con él en el futuro usando esa ubicación.

La carga útil (payload) de esta respuesta, mostrada en hexadecimal en el campo Data, tiene un tamaño de 60 bytes y su contenido es "02c7003c2f4041e5a7769135685f09a50b20f48e09750e637a06755561c0d71a65fa982305ee736bdd019a395597e464c7731d16e2f0636c69656e74", que representa el segundo mensaje del método EAP-PSK (EAP-PSK-2) construido por el EAP Peer.

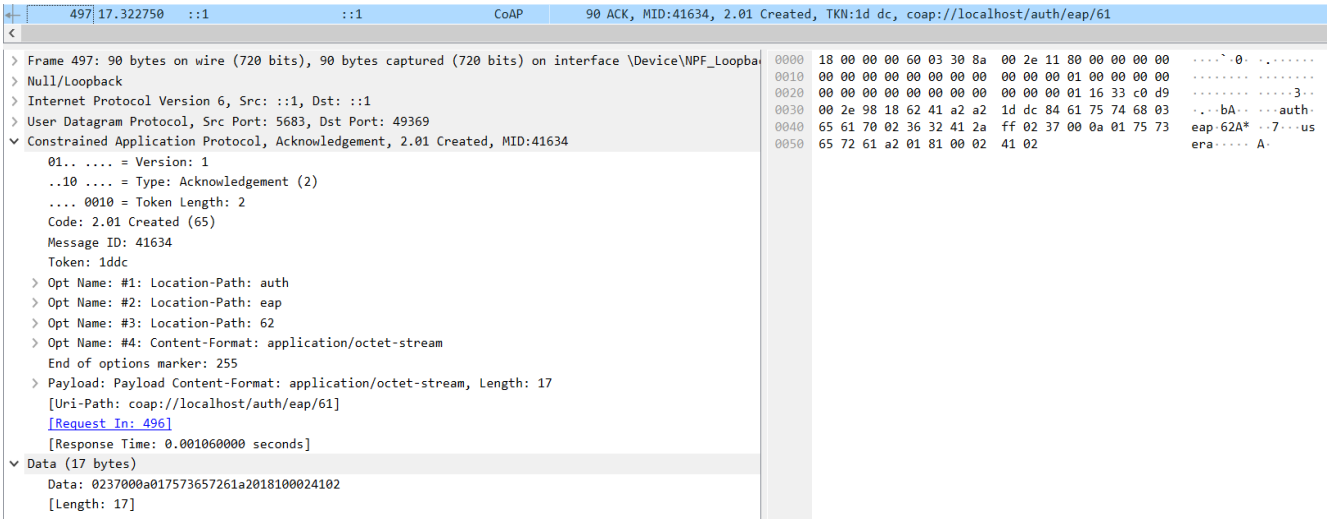


Figura 7.15.- Contenido del décimo paquete de la autenticación exitosa ideal visto en Wireshark

Se puede comprobar que el payload (mensaje EAP-PSK-2) de este décimo paquete es correcto revisando la consola del script peer.py. Además, en esta consola también se pueden ver los valores de los campos obtenidos de parsear el primer mensaje EAP-PSK, el valor de las claves AK y KDK y algunos de los campos de este segundo mensaje EAP-PSK construido por el EAP Peer:

Figura 7.16.- Información del décimo paquete visto en la consola del script peer.py

El undécimo paquete capturado (paquete número 487) se trata de un Access-Request enviado desde el cliente RADIUS hacia el servidor RADIUS que posee los siguientes atributos:

- **User-Name:** usera
- **NAS-IP-Address:** 127.0.0.1
- **Framed-MTU:** 1400
- **Calling-Station-Id:** 00-00-00-00-00-00
- **NAS-Port-Type:** Wireless-802.11
- **Connect-Info:** CON
- **EAP-Message:** En este caso, el contenido es "02c7003c2f4041e5a7769135685f09a50b20f48e09750e637a06755561c0d71a65fa982305ee736bdd019a395597e464c7731d16e2f0636c69656e74", que representa el segundo mensaje del método EAP-PSK (EAP-PSK-2). Los campos de este mensaje detallados son:
  - Code: 2 porque es una respuesta.
  - ID: 199, para asociar este mensaje de respuesta al de solicitud del Access-Challenge anterior.
  - Length: 60 (longitud del mensaje EAP-PSK-2).
  - Type: Pre-Shared Key EAP (EAP-PSK) (47)

- EAP-PSK Flags: 0x40. Configuración de los Flags específicos del segundo mensaje EAP-PSK.
- EAP-PSK RAND_S: "41e5a7769135685f09a50b20f48e0975". Valor aleatorio generado por el servidor (RAND_S), usado en el protocolo EAP-PSK para evitar ataques de repetición y para calcular la MAC_P.
- EAP-PSK RAND_P: "0e637a06755561c0d71a65fa982305ee". Valor aleatorio generado por el cliente (RAND_P), usado en el protocolo EAP-PSK para evitar ataques de repetición, para derivar las claves de sesión y para calcular la MAC_S y la MAC_P.
- EAP-PSK MAC_P: "736bdd019a395597e464c7731d16e2f0". Es un código de autenticación de mensajes generado por el cliente para probar su identidad y asegurar la integridad del mensaje. Es una firma que el cliente incluye en su respuesta para que el servidor pueda verificar la autenticidad de los datos recibidos.
- EAP-PSK ID_P: Identificador del cliente (ID_P), en este caso establecido como "client".

- **State:** El valor de este atributo es el mismo que el del atributo State del Access-Challenge anterior ("7b2e5ae09a97c0d503b47577a2f31792").
- **Message-Authenticator:** Su valor en este mensaje es "e20644e02d6495f44babd3d83d096a56".

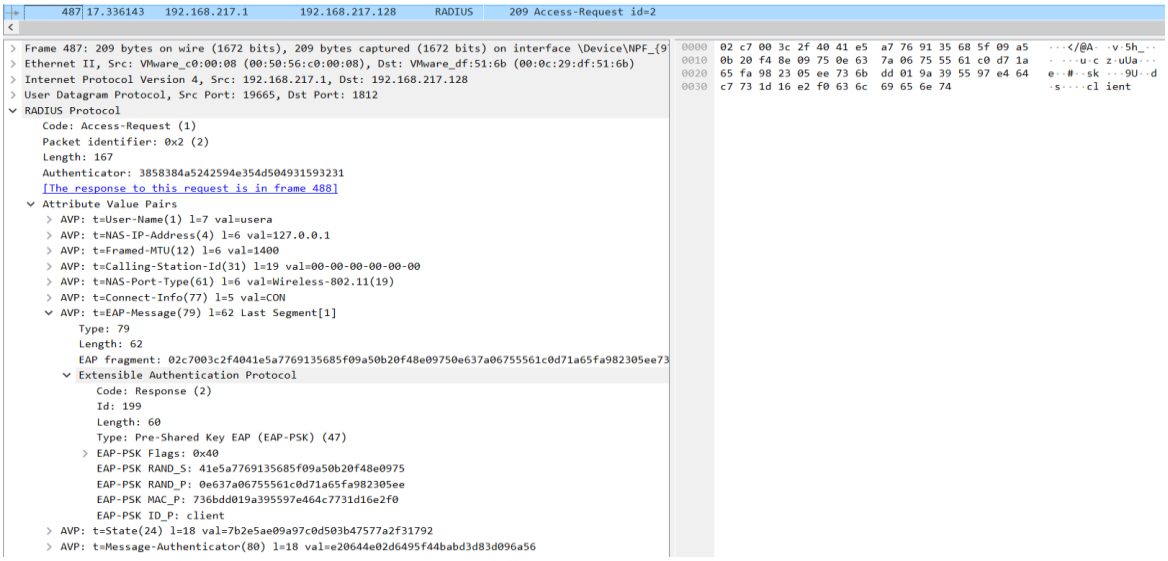


Figura 7.17.- Contenido del undécimo paquete de la autenticación exitosa ideal visto en Wireshark

El duodécimo mensaje capturado (paquete número 488) es un Access-Challenge enviado por el servidor RADIUS al cliente RADIUS del EAP Authenticator en respuesta al Access-Request anterior. Este Access-Challenge tiene los siguientes atributos:

- **EAP-Message:** En este caso, el contenido es "01c8003b2f8041e5a7769135685f09a50b20f48e09753c681b446358f919920afbf1c886bfdf000000000159dc5b738d957e242ba2e71991caa3fe", que representa el tercer mensaje del método EAP-PSK (EAP-PSK-3). Los campos de este mensaje detallados son:
  - Code: 1 porque es una solicitud.
  - ID: 200
  - Length: 59
  - Type: Pre-Shared Key EAP (EAP-PSK) (47)
  - EAP-PSK Flags: 0x80. Configuración de los Flags específicos del tercer mensaje EAP-PSK.
  - EAP-PSK RAND_S: "41e5a7769135685f09a50b20f48e0975". Valor aleatorio generado por el servidor (RAND_S), usado en el protocolo EAP-PSK para evitar ataques de repetición y para derivar claves.
  - EAP-PSK MAC_S: "3c681b446358f919920afbf1c886bfdf". Es un código de autenticación de mensajes generado por el servidor para validar su identidad ante el cliente y garantizar la integridad del mensaje de respuesta.
  - EAP-PSK Protected Channel (encrypted): "000000000159dc5b738d957e242ba2e71991caa3fe". Este campo contribuye a la seguridad y a la robustez de EAP-PSK al proporcionar un canal seguro dentro del protocolo por el que el servidor envía al cliente información sensible cifrada.
- **State:** Al tratarse de un nuevo Access-Challenge el valor de este atributo cambia. En este caso su valor es "3baa97973bcb9a2a61aadff782b50850".
- **Message-Authenticator:** Su valor en este mensaje es "4b6a08b620b518fd4d077817e5ad1d8e".

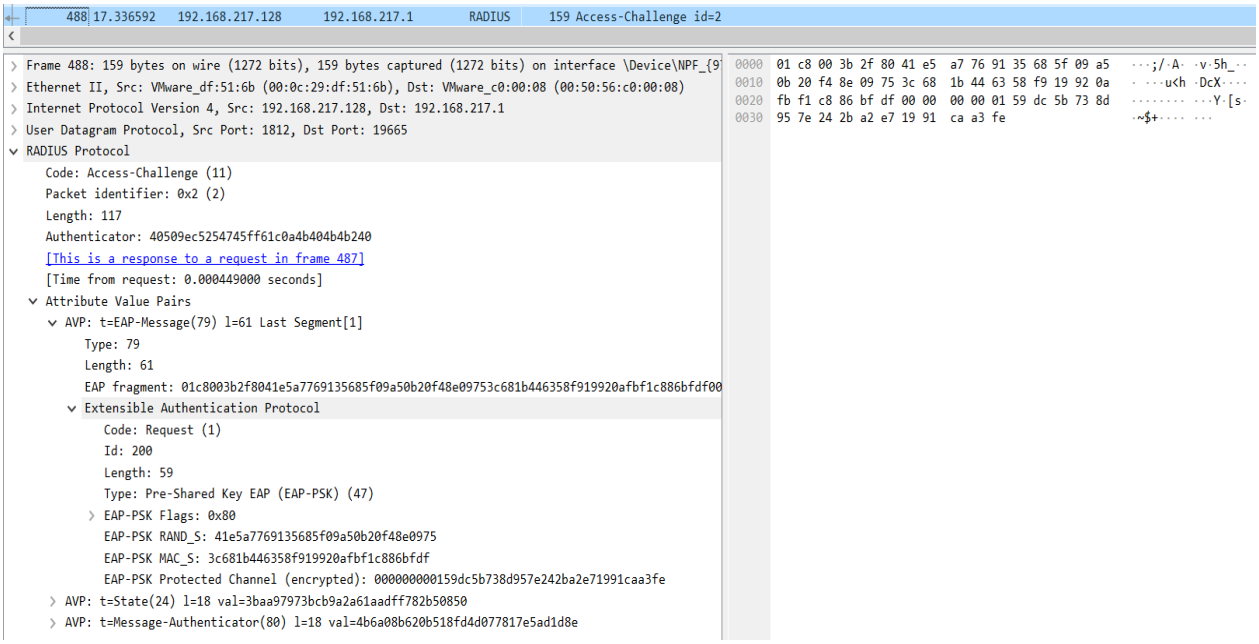


Figura 7.18.- Contenido del duodécimo paquete de la autenticación exitosa ideal visto en Wireshark

El decimotercer paquete capturado (paquete número 500) se trata de una solicitud CoAP de tipo CON (Confirmable) con código POST, enviada desde el EAP Authenticator hacia el EAP Peer. Este paquete consta de las siguientes opciones:

- **Uri-Host:** En este mensaje, la opción Uri-Host tiene el valor localhost. Esto significa que el cliente CoAP se está dirigiendo a un servidor identificado como localhost, que hace referencia a la propia máquina en la que está ejecutándose el cliente, es decir, la dirección de loopback.
- **Uri-Path:** La URI del recurso solicitado está compuesta por tres segmentos, cada uno de los cuales se representa como una opción independiente de tipo URI-Path. Estas opciones juntas forman la URI completa del recurso solicitado (‘auth/eap/63’). Esta URI es el contenido de la opción Location-Path del noveno paquete (solicitud CoAP cuyo payload es el mensaje EAP-PSK-1) por lo que se está cumpliendo con lo establecido en el Draft.
- **Content-Format:** application/octet-stream, indicando que el contenido del payload es un flujo de bytes.

La carga útil (payload) de este paquete, mostrada en hexadecimal en el campo Data, tiene un tamaño de 59 bytes y su contenido es "01c8003b2f8041e5a7769135685f09a50b20f48e09753c681b446358f919920afbf1c886bfdf000000000159dc5b738d957e242ba2e71991caa3fe", que representa el tercer mensaje del método EAP-PSK (EAP-PSK-3) recibido del EAP Server.

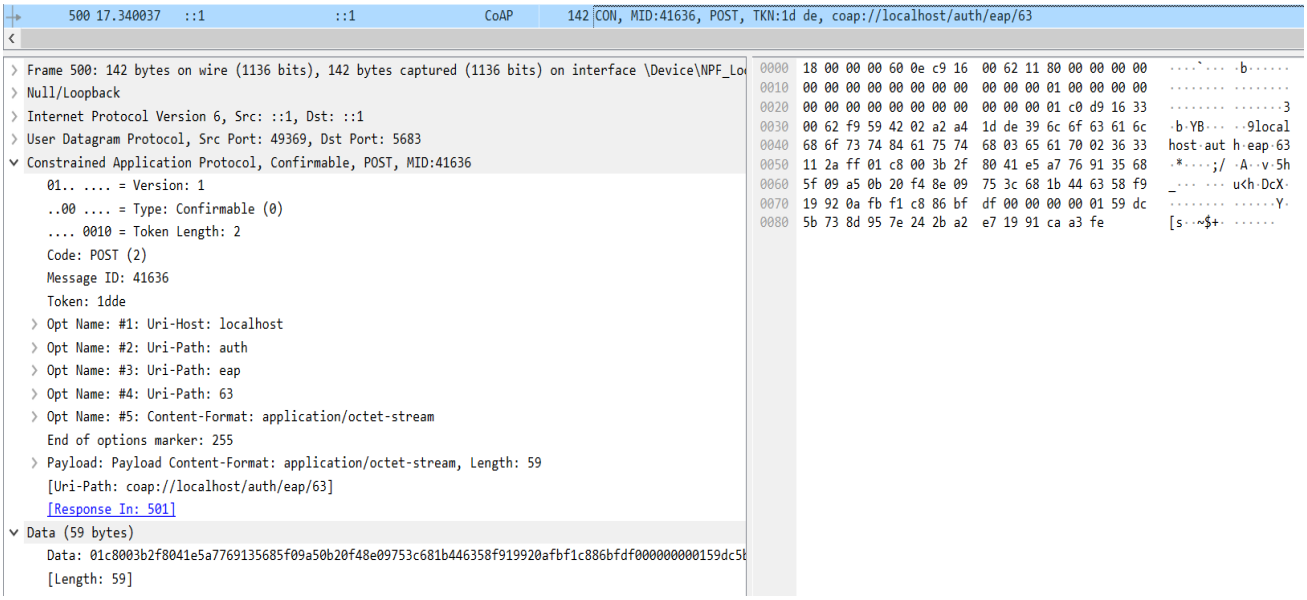


Figura 7.19.- Contenido del decimotercer paquete de la autenticación exitosa ideal visto en Wireshark

Se puede comprobar que la URI del recurso y el payload de este decimotercer paquete son correctos revisando la consola del script controller.py:

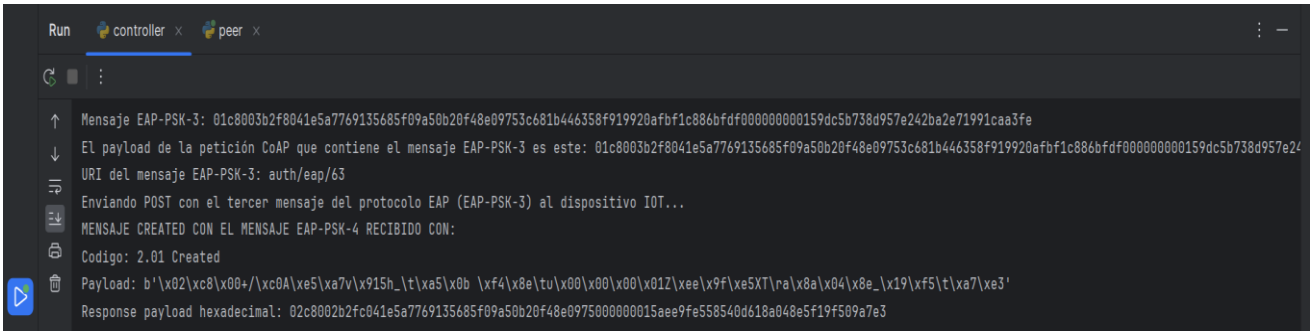


Figura 7.20.- Información del decimotercer paquete visto en la consola del script controller.py

El decimocuarto paquete capturado (paquete número 501) corresponde a una respuesta CoAP enviada por el autenticador al EAP Peer de tipo ACK (Acknowledgement) para confirmar la recepción del paquete de solicitud anterior. Este paquete de respuesta consta de las siguientes opciones:

- **Location-Path:** En el caso de este mensaje, la URI del recurso que ha sido creado como resultado de una solicitud POST exitosa consta de tres segmentos, cada uno de los cuales se representa como una opción independiente de tipo Location-Path. El cliente concatenará esas opciones en el orden recibido para formar la URI completa (‘auth/eap/64’), que sirve para indicar a qué recurso debe enviar el autenticador la petición con el mensaje de éxito (EAP SUCCESS).
- **Content-Format:** application/octet-stream, indicando que el contenido del payload se presenta como un flujo de bytes.

El código de esta respuesta es 2.01 Created para confirmar que la solicitud es válida y que el recurso solicitado fue efectivamente creado o configurado en la URI indicada por las opciones de Location-Path (‘auth/eap/64’ en este caso). Es decir, el mensaje de respuesta 2.01 Created asegura al cliente (EAP Authenticator) que ahora existe un recurso en la URI especificada (/auth/eap/64) y que puede acceder o interactuar con él en el futuro usando esa ubicación.

La carga útil (payload) de esta respuesta, mostrada en hexadecimal en el campo Data, tiene un tamaño de 43 bytes y su contenido es “02c8002b2fc041e5a7769135685f09a50b20f48e0975000000015aee9fe558540d618a048e5f19f509a7e3”, que representa el cuarto mensaje del método EAP-PSK (EAP-PSK-4) construido por el EAP Peer.

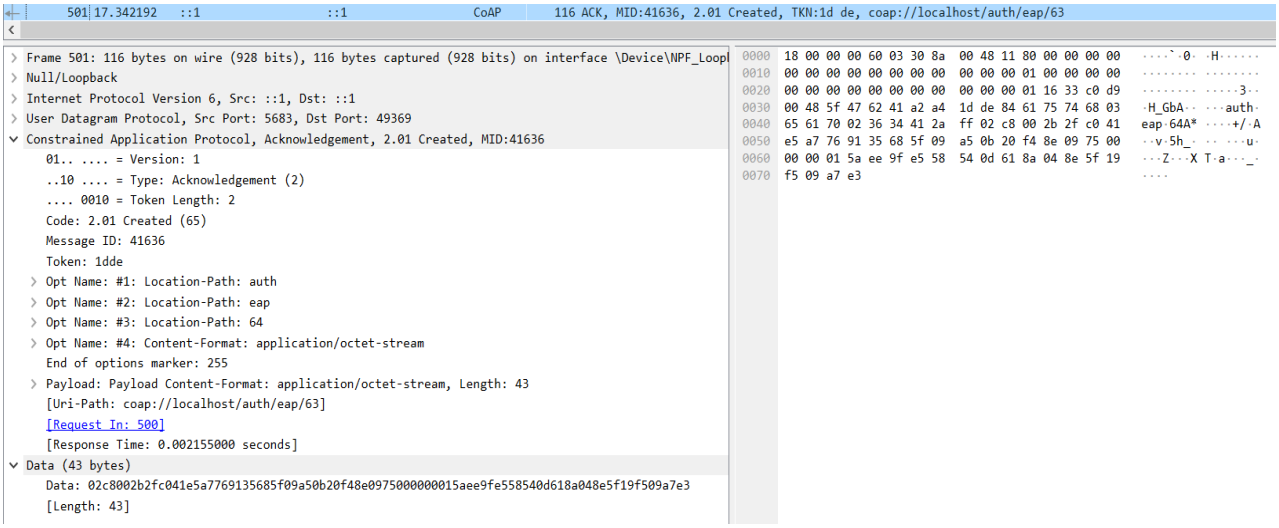


Figura 7.21.- Contenido del decimocuarto paquete de la autenticación exitosa ideal visto en Wireshark

Se puede comprobar que el payload (mensaje EAP-PSK-4) de este décimo paquete es correcto revisando la consola del script peer.py. Además, en esta consola también se pueden ver los valores de los campos obtenidos de parsear el tercer mensaje EAP-PSK, el valor de las claves TEK, MSK y EMSK y algunos de los campos de este cuarto mensaje EAP-PSK construido por el EAP Peer:

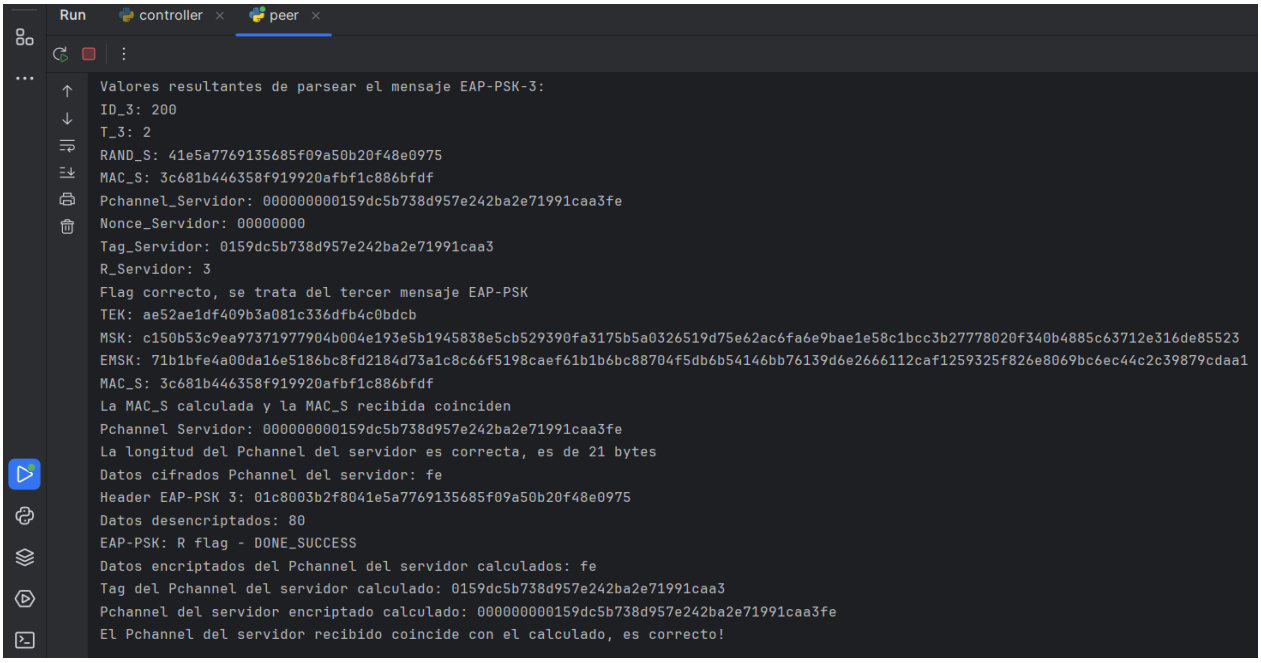


Figura 7.22.- Información del decimocuarto paquete visto en la consola del script peer.py parte 1

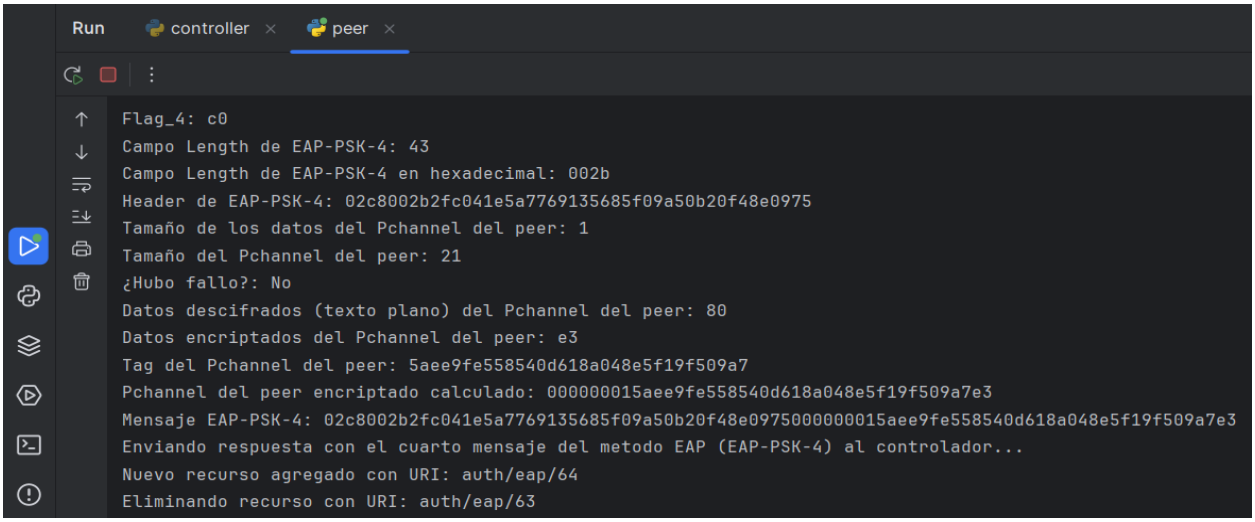


Figura 7.23.- Información del decimocuarto paquete visto en la consola del script peer.py parte 2

El decimoquinto paquete capturado (paquete número 489) se trata de un Access-Request enviado desde el cliente RADIUS hacia el servidor RADIUS que posee los siguientes atributos:

- **User-Name:** usera
- **NAS-IP-Address:** 127.0.0.1
- **Framed-MTU:** 1400
- **Calling-Station-Id:** 00-00-00-00-00-00
- **NAS-Port-Type:** Wireless-802.11
- **Connect-Info:** CON
- **EAP-Message:** En este caso, el contenido es "02c8002b2fc041e5a7769135685f09a50b20f48e0975000000015aee9fe558540d618a048e5f19f509a7e3", que representa el cuarto mensaje del método EAP-PSK (EAP-PSK-4). Los campos de este mensaje detallados son:
  - Code: 2 porque es una respuesta.
  - ID: 200, para asociar este mensaje de respuesta al de solicitud del Access-Challenge anterior.
  - Length: 43 (longitud del mensaje EAP-PSK-3).
  - Type: Pre-Shared Key EAP (EAP-PSK) (47)

- EAP-PSK Flags: 0xc0. Configuración de los Flags específicos del cuarto mensaje EAP-PSK.
- EAP-PSK RAND_S: "41e5a7769135685f09a50b20f48e0975". Valor aleatorio generado por el servidor (RAND_S), usado en el protocolo EAP-PSK para evitar ataques de repetición y para calcular la MAC_P.
- EAP-PSK Protected Channel (encrypted): "000000015aee9fe558540d618a048e5f19f509a7e3". Este campo contribuye a la seguridad y a la robustez de EAP-PSK al proporcionar un canal seguro dentro del protocolo por el que el cliente envía al servidor información sensible cifrada.

- **State:** El valor de este atributo es el mismo que el del atributo State del Access-Challenge anterior ("3baa97973bcb9a2a61aadff782b50850").
- **Message-Authenticator:** Su valor en este mensaje es "c07c0f07ae8a00c675a407ea0a223b93".

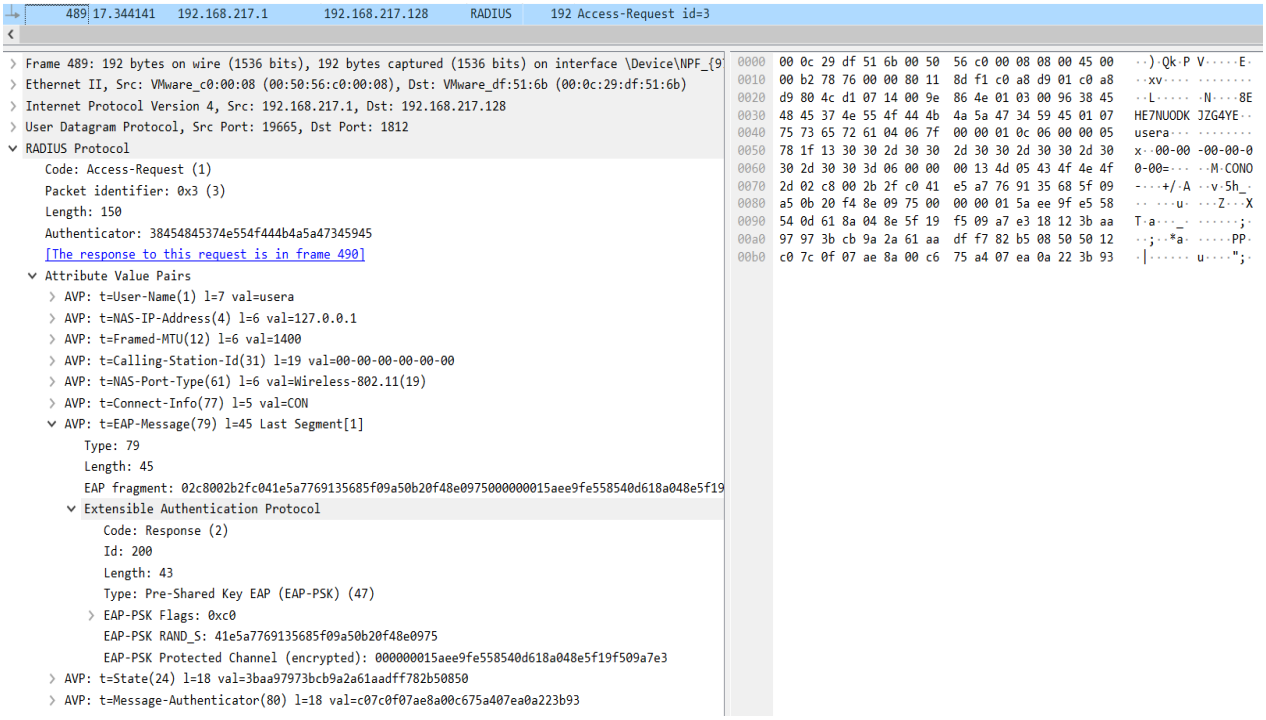


Figura 7.24.- Contenido del decimoquinto paquete de la autenticación exitosa ideal visto en Wireshark

El decimosexto mensaje capturado (paquete número 490) es un Access-Accept enviado por el servidor RADIUS al cliente RADIUS del EAP Authenticator en respuesta al

Access-Request anterior. El envío de un Access-Accept significa que el servidor ha aceptado la solicitud de acceso del cliente tras haber pasado todas las verificaciones de autenticación (se han intercambiado con éxito todos los mensajes del método EAP-PSK). Este paquete tiene los siguientes atributos:

- **Vendor-Specific:** Contiene atributos específicos de Microsoft (ID de proveedor 311). El primer atributo Vendor-Specific contiene la clave MS-MPPE-Recv-Key cifrada y el segundo atributo Vendor-Specific contiene la clave MS-MPPE-Send-Key cifrada. Estas dos claves son muy importantes ya que, si el EAP Authenticator consigue descifrarlas y concatenarlas, obtendrá la MSK necesaria para crear su contexto de seguridad OSCORE y así poder proteger el mensaje de petición CoAP que contiene el mensaje EAP SUCCESS.
- **EAP-Message:** En este caso, el contenido es "03c80004", que representa el mensaje de éxito (EAP SUCCESS). Los campos de este mensaje detallados son:
    - Code: 3. Se trata del código de éxito (Success) para confirmar la finalización exitosa del método EAP-PSK.
    - ID: 200. El mismo que el del mensaje del Access-Request anterior. Al mantener el ID, el servidor trata estos mensajes finales como parte de una respuesta conjunta a un desafío, indicando que el proceso de autenticación está completo y alineado.
    - Length: 4
    - User-Name: usera
- **Message-Authenticator:** Su valor en este mensaje es "97d6ef338ffd057a490b107aa133cedb".

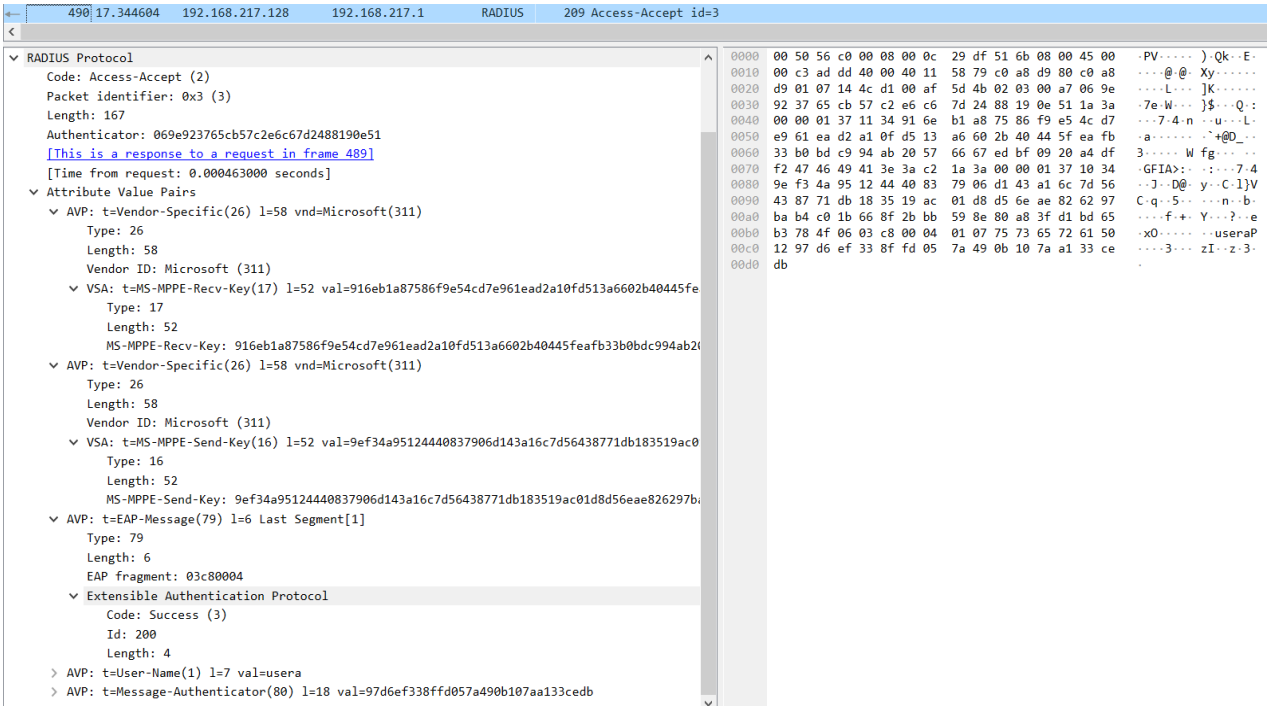


Figura 7.25.- Contenido del decimosexto paquete de la autenticación exitosa ideal visto en Wireshark

El EAP Authenticator, tras recibir este último paquete, tiene que hacer dos cosas antes de construir y proteger con OSCORE el próximo paquete de solicitud que enviará al EAP Peer.

La primera cosa será definir un payload constituido por el mensaje EAP SUCCESS obtenido del paquete Access-Accept previo (“03c80004”) y por una estructura CBOR (“a101197080”) que contiene el tiempo de vida de la sesión (session lifetime). En la consola del script Controller.py se puede ver este payload definido, las claves MPPE cifradas y la URI que va a tener la próxima solicitud CoAP:

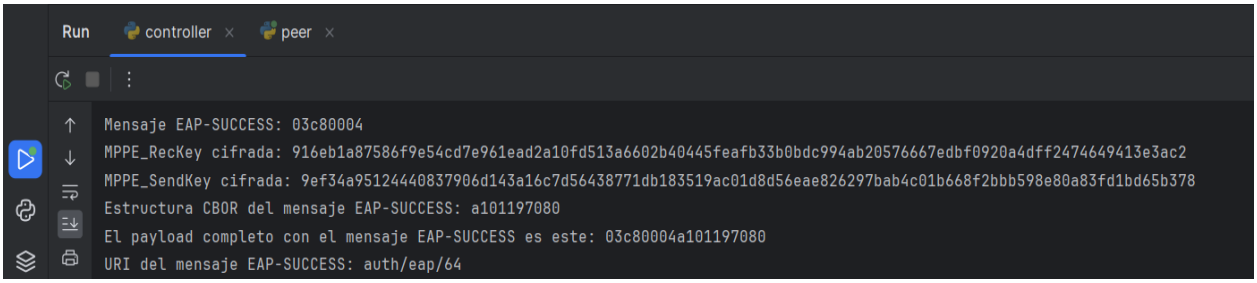


Figura 7.26.- Payload definido con el mensaje EAP SUCCESS y la estructura CBOR el tiempo de vida de la sesión visto desde la consola del script controller.py

La segunda cosa será crear el contexto de seguridad OSCORE del EAP Authenticator, el cual está compuesto por un Sender Context, un Recipient Context y un Common Context. Cada uno de los campos que componen estos contextos se pueden visualizar desde la consola del script controller.py:

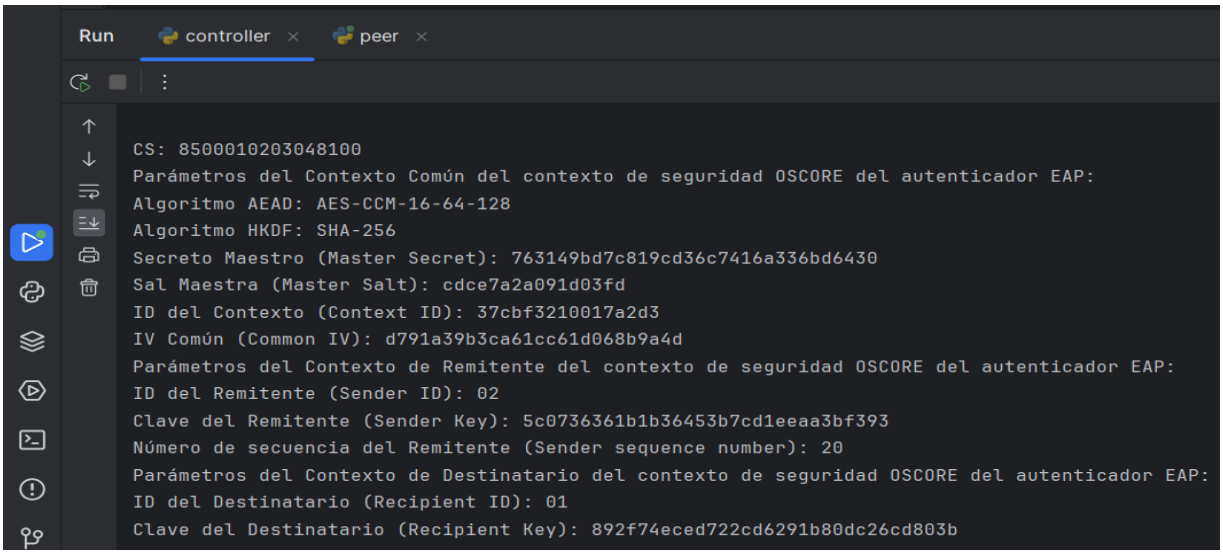


Figura 7.27.- Campos del contexto de seguridad OSCORE del EAP Authenticator visto desde la consola del script controller.py

Una vez hechas estas dos cosas, el EAP Authenticator envía al EAP Peer una solicitud CoAP de tipo CON (Confirmable) con código POST protegida por OSCORE. Se trata del decimoséptimo paquete capturado (paquete número 502) que consta de las siguientes opciones:

- **Uri-Host:** En este mensaje, la opción Uri-Host tiene el valor localhost. Esto significa que el cliente CoAP se está dirigiendo a un servidor identificado como localhost, que hace referencia a la propia máquina en la que está ejecutándose el cliente, es decir, la dirección de loopback.
- **Uri-Path:** La URI del recurso solicitado está compuesta por tres segmentos, cada uno de los cuales se representa como una opción independiente de tipo URI-Path. Estas opciones juntas forman la URI completa del recurso solicitado (‘auth/eap/64’). Esta URI es el contenido de la opción Location-Path del noveno paquete (solicitud CoAP cuyo payload es el mensaje EAP-PSK-3) por lo que se está cumpliendo con lo establecido en el Draft.

- **Content-Format:** application/octet-stream, indicando que el contenido del payload es un flujo de bytes.

La carga útil de este paquete, mostrada en hexadecimal en el campo Data, tiene un tamaño de 61 bytes y su contenido será toda esta petición CoAP (Code, Token, Message ID, Options y el payload definido anteriormente compuesto por el mensaje EAP SUCCESS y la estructura CBOR con el tiempo de vida de la sesión) protegida por OSCORE.

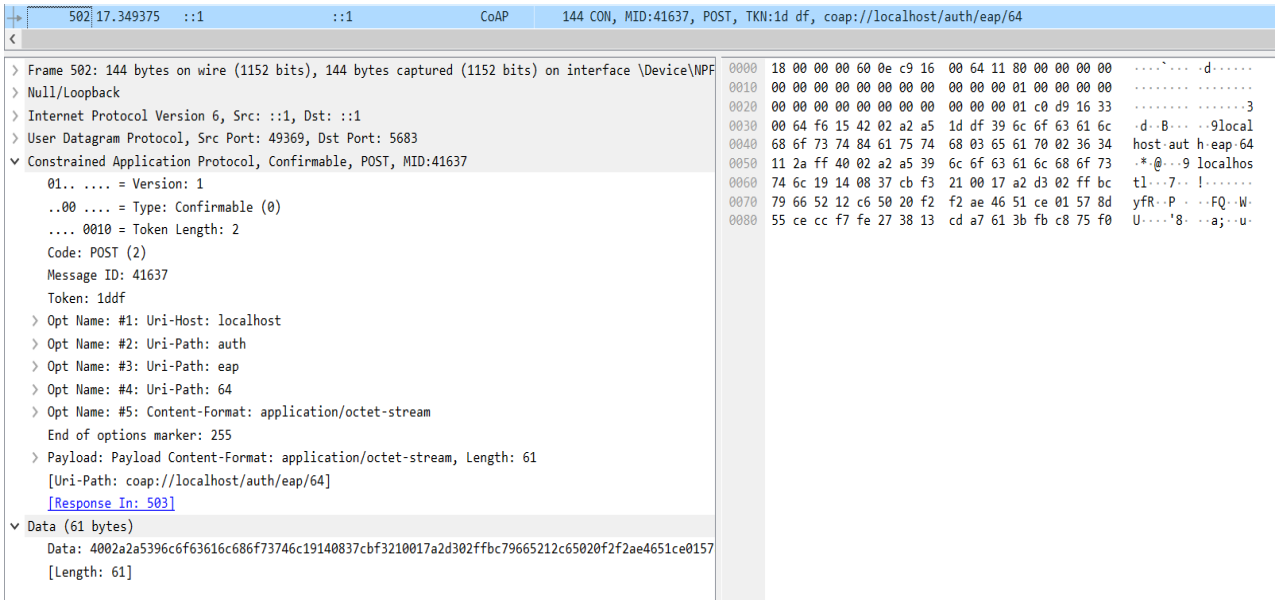


Figura 7.28.- Contenido del decimoséptimo paquete de la autenticación exitosa ideal visto en Wireshark

El EAP Peer, una vez que recibe esta petición, deberá crear su contexto de seguridad OSCORE, el cual está compuesto por un Sender Context, un Recipient Context y un Common Context. Cada uno de los campos que componen estos contextos se pueden visualizar desde la consola del script peer.py:

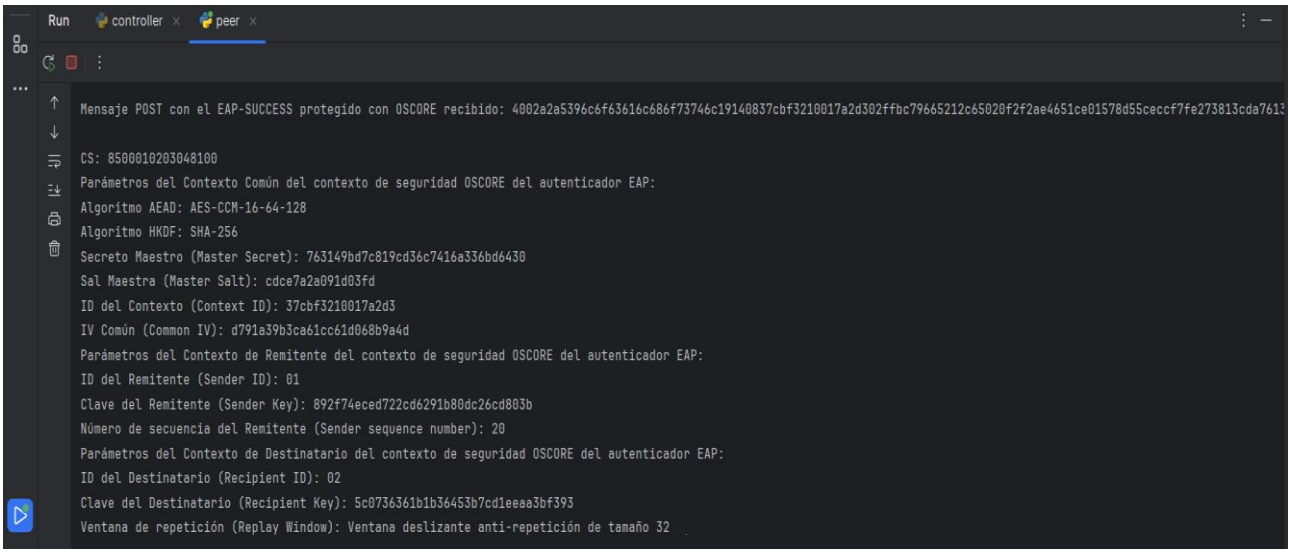


Figura 7.29.- Campos del contexto de seguridad OSCORE del EAP Peer visto desde la consola del script controller.py

Tras esto, el EAP Peer desprotegerá la carga útil de la petición recibida, la procesará y así podrá obtener la información más relevante (código y payload), así como el tiempo de vida de la sesión. Toda esta información se puede ver en la consola del script peer.py:

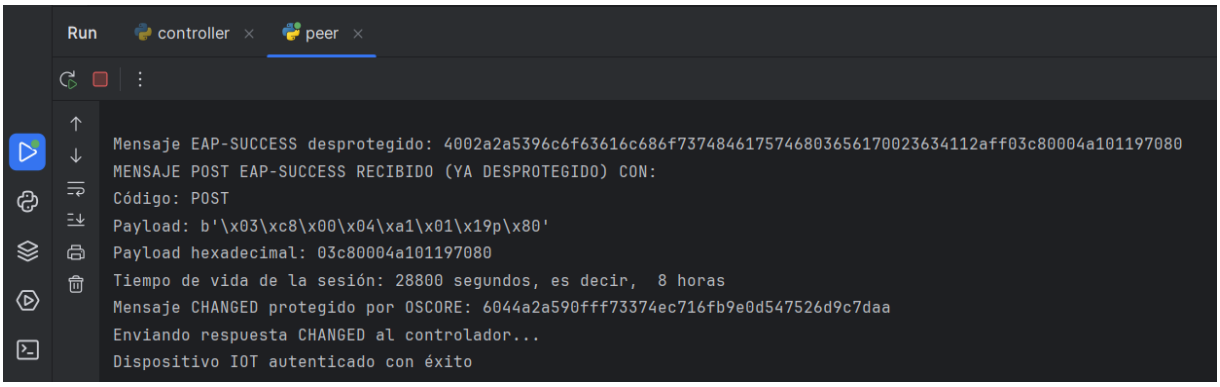


Figura 7.30.- Información de la solicitud con el mensaje EAP SUCCESS desprotegido vista en la consola del script peer.py

Finalmente, el EAP Peer envía al EAP Authenticator una respuesta CoAP protegida por OSCORE de tipo ACK (Acknowledgement) para confirmar la recepción del paquete de solicitud anterior. Este paquete corresponde al decimoctavo paquete capturado (paquete número 503), el cual consta únicamente de la opción Content-Format que será application/octet-stream para indicar que la carga útil contiene un flujo de bytes.

El código de esta respuesta es 2.04 Changed para indicar que la autenticación fue completada y que el recurso o la sesión del cliente ha sido establecida. En cuanto a la carga útil (payload) de esta respuesta, mostrada en hexadecimal en el campo Data, tiene un tamaño de 21 bytes y su contenido será toda la respuesta CHANGED (Code, Token, Message ID, opción Content-Format y el payload "FIN") protegida por OSCORE.

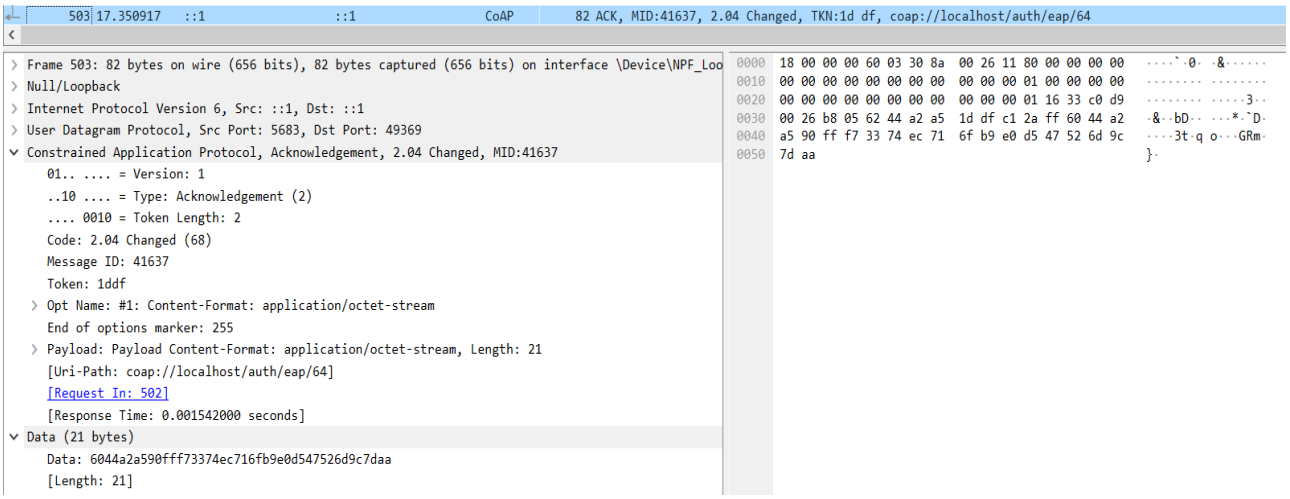


Figura 7.31.- Contenido del decimoctavo paquete de la autenticación exitosa ideal visto en Wireshark

Con esta respuesta el EAP Authenticator sabrá que el proceso de autenticación se ha completado correctamente y que, por tanto, el dispositivo IoT ha pasado a formar parte del dominio de autenticación, lo que significa que ya es reconocido como confiable o autorizado y por consiguiente, también ha comenzado a pertenecer al dominio de seguridad, ya que al estar autenticado puede acceder a recursos protegidos o canales seguros. Además, se demuestra que la implementación responde adecuadamente, conforme a lo establecido en el Draft, ante la autenticación exitosa de un cliente (dispositivo IoT).

7.2.2.- Autenticación exitosa ideal con petición duplicada o retransmisión tardía

Este escenario simula una condición en la que se produce una retransmisión tardía o una duplicación de una solicitud durante el proceso de autenticación. En particular, se reproduce el caso en el que el EAP Authenticator reenvía, tras un breve periodo de tiempo, la solicitud CoAP que contiene en su carga útil el primer mensaje EAP-PSK, lo que permite observar el comportamiento del sistema frente a duplicaciones de mensaje.

En primer lugar, se presenta un diagrama MSC que detalla el flujo teórico de mensajes esperados en caso de que ocurra una duplicación. En este flujo, la retransmisión de la solicitud por parte del EAP Authenticator no debería interrumpir ni afectar la autenticación, dado que el EAP Peer, al recibir la solicitud duplicada, ya ha eliminado el recurso correspondiente a la solicitud original. Esto sucede porque, una vez que el EAP Peer devuelve la respuesta y asigna un nuevo recurso, el recurso previo es eliminado. Este diagrama permite visualizar claramente cada uno de los paquetes que deben intercambiarse, incluyendo los códigos específicos e identificar cómo deben gestionarse estos mensajes duplicados sin alterar el flujo de autenticación.

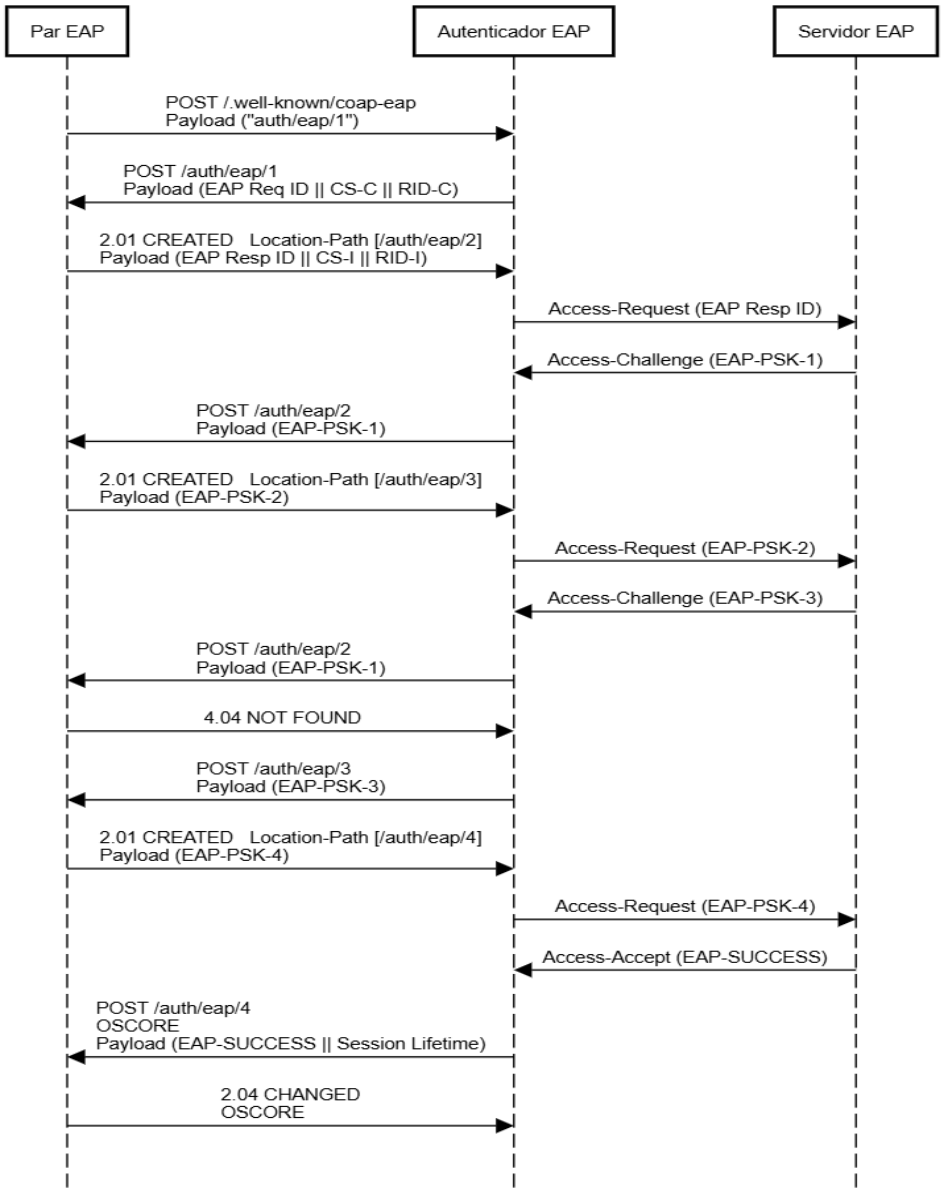


Figura 7.32.- Diagrama MSC del escenario de autenticación exitosa con petición duplicada

A continuación, se muestra la captura de tráfico obtenida en Wireshark al ejecutar este escenario con la implementación explicada en la sección 6.3. A partir de esta captura, se comentarán los paquetes reales observados en comparación con el flujo teórico del diagrama MSC. Se comprobará que cada mensaje esperado esté presente en la captura, que cada uno cuente con los códigos y datos correctos según el estándar, confirmando así que a pesar de la duplicación, la implementación sigue el flujo adecuado para una autenticación exitosa.

coap or radius

No.	Time	Source	Destination	Protocol	Length	Info
183	6.144207	192.168.0.3	192.168.0.3	CoAP	76	CON, MID:4391, POST, TKN:24 47, /.well-known/coap-eap
184	6.145198	192.168.0.3	192.168.0.3	CoAP	85	ACK, MID:4391, 2.31 Continue, TKN:24 47, /.well-known/coap-eap
367	11.895660	::1	::1	CoAP	103	CON, MID:27509, POST, TKN:3e a9, coap://localhost/auth/eap/89
368	11.896554	::1	::1	CoAP	90	ACK, MID:27509, 2.01 Created, TKN:3e a9, coap://localhost/auth/eap/89
355	11.899400	192.168.217.1	192.168.217.128	RADIUS	141	Access-Request id=0
356	11.899764	192.168.217.128	192.168.217.1	RADIUS	105	Access-Challenge id=0
357	11.902283	192.168.217.1	192.168.217.128	RADIUS	159	Access-Request id=1
358	11.902862	192.168.217.128	192.168.217.1	RADIUS	129	Access-Challenge id=1
369	11.904245	::1	::1	CoAP	112	CON, MID:27510, POST, TKN:3e aa, coap://localhost/auth/eap/90
370	11.905402	::1	::1	CoAP	133	ACK, MID:27510, 2.01 Created, TKN:3e aa, coap://localhost/auth/eap/90
359	11.908648	192.168.217.1	192.168.217.128	RADIUS	209	Access-Request id=2
360	11.909059	192.168.217.128	192.168.217.1	RADIUS	159	Access-Challenge id=2
371	11.910449	::1	::1	CoAP	112	CON, MID:27511, POST, TKN:3e ab, coap://localhost/auth/eap/90
372	11.910896	::1	::1	CoAP	58	ACK, MID:27511, 4.04 Not Found, TKN:3e ab, coap://localhost/auth/eap/90
373	11.911155	::1	::1	CoAP	142	CON, MID:27512, POST, TKN:3e ac, coap://localhost/auth/eap/91
374	11.913031	::1	::1	CoAP	116	ACK, MID:27512, 2.01 Created, TKN:3e ac, coap://localhost/auth/eap/91
361	11.914562	192.168.217.1	192.168.217.128	RADIUS	192	Access-Request id=3
362	11.915320	192.168.217.128	192.168.217.1	RADIUS	209	Access-Accept id=3
375	11.918529	::1	::1	CoAP	144	CON, MID:27513, POST, TKN:3e ad, coap://localhost/auth/eap/92
376	11.920064	::1	::1	CoAP	82	ACK, MID:27513, 2.04 Changed, TKN:3e ad, coap://localhost/auth/eap/92

Figura 7.33.- Captura Wireshark del escenario de autenticación exitosa con petición duplicada

Como se puede observar en la captura de Wireshark, todos los paquetes intercambiados son idénticos a los del escenario anterior hasta que el EAP Authenticator, tras recibir el Access-Challenge del servidor con el mensaje EAP-PSK-3 (paquete número 360), envía de nuevo al EAP Peer un paquete de solicitud CoAP dirigido al recurso identificado con la URI ‘auth/eap/90’. Este paquete es una réplica de la petición anterior (paquete número 369) en cuanto a contenido puesto que su payload es el mensaje EAP-PSK-1 pero se le asigna un nuevo ID (MID: 27511).

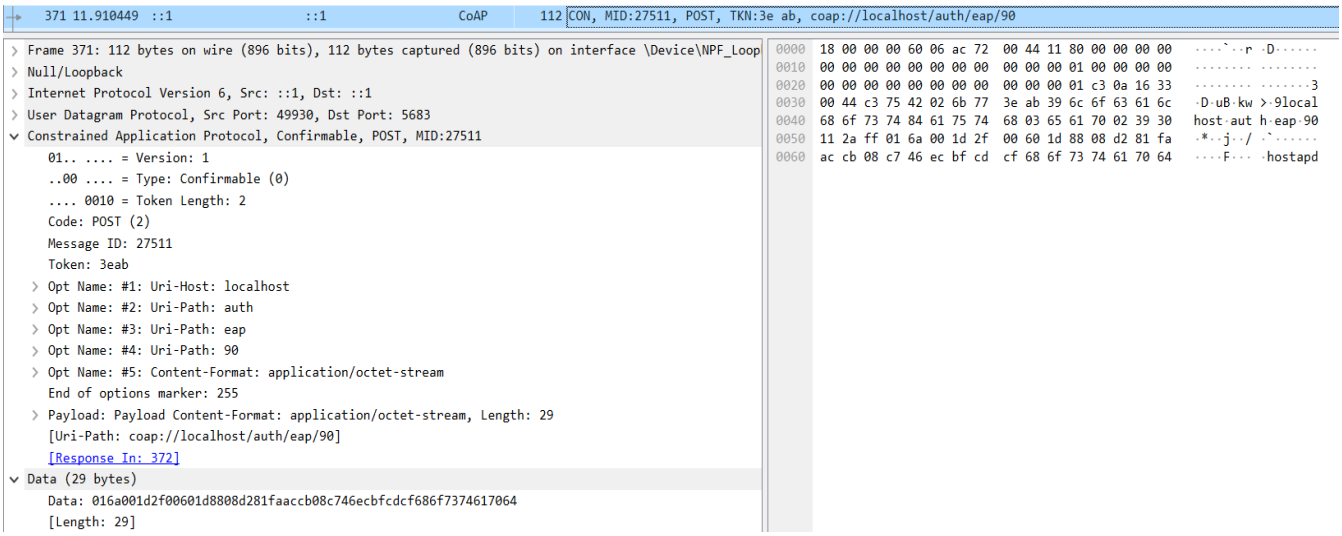


Figura 7.34.- Contenido del paquete duplicado visto en Wireshark

El EAP Peer responde a esta petición duplicada con una respuesta con código 4.04 NOT FOUND, que no contiene ninguna carga útil, cuyo ID (27511) será el mismo que el de la petición para asociarla a ella y que indica que el recurso solicitado ya no existe en el servidor de recursos del EAP Peer puesto que ya lo ha eliminado.

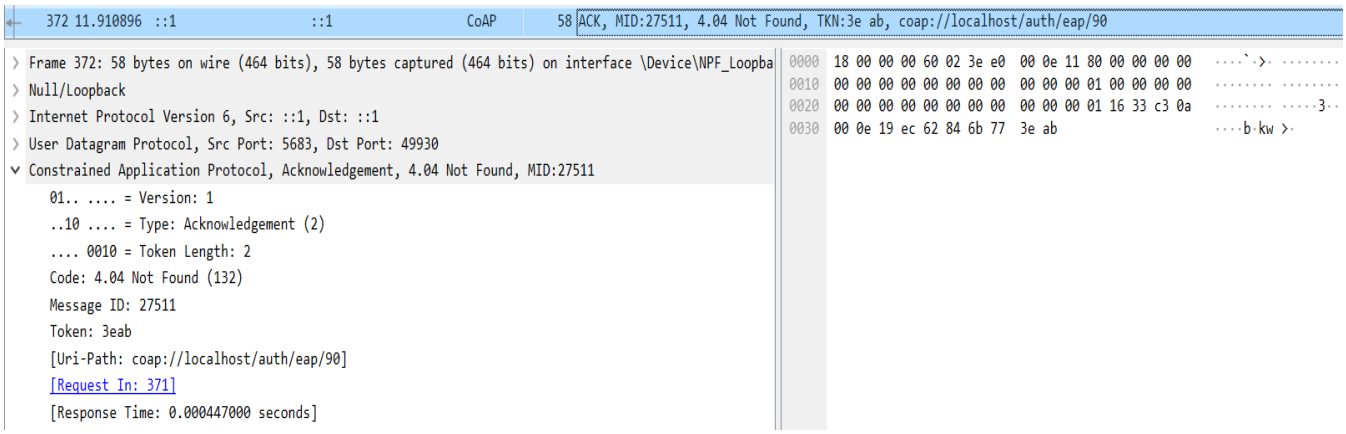


Figura 7.35.- Contenido del paquete de respuesta NOT FOUND visto en Wireshark

Tras esta respuesta, el resto de los paquetes intercambiados vuelven a ser idénticos a los del escenario anterior, de modo que se demuestra que a pesar del mensaje duplicado, el flujo de operación continúa con normalidad hasta que la autenticación se completa con éxito y que la implementación responde correctamente, conforme a lo establecido en el Draft, para las peticiones duplicadas.

7.2.3.- Autenticación fallida

En este escenario se analizará el caso en que el proceso de autenticación falla debido a que el EAP Peer no logra desproteger el mensaje de éxito (EAP Success) protegido con OSCORE. Este fallo ocurre cuando la Master Session Key (MSK) obtenida en ambas entidades no coincide o cuando algún campo necesario para la creación del contexto de seguridad de OSCORE, como el Context ID o el Recipient ID, está mal definido en alguna de las entidades. Para esta prueba, se provocará intencionalmente un error al definir un Context ID distinto en cada entidad y así observar cómo responde el sistema ante esta condición.

En primer lugar, se presenta un diagrama MSC que detalla el flujo teórico esperado cuando se produce un fallo de autenticación debido a problemas de desprotección en el mensaje EAP Success. Este diagrama ayuda a visualizar claramente cada uno de los paquetes que deben intercambiarse, incluyendo los códigos específicos y también ayudará a ver cómo, tras la detección del fallo al desproteger el mensaje, el sistema termina el flujo de autenticación de manera controlada.

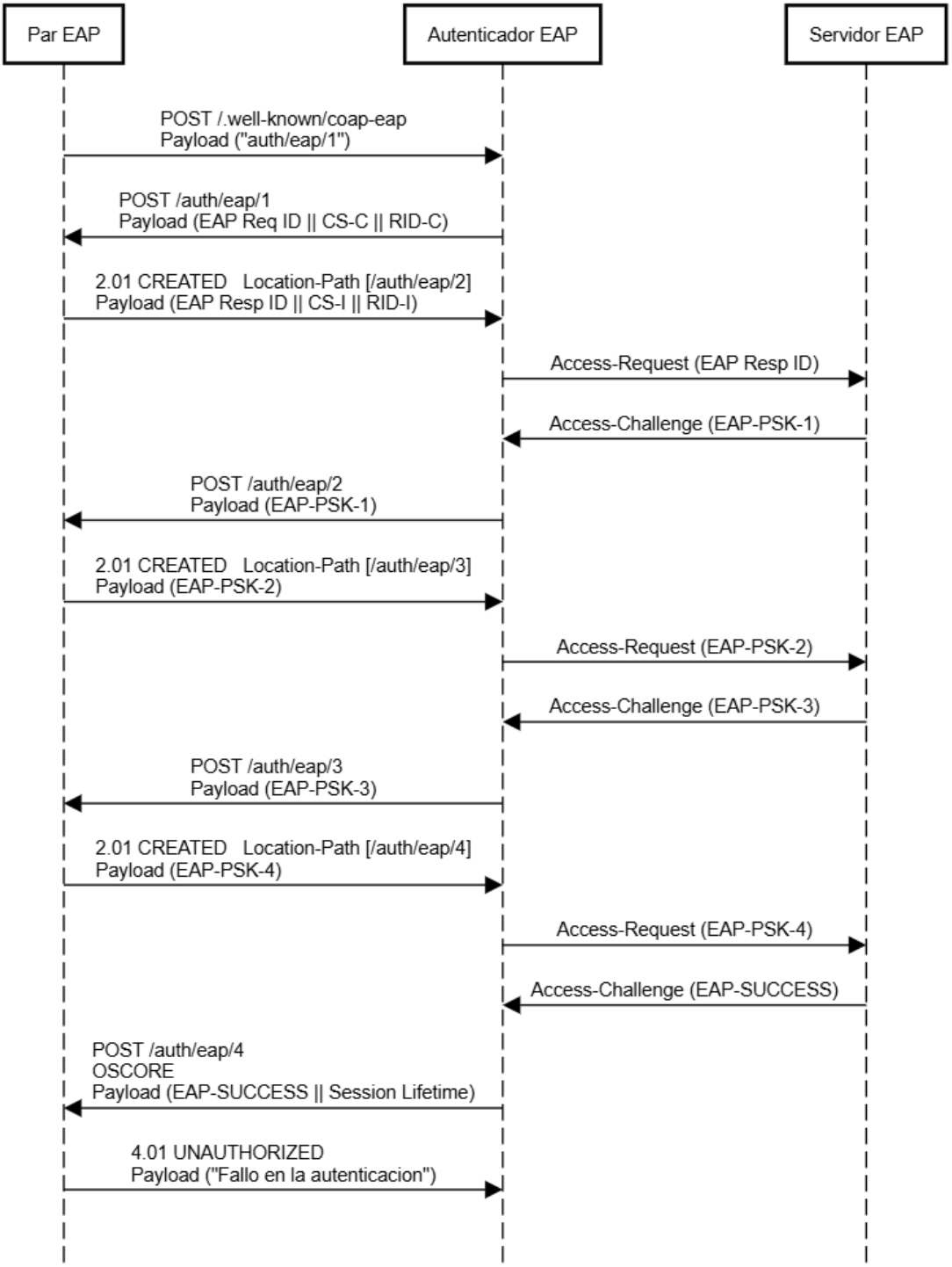


Figura 7.36.- Diagrama MSC del escenario de autenticación fallida

A continuación, se muestra la captura de tráfico obtenida en Wireshark al ejecutar este escenario con la implementación explicada en la sección 6.3. A partir de esta captura, se comentarán los paquetes reales observados en comparación con el flujo teórico del diagrama MSC. Se verificará que, como resultado de la falla de desprotección, el mensaje

EAP Success no se procesa adecuadamente y se comprobará que cada mensaje contenga los códigos y valores que indiquen adecuadamente el estado de fallo, reflejando el comportamiento previsto del sistema cuando no se puede completar la autenticación.

coap or radius

No.	Time	Source	Destination	Protocol	Length	Info
628	16.921050	192.168.0.3	192.168.0.3	CoAP	76	CON, MID:40827, POST, TKN:e8 71, /.well-known/coap-eap
629	16.922077	192.168.0.3	192.168.0.3	CoAP	85	ACK, MID:40827, 2.31 Continue, TKN:e8 71, /.well-known/coap-eap
829	22.933612	::1	::1	CoAP	103	CON, MID:47803, POST, TKN:1c 6b, coap://localhost/auth/eap/24
830	22.934526	::1	::1	CoAP	90	ACK, MID:47803, 2.01 Created, TKN:1c 6b, coap://localhost/auth/eap/24
819	22.937436	192.168.217.1	192.168.217.128	RADIUS	141	Access-Request id=0
820	22.937818	192.168.217.128	192.168.217.1	RADIUS	105	Access-Challenge id=0
821	22.940654	192.168.217.1	192.168.217.128	RADIUS	159	Access-Request id=1
822	22.941255	192.168.217.128	192.168.217.1	RADIUS	129	Access-Challenge id=1
831	22.942601	::1	::1	CoAP	112	CON, MID:47804, POST, TKN:1c 6c, coap://localhost/auth/eap/25
832	22.943821	::1	::1	CoAP	133	ACK, MID:47804, 2.01 Created, TKN:1c 6c, coap://localhost/auth/eap/25
823	22.946990	192.168.217.1	192.168.217.128	RADIUS	209	Access-Request id=2
824	22.947309	192.168.217.128	192.168.217.1	RADIUS	159	Access-Challenge id=2
833	22.948863	::1	::1	CoAP	142	CON, MID:47805, POST, TKN:1c 6d, coap://localhost/auth/eap/26
834	22.950610	::1	::1	CoAP	116	ACK, MID:47805, 2.01 Created, TKN:1c 6d, coap://localhost/auth/eap/26
825	22.951900	192.168.217.1	192.168.217.128	RADIUS	192	Access-Request id=3
826	22.952184	192.168.217.128	192.168.217.1	RADIUS	209	Access-Accept id=3
835	22.957210	::1	::1	CoAP	144	CON, MID:47806, POST, TKN:1c 6e, coap://localhost/auth/eap/27
836	22.958117	::1	::1	CoAP	86	ACK, MID:47806, 4.01 Unauthorized, TKN:1c 6e, coap://localhost/auth/eap/27

Figura 7.37.- Captura Wireshark del escenario de autenticación fallida

Como se puede observar en la captura de Wireshark, todos los paquetes intercambiados son idénticos a los del escenario de autenticación exitosa ideal excepto el último. El EAP Peer recibe la petición CoAP protegida con OCORE que contiene el mensaje EAP SUCCESS (paquete número 835). Este EAP Peer intentará desproteger el mensaje de éxito para procesarlo pero, como el ID de su Contexto OSCORE es distinto al Context ID OSCORE del EAP Authenticator, no podrá hacerlo. Entonces, el EAP Peer en vez de responder con una respuesta con código 2.04 Changed protegida con OSCORE (será el indicador de que la autenticación ha sido exitosa), responde con una respuesta 4.01 Unauthorized sin proteger, cuyo ID (27511) será el mismo que el de la petición para asociarla a ella y cuya carga útil será el mensaje “Fallo en la autenticación”, como se puede ver en texto plano.

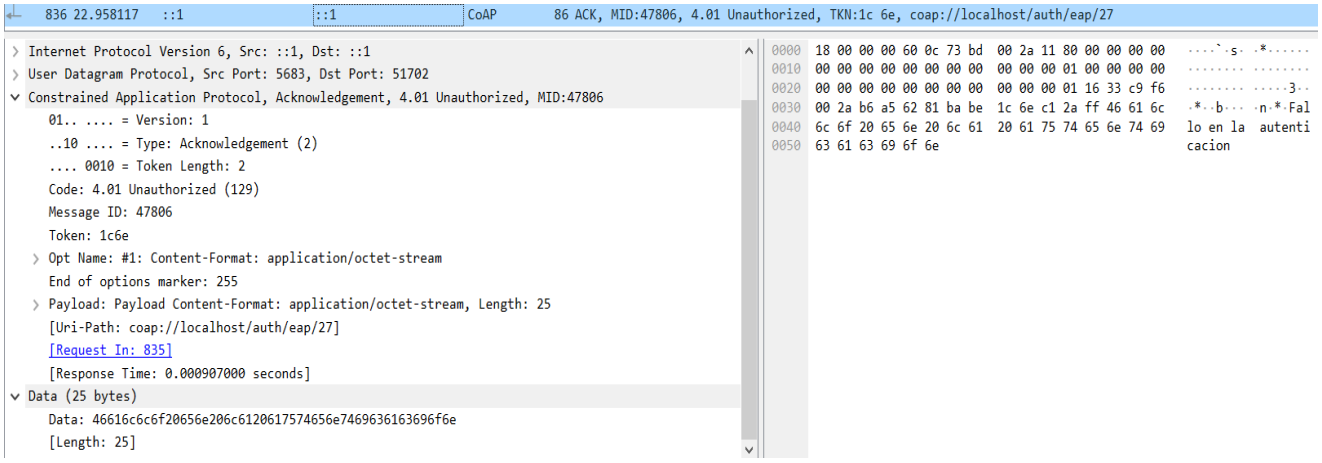


Figura 7.38.- Contenido del paquete de respuesta UNAUTHORIZED visto en Wireshark

Con esta respuesta el EAP Authenticator sabrá que la autenticación no se ha completado con éxito y que, por tanto, el dispositivo IoT no entra a formar parte del dominio de seguridad, no teniendo permiso para realizar ciertas acciones o acceder a ciertos recursos al no estar autenticado.

Además, se demuestra que la implementación responde correctamente, conforme a lo establecido en el Draft, ante los fallos de autenticación provocados por la imposibilidad de desproteger el mensaje de éxito.

7.2.4.- Eliminación del estado CoAP-EAP tras autenticación exitosa

Este escenario evalúa el comportamiento del sistema cuando el EAP Authenticator decide finalizar el estado CoAP-EAP de un dispositivo IoT antes de que expire el tiempo de vida de la sesión. Este caso puede darse por diversos motivos, como una decisión administrativa, cambios en las políticas de acceso o eventos que requieren revocar la autenticación activa del dispositivo. En esta situación, el autenticador envía un mensaje DELETE para eliminar el estado CoAP-EAP asociado al EAP Peer, invalidando su sesión de autenticación de forma anticipada.

En primer lugar, se presenta un diagrama MSC que detalla el flujo teórico esperado para este caso. Este diagrama ayuda a visualizar claramente cada uno de los paquetes que deben intercambiarse, incluyendo los códigos específicos.

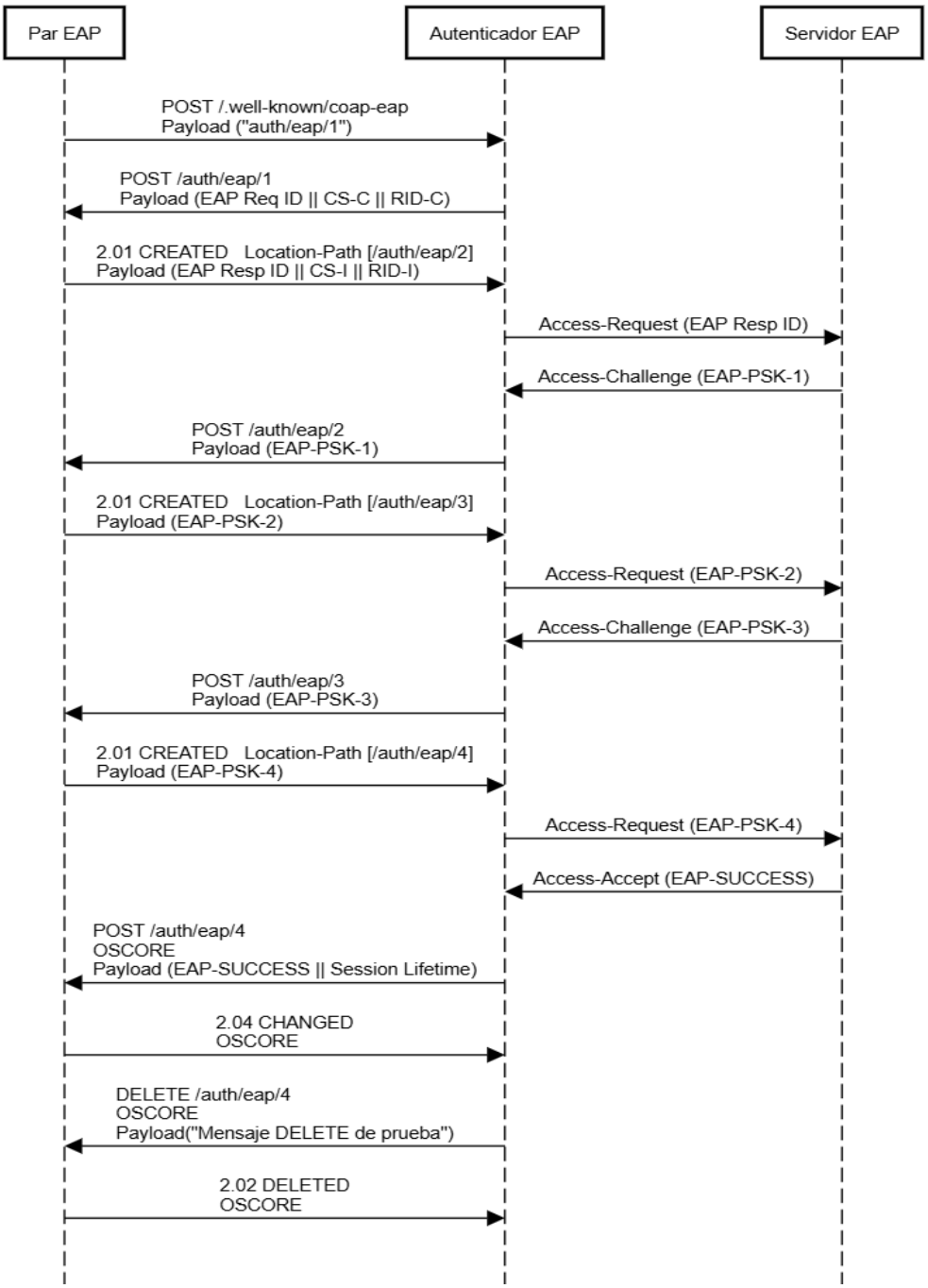
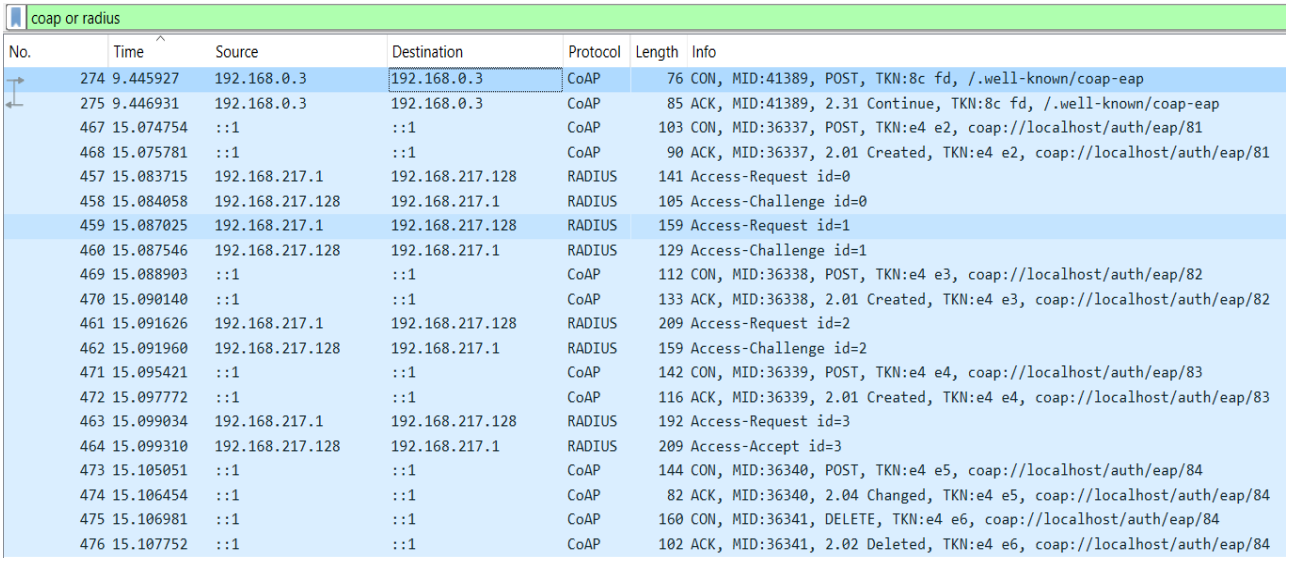


Figura 7.39.- Diagrama MSC del escenario de eliminación del dominio de autenticación tras autenticación exitosa

A continuación, se muestra la captura de tráfico obtenida en Wireshark al ejecutar este escenario con la implementación explicada en la sección 6.3. A partir de esta captura, se comentarán los paquetes reales observados en comparación con el flujo teórico del diagrama MSC. Se verificará que cada mensaje contenga los códigos y valores que indiquen

adecuadamente la voluntad del EAP Authenticator de eliminar el estado de autenticación CoAP-EAP asociado al EAP Peer, reflejando el comportamiento previsto del sistema cuando se requiere que un dispositivo IoT deje de estar autenticado por motivos de fuerza mayor.

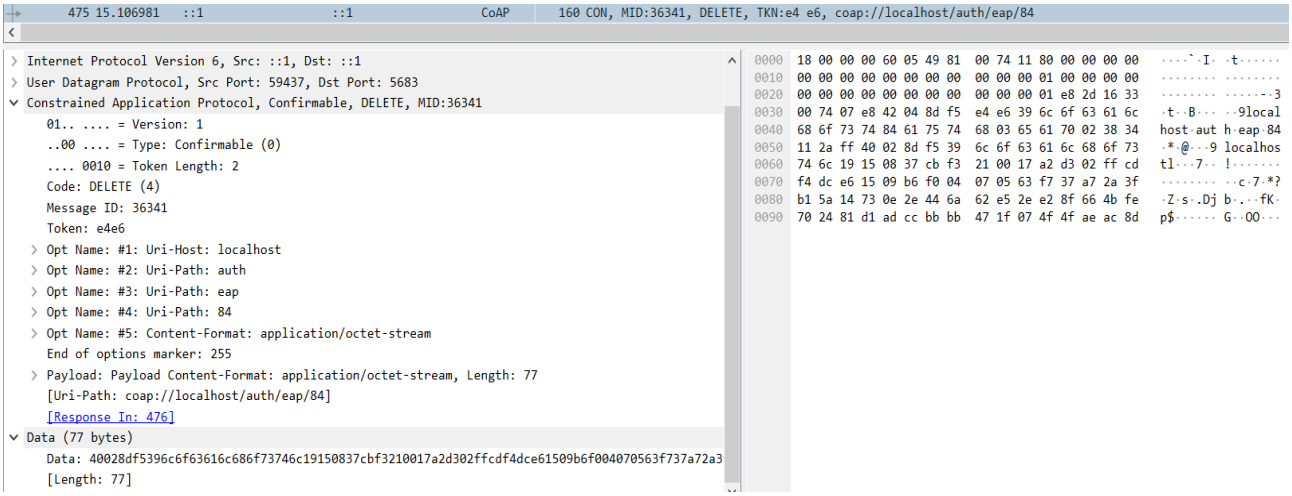


coap or radius

No.	Time	Source	Destination	Protocol	Length	Info
274	9.445927	192.168.0.3	192.168.0.3	CoAP	76	CON, MID:41389, POST, TKN:8c fd, /.well-known/coap-eap
275	9.446931	192.168.0.3	192.168.0.3	CoAP	85	ACK, MID:41389, 2.31 Continue, TKN:8c fd, /.well-known/coap-eap
467	15.074754	::1	::1	CoAP	103	CON, MID:36337, POST, TKN:e4 e2, coap://localhost/auth/eap/81
468	15.075781	::1	::1	CoAP	90	ACK, MID:36337, 2.01 Created, TKN:e4 e2, coap://localhost/auth/eap/81
457	15.083715	192.168.217.1	192.168.217.128	RADIUS	141	Access-Request id=0
458	15.084058	192.168.217.128	192.168.217.1	RADIUS	105	Access-Challenge id=0
459	15.087025	192.168.217.1	192.168.217.128	RADIUS	159	Access-Request id=1
460	15.087546	192.168.217.128	192.168.217.1	RADIUS	129	Access-Challenge id=1
469	15.088903	::1	::1	CoAP	112	CON, MID:36338, POST, TKN:e4 e3, coap://localhost/auth/eap/82
470	15.090140	::1	::1	CoAP	133	ACK, MID:36338, 2.01 Created, TKN:e4 e3, coap://localhost/auth/eap/82
461	15.091626	192.168.217.1	192.168.217.128	RADIUS	209	Access-Request id=2
462	15.091960	192.168.217.128	192.168.217.1	RADIUS	159	Access-Challenge id=2
471	15.095421	::1	::1	CoAP	142	CON, MID:36339, POST, TKN:e4 e4, coap://localhost/auth/eap/83
472	15.097772	::1	::1	CoAP	116	ACK, MID:36339, 2.01 Created, TKN:e4 e4, coap://localhost/auth/eap/83
463	15.099034	192.168.217.1	192.168.217.128	RADIUS	192	Access-Request id=3
464	15.099310	192.168.217.128	192.168.217.1	RADIUS	209	Access-Accept id=3
473	15.105051	::1	::1	CoAP	144	CON, MID:36340, POST, TKN:e4 e5, coap://localhost/auth/eap/84
474	15.106454	::1	::1	CoAP	82	ACK, MID:36340, 2.04 Changed, TKN:e4 e5, coap://localhost/auth/eap/84
475	15.106981	::1	::1	CoAP	160	CON, MID:36341, DELETE, TKN:e4 e6, coap://localhost/auth/eap/84
476	15.107752	::1	::1	CoAP	102	ACK, MID:36341, 2.02 Deleted, TKN:e4 e6, coap://localhost/auth/eap/84

Figura 7.40.- Captura Wireshark del eliminación del dominio de autenticación tras autenticación exitosa

Como se puede observar en la captura de Wireshark, tras llevarse a cabo la autenticación de forma exitosa, el EAP Authenticator envía al EAP Peer una petición CoAP con código DELETE (paquete número 72078). Esta petición, tal y como se especifica en el Draft, se envía al último recurso de estado CoAP-EAP dado por el servidor CoAP (‘auth/eap/84’) y está protegido por OSCORE para evitar falsificaciones.



475 15.106981 ::1 ::1 CoAP 160 CON, MID:36341, DELETE, TKN:e4 e6, coap://localhost/auth/eap/84

```
Internet Protocol Version 6, Src: ::1, Dst: ::1
User Datagram Protocol, Src Port: 59437, Dst Port: 5683
Constrained Application Protocol, Confirmable, DELETE, MID:36341
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0010 = Token Length: 2
    Code: DELETE (4)
    Message ID: 36341
    Token: e4e6
  > Opt Name: #1: Uri-Host: localhost
  > Opt Name: #2: Uri-Path: auth
  > Opt Name: #3: Uri-Path: eap
  > Opt Name: #4: Uri-Path: 84
  > Opt Name: #5: Content-Format: application/octet-stream
    End of options marker: 255
  > Payload: Payload Content-Format: application/octet-stream, Length: 77
    [Uri-Path: coap://localhost/auth/eap/84]
    [Response In: 476]
Data (77 bytes)
    Data: 40028df5396c6f63616c686f73746c19150837cbf3210017a2d302ffcdf4dce61509b6f004070563f737a72a3
    [Length: 77]
```

```
0000  18 00 00 00 60 05 49 81  00 74 11 80 00 00 00 00   ····`·I· ·t·····
0010  00 00 00 00 00 00 00 00  00 00 00 01 00 00 00 00   ········ ········
0020  00 00 00 00 00 00 00 00  00 00 00 01 e8 2d 16 33   ········ ·····-·3
0030  00 74 07 e8 42 04 8d f5  e4 e6 39 6c 6f 63 61 6c   ·t··B··· ··9local
0040  68 6f 73 74 84 61 75 74  68 03 65 61 70 02 38 34   host·aut h·eap·84
0050  11 2a ff 40 02 8d f5 39  6c 6f 63 61 6c 68 6f 73   ·*·@···9 localhos
0060  74 6c 19 15 08 37 cb f3  21 00 17 a2 d3 02 ff cd   tl···7·· !·····
0070  f4 dc e6 15 09 b6 f0 04  07 05 63 f7 37 a7 2a 3f   ········ ··c·7·*?
0080  b1 5a 14 73 0e 2e 44 6a  62 e5 2e e2 8f 66 4b fe   ·Z·s··Dj b····fK·
0090  70 24 81 d1 ad cc bb bb  47 1f 07 4f 4f ae ac 8d   p$····· G··OO···
```

Figura 7.41.- Contenido del paquete DELETE visto en Wireshark

La carga útil de este paquete será toda la petición DELETE (Code, Token, Message ID, Options y el payload "Mensaje DELETE de prueba") protegida por OSCORE. El EAP Peer desprotegerá esta carga útil, la procesará y así podrá obtener la información más relevante de la petición recibida (código y payload). En la consola del script peer.py se puede ver toda esta información:

Figura 7.42.- Información del paquete DELETE desprotegido vista en la consola del script peer.py

Entonces, el EAP Peer responde con un paquete de respuesta 2.02 Deleted, cuyo ID (36341) será el mismo que el de la petición para asociarla a ella y cuya carga útil será toda la respuesta DELETED (Code, Token, Message ID, opción Content-Format y el payload "MENSAJE DELETE RECIBIDO") protegida por OSCORE.

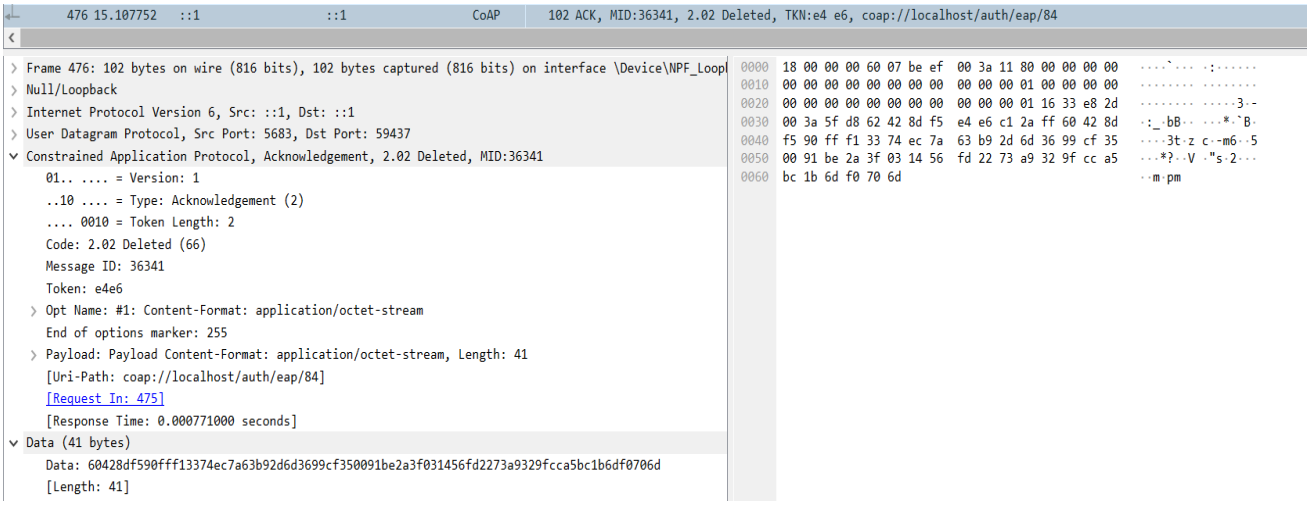


Figura 7.43.- Contenido del paquete de respuesta DELETED visto en Wireshark

Con esta respuesta el EAP Authenticator sabrá que el recurso especificado por la URI de la solicitud ha sido eliminado, esto indica que se ha suprimido el estado de autenticación CoAP-EAP del EAP Peer y que, por tanto, el dispositivo IoT ha dejado de formar parte del dominio de autenticación, lo que significa que ya no es reconocido como confiable o autorizado y por consiguiente, también ha dejado de pertenecer al dominio de seguridad, ya que la falta de autenticación invalida el acceso a recursos protegidos o canales seguros.

Además, se demuestra que la implementación responde correctamente, conforme a lo establecido en el Draft, ante la necesidad de retirar la autenticación a un cliente (dispositivo IoT).

Una vez analiza la respuesta del sistema implementado a estos escenarios de prueba, queda completamente demostrado que todos los paquetes intercambiados son los adecuados, que siguen a rajatabla todo lo especificado en el Draft y que se cumplen todos los requisitos funcionales definidos en la sección 5.1.1.

8. Conclusiones.

A lo largo del desarrollo de este Trabajo Fin de Grado he tenido la oportunidad de profundizar en temas que previamente solo había tratado de manera superficial, particularmente en el ámbito de la autenticación y el control de acceso en el Internet de las Cosas (IoT), un área de estudio en creciente relevancia. Implementar y evaluar el Internet Draft "EAP-based Authentication Service for CoAP" no solo ha representado un desafío técnico, sino también una oportunidad invaluable para entender en mayor detalle los desafíos de seguridad y las necesidades específicas de los dispositivos IoT, sobre todo en lo que respecta a sus limitaciones de recursos y la gran diversidad tecnológica que los caracteriza.

Uno de los principales retos ha sido enfrentarme a la implementación desde cero de un protocolo basado en un draft del IETF, lo que ha requerido seguir especificaciones detalladas, una comprensión exhaustiva tanto de los aspectos técnicos como de los requisitos de seguridad involucrados y, al mismo tiempo, tomar decisiones de diseño adecuadas para los entornos restringidos del IoT. Este proceso ha implicado múltiples etapas, desde la planificación inicial hasta la implementación final, asegurando que cada componente cumpla con los estándares definidos. También ha requerido una revisión constante de los principios de seguridad, de la optimización de recursos y de la eficiencia en la transmisión de datos, ya que los dispositivos IoT, como se sabe, suelen operar con restricciones de energía, procesamiento y almacenamiento. La optimización de estos elementos ha sido esencial para cumplir con los requisitos de la autenticación en redes IoT, todo ello sin comprometer la seguridad ni la integridad de los datos.

A nivel de desarrollo de software, el proyecto ha sido una experiencia integral. No solo me ha proporcionado una mayor familiaridad con las metodologías de desarrollo de software, sino que también me ha permitido consolidar habilidades y profundizar en herramientas y lenguajes de programación clave como Python, trabajando con librerías especializadas como aiocoap, cuyo uso se ha mostrado esencial para la ejecución del proyecto. Además, el uso de buenas prácticas de desarrollo, como el control de versiones, pruebas unitarias y documentación adecuada, me ha permitido asegurar que el código

implementado sea comprensible y mantenible, aspectos esenciales en proyectos a largo plazo.

La implementación de EAP-PSK y de RADIUS y la visualización de los intercambios de mensajes capturados con Wireshark, han sido fundamentales para comprender cómo diferentes métodos de autenticación pueden interactuar con protocolos de aplicación, como CoAP, en escenarios reales. Por último, la implementación y evaluación de algoritmos ligeros de criptografía para dispositivos con limitaciones de energía y procesamiento ha sido uno de los puntos más complejos, pero a la vez gratificantes, del proyecto.

Una vez concluido, puedo decir que me siento muy satisfecho con los resultados obtenidos y que este Trabajo Fin de Grado no solo ha cumplido con los objetivos iniciales, sino que ha superado mis expectativas en cuanto al conocimiento adquirido. Aunque siempre hay margen para mejorar, se han cumplido los objetivos principales, he ampliado mis competencias técnicas y académicas y he adquirido una variedad de habilidades que serán de gran utilidad en futuros proyectos y trabajos.

9. Trabajos futuros.

Durante el desarrollo de este proyecto, se han identificado varias ideas de futuras ampliaciones y mejoras que podrían implementarse para seguir evolucionando la solución propuesta. Algunas de estas ideas no han sido incluidas por cuestiones de tiempo o por quedar fuera del alcance inicial del trabajo, pero presentan una excelente oportunidad para ampliar las funcionalidades y optimizar la implementación actual.

Una de las primeras ampliaciones sería la implementación completa de la máquina de estados tanto del EAP Peer como del Controlador (EAP Authenticator), lo que permitiría gestionar de manera más eficiente las diferentes fases del proceso de autenticación. Este desarrollo facilitaría la implementación de una funcionalidad crucial: la reautenticación, especialmente importante cuando el estado CoAP-EAP está cerca de caducar. La reautenticación permitiría que los dispositivos IoT renueven su estado antes de perder la conexión, derivando nuevos materiales criptográficos (MSK/EMSK) y generando un nuevo contexto de seguridad OSCORE, aumentando así la protección frente a posibles filtraciones de información y reforzando la integridad de la comunicación.

Otro aspecto interesante sería permitir que la solución sea capaz de autenticar a múltiples dispositivos IoT simultáneamente. Actualmente, el sistema se ha diseñado para un único dispositivo, pero el Internet de las Cosas abarca redes con decenas o incluso cientos de dispositivos conectados, por lo que extender la autenticación a varios nodos mejoraría la escalabilidad y validaría el rendimiento del protocolo en situaciones más complejas y realistas.

Una mejora importante sería la incorporación de soporte para múltiples métodos EAP. En esta versión del proyecto, se ha trabajado principalmente con EAP-PSK, pero la inclusión de métodos adicionales, como EAP-TLS o EAP-AKA, permitiría adaptar el sistema a diferentes escenarios y a la diversidad de dispositivos IoT que existen, lo que incrementaría la flexibilidad y aplicabilidad de la solución en entornos más heterogéneos.

A su vez, probar la implementación en un entorno real también sería una gran mejora. Si bien el trabajo se ha desarrollado en un entorno controlado, validarlo con dispositivos IoT reales conectados a una red operativa permitiría analizar tanto el rendimiento como la seguridad del sistema bajo condiciones más representativas. Esto ofrecería datos relevantes sobre la eficiencia del uso de recursos, la estabilidad de la conexión, y cómo se comporta ante posibles fallos o amenazas de seguridad.

Por último, dado que la biblioteca aiocoap utilizada aún no tiene una implementación completamente madura de OSCORE, sería ideal mejorar esta funcionalidad a medida que la biblioteca evolucione. Con el tiempo, cuando el soporte para OSCORE esté mejor implementado, se podrían revisar y optimizar los mecanismos de protección de los mensajes de autenticación y su seguridad criptográfica, aprovechando las actualizaciones sin necesidad de desarrollar una implementación propia desde cero.

En resumen, este proyecto sienta las bases para una solución robusta de autenticación en IoT, pero existen múltiples oportunidades para seguir ampliando su funcionalidad. Estas mejoras no solo ofrecerían un sistema más completo y adaptable, sino que podrían ser desarrolladas en futuros trabajos de investigación o incluso en proyectos de fin de grado o master de otros estudiantes. De esta forma, el proyecto no solo se adapta a las necesidades actuales, sino que se deja espacio para su evolución continua.

10. Bibliografía.

[1] «¿Qué es el control de acceso a la red (NAC)?», Fortinet. Último acceso: 4 de octubre de 2024. [En línea]. Disponible en: https://www.fortinet.com/lat/resources/cyberglossary/what-is-network-access-control.html

[2] E. Eval, «Control de acceso a IoT: reforzar la ciberseguridad», Eval. Último acceso: 17 de octubre de 2024. [En línea]. Disponible en: https://eval.digital/es/blog/proteccion-de-datos/control-de-acceso-a-iot-reforzar-la-ciberseguridad/

[3] M. Beschokov, «¿Qué es el protocolo CoAP? Significado y arquitectura», Wallarm. Último acceso: 4 de octubre de 2024. [En línea]. Disponible en: https://lab.wallarm.com/what/que-es-el-protocolo-coap-significado-y-arquitectura/?lang=es

[4] J. Elder, «Cómo Kevin Ashton nombró El Internet de las Cosas», Avast. Último acceso: 8 de octubre de 2024. [En línea]. Disponible en: https://blog.avast.com/es/kevin-ashton-named-the-internet-of-things

[5] J. Gomez, «El Internet de las cosas (IoT) y la interconexión de dispositivos», Dync Solutions. Último acceso: 8 de octubre de 2024. [En línea]. Disponible en: https://dyncsolutions.com/general/el-internet-de-las-cosas-iot-y-la-interconexion-de-dispositivos/

[6] A. Behr, «Los mejores usos de la tecnología de comunicación inalámbrica IoT», Industry Today - Leader in Manufacturing & Industry News. Último acceso: 9 de octubre de 2024. [En línea]. Disponible en: https://industrytoday.com/best-uses-of-wireless-iot-communication-technology/

[7] J. Romero, «¿Qué es el NFC y para qué sirve? - Definición», GEEKNETIC. Último acceso: 9 de octubre de 2024. [En línea]. Disponible en: https://www.geeknetic.es/NFC/que-es-y-para-que-sirve

[8] I. Melero, «¿Qué es RFID y cómo funciona? Todo lo que necesitas saber», ADNID. Último acceso: 9 de octubre de 2024. [En línea]. Disponible en: https://adnid.com/blog/que-es-rfid-y-como-funciona-todo-lo-que-necesitas-saber/

[9] «Qué es Bluetooth y cómo funciona | TecnoNautas». Último acceso: 10 de octubre de 2024. [En línea]. Disponible en: https://tecnonautas.net/que-es-bluetooth-y-como-funciona/

[10] J. Romero, «¿Qué es el Bluetooth y para qué sirve? - Definición», GEEKNETIC. Último acceso: 10 de octubre de 2024. [En línea]. Disponible en: https://www.geeknetic.es/Bluetooth/que-es-y-para-que-sirve

[11] T. Moes, «¿Qué es Wi-Fi? Todo lo que necesita saber». Último acceso: 10 de octubre de 2024. [En línea]. Disponible en: https://softwarelab.org/es/blog/que-es-wi-fi/

[12] R. Greyrat, «Introducción de la tecnología IEEE 802.15.4», GeeksforGeeks. Último acceso: 11 de octubre de 2024. [En línea]. Disponible en: https://www.geeksforgeeks.org/introduction-of-ieee-802-15-4-technology/

[13] «¿Qué es el protocolo ZigBee en IoT?», Alotcer. Último acceso: 11 de octubre de 2024. [En línea]. Disponible en: https://www.alotceriot.com/es/que-es-el-protocolo-zigbee-en-iot/

[14] «Qué es ZigBee, cómo funciona y características principales», Venco Electrónica. Último acceso: 11 de octubre de 2024. [En línea]. Disponible en: https://www.vencoel.com/que-es-zigbee-como-funciona-y-caracteristicas-principales/

[15] «WirelessHART», Wikipedia. 6 de abril de 2024. Último acceso: 11 de octubre de 2024. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=WirelessHART&oldid=1217580217

[16] R. Greyrat, «¿Qué es 6LoWPAN?», GeeksforGeeks. Último acceso: 17 de octubre de 2024. [En línea]. Disponible en: https://www.geeksforgeeks.org/what-is-6lowpan/

[17] «LPWAN: qué son y para qué se utilizan», Becolve Digital. Último acceso: 14 de octubre de 2024. [En línea]. Disponible en: https://becolve.com/blog/lpwan-que-son-y-para-que-se-utilizan/

[18] A. Walton, «Narrowband-IoT (NB-IoT): ¿Qué es y para qué sirve? » Redes CCNA», CCNA desde Cero. Último acceso: 14 de octubre de 2024. [En línea]. Disponible en: https://ccnadesdecero.es/que-es-nb-iot/

[19] «¿Qué es sigfox? – SIGFOX», TEISA integración global- SIGFOX Operator. Último acceso: 17 de octubre de 2024. [En línea]. Disponible en: https://sigfox.com.py/que-es-sigfox/

[20] «LoRaWAN: ¿Qué es y para qué sirve?», UNIR. Último acceso: 14 de octubre de 2024. [En línea]. Disponible en: https://www.unir.net/revista/ingenieria/lorawan/

[21] «¿Qué es el Internet de las cosas (IoT)? | IBM», IBM. Último acceso: 14 de octubre de 2024. [En línea]. Disponible en: https://www.ibm.com/mx-es/topics/internet-of-things

[22] W. T. Toor, M. Alvi, y M. Agiwal, «Combined access barring scheme for IoT devices using bayesian estimation», Electronics, vol. 9, n.o 12, pp. 1-15, dic. 2020, doi: 10.3390/electronics9122191.

[23] «Métodos de autenticación en IoT: guía completa». Último acceso: 18 de octubre de 2024. [En línea]. Disponible en: https://techformacion.net/ciberseguridad/abc-autenticacion-iot-metodos-esenciales-dispositivos-conectados/

[24] M. Trnka, T. Cerny, y N. Stickney, «Survey of authentication and authorization for the Internet of Things», Secur. Commun. Netw., vol. 2018, pp. 1-17, jun. 2018, doi: 10.1155/2018/4351603.

[25] T. Moes, «¿Qué es la seguridad IoT? Todo lo que necesita saber». Último acceso: 15 de octubre de 2024. [En línea]. Disponible en: https://softwarelab.org/es/blog/que-es-la-seguridad-iot/

[26] C. Pu, A. Imtiaz, y S. Chakravarty, «Resource-efficient and data type-aware authentication protocol for internet of things systems», en 2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), Atlanta, GA, USA: IEEE, nov. 2023, pp. 1-10. doi: 10.1109/TPS-ISA58951.2023.00022.

[27] M. Abomhara y G. M. Koien, «Security and privacy in the Internet of Things: Current status and open issues», en 2014 International Conference on Privacy and Security in Mobile Systems (PRISMS), Aalborg, Denmark: IEEE, may 2014, pp. 1-8. doi: 10.1109/PRISMS.2014.6970594.

[28] S. C. Mukhopadhyay y N. K. Suryadevara, «Internet of Things: Challenges and Opportunities», en Internet of Things, vol. 9, S. C. Mukhopadhyay, Ed., en Smart Sensors, Measurement and Instrumentation, vol. 9. , Suiza: Springer International Publishing, 2014, pp. 1-17. doi: 10.1007/978-3-319-04223-7_1.

[29] M. A. Khan y K. Salah, «IoT security: Review, blockchain solutions, and open challenges», Future Gener. Comput. Syst., vol. 82, pp. 395-411, may 2018, doi: 10.1016/j.future.2017.11.022.

[30] «¿Qué es un token de autenticación?», Fortinet. Último acceso: 18 de octubre de 2024. [En línea]. Disponible en: https://www.fortinet.com/lat/resources/cyberglossary/authentication-token.html

[31] «¿Qué es la autenticación multifactor? - Explicación de la autenticación multifactor - AWS», Amazon Web Services, Inc. Último acceso: 18 de octubre de 2024. [En línea]. Disponible en: https://aws.amazon.com/es/what-is/mfa/

[32] D. García Carrillo, «Tema 5: Internet de las cosas», presentado en Servicios Telemáticos de nueva generación, Universidad de Oviedo.

[33] C. Bormann y P. E. Hoffman, «Concise Binary Object Representation (CBOR)», Internet Engineering Task Force, Request for Comments RFC 8949, dic. 2020. doi: 10.17487/RFC8949.

[34] J. Schaad, «CBOR Object Signing and Encryption (COSE)», Internet Engineering Task Force, Request for Comments RFC 8152, jul. 2017. doi: 10.17487/RFC8152.

[35] E. editorial de Ionos, «¿Qué es la autenticación de red IEEE 802.1X?», IONOS Digital Guide. Último acceso: 21 de octubre de 2024. [En línea]. Disponible en: https://www.ionos.es/digitalguide/servidores/know-how/ieee-8021x/

[36] J. Vollbrecht y L. Blunk, «PPP Extensible Authentication Protocol (EAP)», Internet Engineering Task Force, Request for Comments RFC 2284, mar. 1998. doi: 10.17487/RFC2284.

[37] D. Forsberg, P. Basavaraj, Y. Alper E., Y. Ohba, y H. Tschofenig, «Protocol for Carrying Authentication for Network Access (PANA)», Internet Engineering Task Force, Request for Comments RFC 5191, may 2008. doi: 10.17487/RFC5191.

[38] «Protocol for Carrying Authentication for Network Access», Wikipedia, la enciclopedia libre. 25 de marzo de 2022. Último acceso: 21 de octubre de 2024. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=Protocol_for_Carrying_Authentication_for_Network_Access&oldid=1079180057

[39] E. Rescorla y N. Modadugu, «Datagram Transport Layer Security», Internet Engineering Task Force, Request for Comments RFC 4347, abr. 2006. doi: 10.17487/RFC4347.

[40] F. Bersani y H. Tschofenig, «The EAP-PSK Protocol: A Pre-Shared Key Extensible Authentication Protocol (EAP) Method», Internet Engineering Task Force, Request for Comments RFC 4764, ene. 2007. doi: 10.17487/RFC4764.

[41] D. Simon, R. Hurst, y B. D. Aboba, «The EAP-TLS Authentication Protocol», Internet Engineering Task Force, Request for Comments RFC 5216, mar. 2008. doi: 10.17487/RFC5216.

[42] «Visión general de EAP-TLS: Definición, cómo funciona y beneficios | TRSPOS», TRSPOS. Último acceso: 21 de octubre de 2024. [En línea]. Disponible en: https://trspos.com/eap-tls/

[43] J. Arkko, V. Lehtovirta, y P. Eronen, «Improved Extensible Authentication Protocol Method for 3rd Generation Authentication and Key Agreement (EAP-AKA')», Internet Engineering Task Force, Request for Comments RFC 5448, may 2009. doi: 10.17487/RFC5448.

[44] C. Kaufman, P. E. Hoffman, Y. Nir, P. Eronen, y T. Kivinen, «Internet Key Exchange Protocol Version 2 (IKEv2)», Internet Engineering Task Force, Request for Comments RFC 7296, oct. 2014. doi: 10.17487/RFC7296.

[45] S. Farrell y H. Tschofenig, «Pervasive Monitoring Is an Attack», Internet Engineering Task Force, Request for Comments RFC 7258, may 2014. doi: 10.17487/RFC7258.

[46] H. Zhou, N. Cam-Winget, J. A. Salowey, y S. Hanna, «Tunnel Extensible Authentication Protocol (TEAP) Version 1», Internet Engineering Task Force, Request for Comments RFC 7170, may 2014. doi: 10.17487/RFC7170.

[47] G. Selander, J. P. Mattsson, F. Palombini, y L. Seitz, «Object Security for Constrained RESTful Environments (OSCORE)», Internet Engineering Task Force, Request for Comments RFC 8613, jul. 2019. doi: 10.17487/RFC8613.

[48] G. Selander, J. P. Mattsson, y F. Palombini, «Ephemeral Diffie-Hellman Over COSE (EDHOC)», Internet Engineering Task Force, Request for Comments RFC 9528, mar. 2024. doi: 10.17487/RFC9528.

[49] T. Aura, M. Sethi, y A. Peltonen, «Nimble Out-of-Band Authentication for EAP (EAP-NOOB)», Internet Engineering Task Force, Request for Comments RFC 9140, dic. 2021. doi: 10.17487/RFC9140.

[50] R. Moskowitz, R. Hummen, y M. Komu, «HIP Diet EXchange (DEX)», Internet Engineering Task Force, Internet Draft draft-ietf-hip-dex-24, ene. 2021. Último acceso: 22 de octubre de 2024. [En línea]. Disponible en: https://datatracker.ietf.org/doc/draft-ietf-hip-dex

[51] J. P. Mattsson y M. Sethi, «EAP-TLS 1.3: Using the Extensible Authentication Protocol with TLS 1.3», Internet Engineering Task Force, Request for Comments RFC 9190, feb. 2022. doi: 10.17487/RFC9190.

[52] E. Rescorla, H. Tschofenig, y N. Modadugu, «The Datagram Transport Layer Security (DTLS) Protocol Version 1.3», Internet Engineering Task Force, Request for Comments RFC 9147, abr. 2022. doi: 10.17487/RFC9147.

[53] R. Marin Lopez y D. Garcia Carrillo, «EAP-based Authentication Service for CoAP», Internet Engineering Task Force, Internet Draft draft-ietf-ace-wg-coap-eap-12, dic. 2024. Último acceso: 20 de diciembre de 2024. [En línea]. Disponible en: https://datatracker.ietf.org/doc/draft-ietf-ace-wg-coap-eap-12

[54] «Federación de Identidades UCOL», Universidad de Colima. Último acceso: 22 de octubre de 2024. [En línea]. Disponible en: https://portal.ucol.mx/federacion-identidades/

[55] Z. Shelby, K. Hartke, y C. Bormann, «The Constrained Application Protocol (CoAP)», Internet Engineering Task Force, Request for Comments RFC 7252, jun. 2014. doi: 10.17487/RFC7252.

[56] J. Vollbrecht, J. D. Carlson, L. Blunk, B. D. Aboba, y H. Levkowetz, «Extensible Authentication Protocol (EAP)», Internet Engineering Task Force, Request for Comments RFC 3748, jun. 2004. doi: 10.17487/RFC3748.

[57] D. García Garrillo, «From the paper to the IETF: A journey towards standardisation», Trabajo de Fin de Grado, Universidad de Murcia, 2023.

[58] D. Garcia Carrillo y R. Marin Lopez, «Lightweight CoAP-Based Bootstrapping Service for the Internet of Things», Sensors, vol. 16, n.o 3, Art. n.o 3, mar. 2016, doi: 10.3390/s16030358.

[59] G. Selander, J. P. Mattsson, F. Palombini, y L. Seitz, «Object Security for Constrained RESTful Environments (OSCORE)», Internet Engineering Task Force, Request for Comments RFC 8613, jul. 2019. doi: 10.17487/RFC8613.

[60] «Protocolo RADIUS: Características y fases de autenticación», CEC. Último acceso: 15 de noviembre de 2024. [En línea]. Disponible en: https://www.cec.es/protocolo-radius-caracteristicas-y-fases-de-autenticacion/

[61] A. López, «Protocolos AAA y control de acceso a red: Radius», INCIBE. Último acceso: 15 de noviembre de 2024. [En línea]. Disponible en: https://www.incibe.es/incibe-cert/blog/protocolos-aaa-radius

[62] A. Rubens, C. Rigney, S. Willens, y W. A. Simpson, «Remote Authentication Dial In User Service (RADIUS)», Internet Engineering Task Force, Request for Comments RFC 2865, jun. 2000. doi: 10.17487/RFC2865.

[63] «¿Qué es Python y para qué sirve? | Características, cómo funciona y qué se puede hacer», Web Devs. Último acceso: 21 de noviembre de 2024. [En línea]. Disponible en:

https://desarrolladoresweb.org/python/que-es-python-y-para-que-sirve-caracteristicas-como-funciona-y-que-se-puede-hacer/

[64] R. Maldonado, «Ventajas y desventajas de Python | KeepCoding Bootcamps», KEEPCODING Tech School. Último acceso: 21 de noviembre de 2024. [En línea]. Disponible en: https://keepcoding.io/blog/ventajas-y-desventajas-de-python/

[65] «PyCharm : Todo sobre el IDE de Python más popular», Formación en ciencia de datos | DataScientest.com. Último acceso: 21 de noviembre de 2024. [En línea]. Disponible en: https://datascientest.com/es/pycharm

[66] A. Manotoa, «Tutorial de PyCharm: características, instalación y usos», Platzi. Último acceso: 21 de noviembre de 2024. [En línea]. Disponible en: https://platzi.com/blog/pycharm/

[67] «Guía completa sobre el funcionamiento de la máquina virtual VMware», Tuto Window. Último acceso: 21 de noviembre de 2024. [En línea]. Disponible en: https://tutowindow.com/como-funciona-la-maquina-virtual-vmware/

[68] «Wireshark · About», Wireshark. Último acceso: 22 de noviembre de 2024. [En línea]. Disponible en: https://www.wireshark.org/about.html

[69] R. Altube, «Wireshark: Qué es y ejemplos de uso | OpenWebinars», OpenWebinars.net. Último acceso: 22 de noviembre de 2024. [En línea]. Disponible en: https://openwebinars.net/blog/wireshark-que-es-y-ejemplos-de-uso/

[70] «aiocoap – The Python CoAP library», aiocoap. Último acceso: 22 de noviembre de 2024. [En línea]. Disponible en: https://aiocoap.readthedocs.io/en/latest/

[71] M. Padua, «Patrones de diseño, descripciones estandarizadas para problemas repetitivos», IT Masters Mag. Último acceso: 25 de noviembre de 2024. [En línea]. Disponible en: https://www.itmastersmag.com/noticias-analisis/patrones-de-diseno-descripciones-estandarizadas-para-problemas-repetitivos/

[72] S. Nath, «Request Response Model, usages, anatomy and drawbacks», Medium. Último acceso: 25 de noviembre de 2024. [En línea]. Disponible en: https://medium.com/@sujoy.swe/request-response-model-usages-anatomy-and-drawbacks-42464e475cf5

[73] G. Zorn, «Microsoft Vendor-specific RADIUS Attributes», Internet Engineering Task Force, Request for Comments RFC 2548, mar. 1999. doi: 10.17487/RFC2548.