API security audit for an IoT environment

Auditoría en seguridad sobre una API aplicada a un entorno IoT

HOMERO TORRIJOS CHAPARRO

MÁSTER EN INTERNET DE LAS COSAS.
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Máster en Internet de las Cosas

Curso: 2020/2021

Fecha 4 de febrero de 2021

Convocatoria: Febrero de 2021
Nota: 6,5

Directores:

Prof. Juan Carlos Sáez Alcaide
Prof. Fernando Castro Rodríguez

Resumen

El *Internet de las Cosas* (**IoT**) es una tecnología claramente disruptiva en crecimiento, con impacto y capacidad inconcebible. Las **API REST** constituye una tecnología capaz de registrar e interconectar todo. Por la tanto es importante introducir una API segura que permita acaparar y/o defender la infiltración de atacantes en la red hacia los dispositivos IoT. La seguridad en una API representa una búsqueda de vulnerabilidades constantemente.

Las vías de comunicación entre el cliente y el servidor son el medio por el que las API sufren ataques que alteran el estado o funcionamiento del sistema. Asegurar el formato de las peticiones, los tipos de datos que se reciben o la validación de éstos representan una respuesta a posibles riesgos que afectan al rendimiento de la API. Por lo tanto auditar la seguridad de una API en desarrollo, diseño o implementación permite asegurar el correcto funcionamiento, gracias a la aplicación de buenas prácticas y el uso de herramientas de seguridad que permitan descubrir esas vulnerabilidades.

En el presente Trabajo Fin de Máster (TFM) se desarrolla una API de autoría propia adaptada a un entorno IoT con el objetivo de realizar una auditoría de seguridad y por consiguiente asegurar el sistema a posibles ataques, evitando la perdida de datos o alteración de los mismos tanto del lado del cliente como del servidor. La auditoría se llevará a cabo siguiendo las buenas prácticas de la **OWASP Foundation** y haciendo uso de **ZAP** como herramienta de **pentesting**. De igual forma se adhiere a este proyecto **OpenAPI Specification** para generar la documentación mediante el framework **Swagger**. La adquisición de los datos se realiza con **BLE** desde un nodo sensor genérico conectado a **Node RED** y se envían los datos por peticiones REST hacia el servidor, donde se almacenan todos los datos en **MongoDB**. Para el desarrollo de la API se utilizó **Node.js**.

Palabras clave

Lista de palabras clave: API REST, Pentesting, BLE, Node RED, OWASP Foundation, Swagger, OpenAPI Specification, ZAP, MongoDB, Node.js.

Abstract

Internet of Things (IoT) is a clearly disruptive technology in growth, with impact and capability unthinkable. One REST API's role is a technology capable of tracking and interconnecting everything. For this reason it is important to introduce a secure API that allows to control and/or defense the infiltration of attackers in the network towards the IoT devices. A secure API represents a constant searching of vulnerabilities.

Client-server communication channels are the way APIs suffer attacks that alter the state or functioning of the system. Secure the format of the requests, the types media of data received or the validation of these represent a response to possible risks that may alter API performance. Auditing API security under development, design or implementation makes it possible to ensure its optimum operation, through the application of proper practices and the use of security tools that allow these vulnerabilities to be discovered.

In this final work (TFM) is developed an API with the goal of conducting a security audit and then ensure the system to possible attacks, avoiding the loss of data or altering them on the client side and server side. The audit will be implemented following OWASP best practices and making use of ZAP as a pentesting tool. OpenAPI Specification is also added to this project and is implemented to integrate the documentation through Swagger framework. Data acquisition is done by BLE from a generic sensor node connected to RED Node and sent using REST requests to the server, which stores all data in MongoDB. Node.js was employed for API development.

Keywords

List of keywords: API REST, Pentesting, BLE, Node RED, OWASP, Swagger, OpenAPI Specification, ZAP, MongoDB, Node.js..

Índice de figuras

Índice de tablas

Índice general

Agradecimientos

Al Prof. Juan Carlos Sáez Alcaide, por su tiempo al perfilar el trabajo en su conjunto, así como la orientación técnica y comentarios por parte del Prof. Fernando Castro Rodriguez.

Dedicatoria

A mi madre y hermanos por ser parte de mi familia

Capítulo 1

Introducción

"Aut non tentaris, aut perficie. (No lo intentes, o llévalo hasta su fín)".
– OVIDIO, 43 a.C. – 17 d.C.

1.1. Antecedentes

La adopción de las API a nivel empresarial ha superado las expectativas dado que se ha visto la proliferación de é stas en casi todas las industrias. No es una exageración decir que un negocio sin una API es como tener una computadora sin acceso a internet. Uno de los aspectos clave detrás del éxito del Internet de las Cosas (IoT, Internet of Things) son las API[56] el fundamento para la creación de canales de comunicación en éste contexto.

Según un informe del Wipro Council for Industry Research[72], un vuelo de seis horas en un Boeing 737 desde Nueva York a Los Ángeles genera 120 terabytes de datos que se recopilan y almacenan en el avión. Con la explosión de sensores y dispositivos que se apoderan del mundo, es necesario que exista una forma adecuada de almacenar, administrar y analizar los datos. En 2014 se estimaba que se almacenaban 4 Zettabytes de información en todo el mundo, y se estima que, para 2020, ese número aumentaría hasta 35 Zettabytes[4]. Lo más interesante es que el 90 % de los datos que tenemos se generan solo durante los últimos diez años. El papel de las API en el contexto de Internet de las Cosas es en definitiva el medio que conecta los dispositivos a otros dispositivos y/o a la nube.

En un articulo publicado por Cisco[7] sobre el Internet de las Cosas se estima que durante 2008, la cantidad de cosas conectadas a internet superó el número de habitantes en la tierra. Más de 12.5 mil millones de dispositivos se conectaron a internet en 2012 y 25 mil millones de dispositivos a finales de 2015. Se estima que para fines de 2015 y 2021 se conectarán 50 mil millones de dispositivos.

El mundo está más conectado que nunca. Nosotros compartimos fotos en nuestras redes sociales como Instragam, Facebook, Twitter, etc. Las conexiones es ilimitada a través de una API, en consecuencia todo esto es posible solo gracias a las API, las cuales han proliferado en los últimos años. eBay, Salesforce, Expedia y muchas otras compañías generan anualmente una gran cantidad de ingresos con el uso de las API. Cabe destacar de igual forma los medios o perfiles con las tecnologías que se integran las API son amplias. Por ejemplo las Single-Page Applications (SPAs), Native Mobile Applications (NMA) de Android o iOS, y Browser-Less Applications, etc.

La seguridad de las API ha evolucionado mucho en los últimos años. JSON juega un papel importante en la comunicación de una API. La mayoría de las API desarrolladas en la actualidad solo admiten JSON, no XML[14].

Las API empresariales se han convertido en la forma habitual de exponer las funciones comerciales al mundo exterior a través de internet. Exponer la funcionalidad es conveniente, pero por supuesto conlleva un riesgo de transición. Como es el caso de cualquier sistema software, los desarrolladores tienden a ignorar el elemento de seguridad durante la fase de diseño de la API. Solo en la implementación o en el momento de la integración comienzan a preocuparse por la seguridad.

Entre los niveles de seguridad que tienen riesgo de transición es la autenticación y autorización, medio por el cual los datos personales son vulnerables dentro y fuera de la aplicación, en el caso de ser clientes o usuarios. De igual forma sucede en los canales por donde fluyen los datos, que a su vez son constantemente atacados.

Finalmente para afrontar este problema se presenta este Trabajo Fin de Máster en el

cuál se implementa una arquitectura REST de autoría propia que se auditará en términos de seguridad y documentación. La auditoría de seguridad se llevará a cabo en los niveles de autorización y autenticación. Esto que permitirá conocer la vulnerabilidades correspondientes y hacer la corrección en la API. Para llevar a cabo esto se implementarán tecnologías de pentesting, documentación y desarrollo así como las recomendaciones técnicas de desarrollo a nivel de seguridad.

1.2. Objetivo

El principal objetivo de este Trabajo Fin de Máster será desarrollar una API adaptada a un entorno IoT, así como auditar su funcionamiento global en términos de seguridad y documentación.

Para ello se ha hecho una infraestructura en la que se han ido eligiendo distintas tecnologías que permiten agilizar el desarrollo, documentación y auditoría en seguridad de la API.

En la figura 1.1 se muestra un diagrama que contempla tres secciones de interacción de la API que a continuación se detalla su funcionalidad principal:

1. *Fuente de datos*: El dispositivo SensorTAG CC2650 generará los datos relativos a la *humedad*, *temperatura*, *presión* e *intensidad lumínica*. Mismos que serán enviados vía bluetooth así la Raspberry Pi en un tiempo determinado para cada sensor.
2. *Cliente*: Node RED funcionará como cliente dado que recibirá los datos del SensorTAG en la Raspberry Pi y estos a su vez serán enviados por peticiones POST hacia la API. La cabecera de la petición contendrá un JWT, por lo que la API autorizará o rechazará la petición. Estás peticiones se enviarán por Wi-Fi en la red local .
3. *Servidor*: Administra las peticiones de los clientes autorizados y autenticados para borrar, obtener, consultar y modificar los recursos. La API esta conectada a una base de datos NoSQL (MongoDB). Swagger se usa como framework para llevar a cabo la documentación de la API, misma que se integrará en un endpoint de la API. Swag-

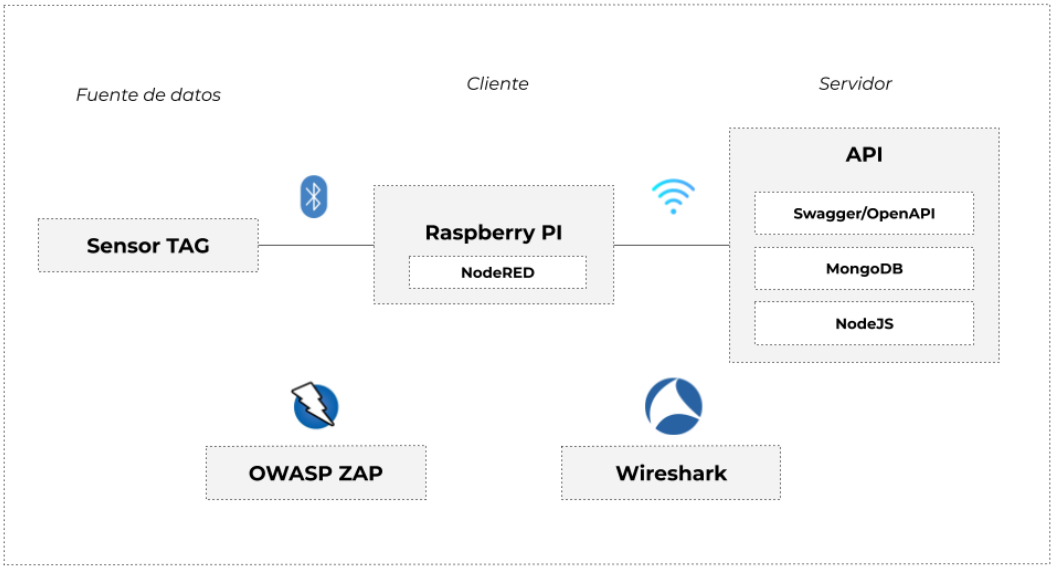

Figura 1.1: *Diagrama global de auditoría y comunicación de la API*

ger implementa OpenAPI Specification en el servidor. Aquí se encuentran definidos los endpoints, los esquemas, los tipos de datos, las peticiones y respuestas de cada endpoint.

1.3. Plan de trabajo

Para llevar a cabo la integración de éste Trabajo Fin de Máster en la figura 1.2 se muestran las actividades a realizar. Lo cuál comprende una breve introducción de la arquitectura REST.

Para integrar la API se definirán los esquemas del cliente y sensortag. Esto abarca los tipos de datos y validaciones correspondientes. Mientras tanto en los endpoints se definirán el tipo de operaciones a realizar, así como sus respuestas que devuelven para cada una. La API tendrá dos niveles de seguridad, en primera instancia los clientes registrados se deben autenticar mediante correo electrónico y contraseña, con el objetivo de identificarse y con esto el servidor responderá con un token que permitirá autorizar las peticiones del cliente y validarlas en el servidor; en su conjunto esto abarcará el desarrollo e implementación de la

API.

Para llevar a cabo la auditoría de la API ya en funcionamiento se utilizarán ZAP y Wireshark. ZAP funcionará como herramienta de pentesting para encontrar las vulnerabilidades de la API desarrollada. Wireshark se usará para escanear la red por donde viajarán los datos, de esta forma permite conocer los puntos débiles que afrontan las API, como son los puertos, las cabeceras y paquetes que son capturados para obtener datos sensibles de la API. De igual forma para asegurar el transito de los datos entre el cliente y servidor, se usará una comunicación cifrada.

Nombre	Fecha de ini.	Fecha de fin
Arquitectura REST	2/03/20	20/03/20
Antecendentes	2/03/20	11/03/20
Seguridad	12/03/20	20/03/20
Requerimientos	23/03/20	1/05/20
Tecnología de auditoría	23/03/20	1/04/20
Tecnología de documentación	2/04/20	14/04/20
Tecnología de bases de datos	15/04/20	22/04/20
Tecnología de desarrollo	23/04/20	1/05/20
Desarrollo e implementación de la API	4/05/20	26/06/20
Esquemas de la API	4/05/20	15/05/20
Rutas de la API	18/05/20	29/05/20
Autenticación y Autorización de la API	1/06/20	12/06/20
Documentación de la API	15/06/20	26/06/20
Auditoría de la API	29/06/20	24/07/20
Escenarios vulnerables de la API	29/06/20	10/07/20
Escenarios seguros de la API	13/07/20	24/07/20
Conclusiones	27/07/20	31/07/20

Figura 1.2: *Actividades del plan de trabajo*

Para estructurar las actividades respecto del tiempo se presenta en la figura 1.3 las etapas y sus tareas independientes con las que se planea trabajar.

1. Estudiar la arquitectura de un sistema REST.
2. Desarrollar una API en Node.js que permita administrar recursos del sensortag, administrar la seguridad de los clientes por medio de autorización y autenticación adaptado a una arquitectura REST.
3. Documentar la API haciendo uso de la OpenAPI Specification con la implementación

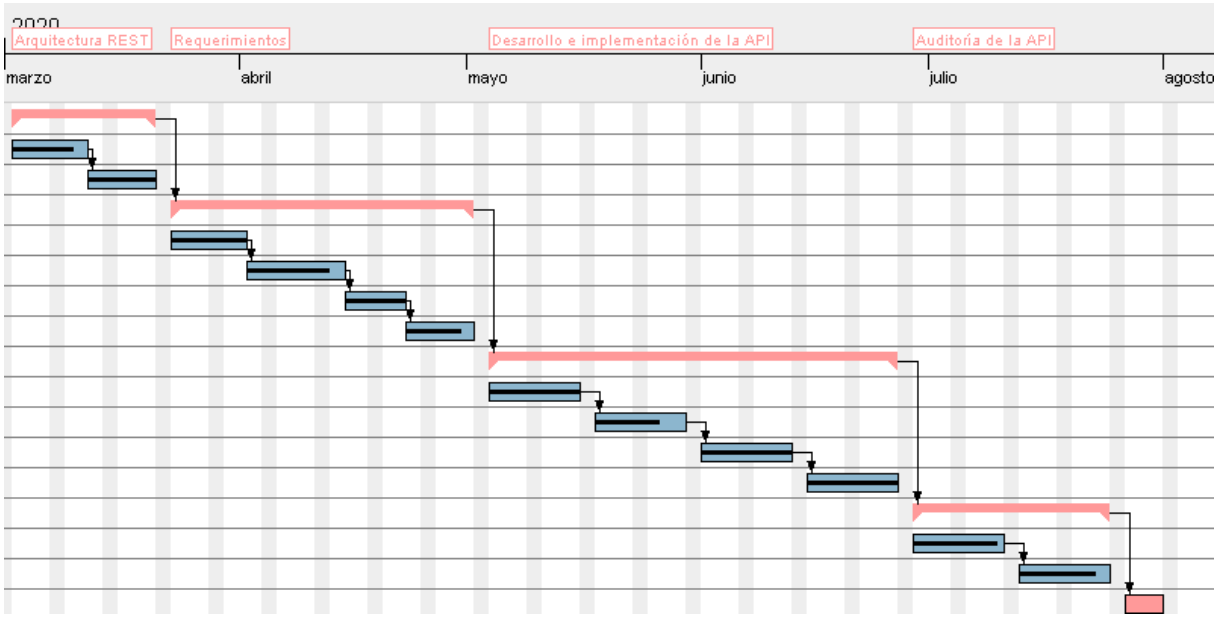

Figura 1.3: *Plan de trabajo en función del tiempo*

del framework Swagger UI, integrándolo a la API desarrollada.

4. Analizar las distintas fallas de seguridad de una arquitectura REST, listadas por la OWASP en la API desarrollada.
5. Analizar y aplicar las buenas prácticas de desarrollo propuestas por el proyecto OWASP API Security Top 10 2019.
6. Vulnerar y asegurar la API, por medio ataques de fuerza bruta, ataque de `man in the middle` con Wireshark.
7. Asegurar las vías de comunicación entre cliente y servidor, por medio de certificados, claves públicas y privadas, que permitan firmar cada uno de los mensajes sin que sean alterados.
8. Usar ZAP como herramienta para explotar posibles vulnerabilidades de seguridad en la aplicación y resolverlas dentro del sistema diseñado.

Mientras tanto en el resto del documento se presenta la estructura de la siguiente forma:

- En el **capítulo** 2, se hace una breve introducción de los conceptos asociados a la seguridad a la seguridad de un sistema REST.

- En el **capítulo** 3, se describe el conjunto de tecnologías que integra el desarrollo de la API. Se resaltan los aspectos hardware y software del Trabajo Fin de Máster. Así como los elementos que interactúan en el funcionamiento de la API.

- En el **capítulo** 4, se describe en su totalidad la arquitectura de la API desarrollada.

- En el **capítulo** 5, se ilustra de manera general la incorporación de OpenAPI Specification en la API a través de Swagger.

- En el **capítulo** 6, se desarrollan los escenarios vulnerables y seguros de la API. Se utiliza ZAP como herramienta de pentesting. Al igual se aplican las buenas prácticas según las OWASP API Security Project Top 10 en la API desarrollada.

- En el **capítulo** 7, se delimitan la conclusiones finales y trabajo futuro que permitan extender la usabilidad técnica de este trabajo.

Para reforzar el contenido de éste Trabajo Fin de Máster se presenta en el Apéndice C el contexto del desarrollo de las API, lo cual involucra los mecanismos de seguridad más usados, frameworks de desarrollo, lenguajes de programación más usados y algunos módulos de desarrollo en Node.js y finalmente se presenta en el Apéndice D el código de la API desarrollada en Node.js, alojada en GitHub.

Capítulo 2

Marco teórico

"Errare Humanum Est, perseverare autem diabolicum, et tertia non datur. (Equivocarse es de humanos, no pasa nada pero no aprender de los errores es realmente un problema.)". – Séneca, 4 a.C. – 65 d.C.

En septiembre de 2018 el equipo de Facebook descubrió un ataque que puso en riesgo la información personal de más de 50 millones de usuarios de Facebook[27]. Los atacantes explotaron múltiples problemas del código fuente de Facebook y se apoderaron de los tokens de acceso[22] que pertenecen a más de 50 millones de usuarios.

En esencia el token de acceso es una especie de clave temporal usada para acceder a un recurso en nombre de otra persona. Por ejemplo, si un usuario o cliente quiere compartir fotos subidas a Instagram en el muro de Facebook, le daría un token de acceso correspondiente a este muro de Facebook, el cual se obtiene en Facebook hacia Instagram.

Tomando en cuenta el sistema de Facebook, y particularmente en una API se tiene en referencia que el papel de la seguridad es un elemento crítico de cualquier sistema, dado que los nuevos problemas de seguridad y las vulnerabilidades siempre se están redescubriendo, y es importante protegerlos de ataques.

Para garantizar que una aplicación sea segura, hay muchas cosas que los ingenieros tienden a hacer. Esto incluye la validación de entradas, el uso del protocolo SSL (Secure Socket Layer) para el cifrado de las comunicaciones, la validación de los tipos de contenido, el mantenimiento de los registros de auditoría y la protección contra ataques CSRF (Cross Site Request Forgery)[48], así como también del XSS (Cross Site Scripting)[47].

Es complejo construir un sistema 100 % seguro. Lo único que se puede hacer es hacer más difícil el trabajo al atacante. En consecuencia en este capítulo se abordarán los conceptos teóricos de seguridad.

2.1. Prueba de seguridad

Una **prueba de seguridad** es el "*proceso que consiste en evaluar y verificar un sistema con el fin de descubrir los riesgos de seguridad y las vulnerabilidades del sistema y los datos que contiene*"[32]. No hay una terminología universal. Para los fines de este Trabajo Fin de Máster se define la **prueba de auditoría** como el "*análisis y el descubrimiento de vulnerabilidades sin intentar explotarlas realmente*"[32]. Del mismo modo, la **prueba** o **test** se define como el "*descubrimiento y el intento de explotación de las vulnerabilidades*"[32].

Las pruebas de seguridad se realizan a menudo, de forma arbitraria, según el tipo de vulnerabilidad que se pruebe o el tipo de prueba que se realice. Un ejemplo común es:

1. *Evaluación de las vulnerabilidades.* El sistema es escaneado y analizado por aspectos de seguridad.
2. *Prueba de penetración.* El sistema se somete a un análisis y un ataque malicioso simulado.
3. *Prueba de ejecución.* El sistema se somete a un análisis y pruebas de seguridad de un usuario final.
4. *Revisión del código.* El código del sistema se somete a una revisión y análisis riguroso en busca de vulnerabilidades de seguridad.

Cabe señalar que la evaluación de riesgos suele figurar como parte de las pruebas de seguridad. Una evaluación de riesgos no es en realidad una prueba, sino el análisis de la gravedad percibida de los diferentes riesgos (seguridad del software, seguridad del personal, seguridad del hardware, etc.) y cualquier medida de mitigación de esos riesgos.

Una prueba de penetración (pentesting) se lleva a cabo como si el *tester* fuera un atacante externo, con el objetivo de ingresar en el sistema y robar datos o llevar a cabo algún tipo

de ataque de denegación de servicio. La prueba de penetración tiene la ventaja de ser más precisa porque tiene menos falsos positivos (resultados que indican la existencia de una vulnerabilidad que no está realmente presente), pero puede llevar mucho tiempo ejecutarla.

El pentesting también se utiliza para probar los mecanismos de defensa, verificar los planes de respuesta y confirmar el cumplimiento de la política de seguridad. El pentesting automatizado es un elemento fundamental en la validación de la integración continua. Contribuye a descubrir nuevas vulnerabilidades así como a realizar correcciones de vulnerabilidades anteriores en un entorno que cambia rápidamente, y para el cual el desarrollo puede ser altamente colaborativo y distribuido.

2.1.1. Proceso de pentesting

Tanto el pentesting manual como el automatizado se emplean, a menudo en conjunto, para probarlo desde los servidores hasta las redes, a los dispositivos, o endpoints.

El pentesting suele seguir las siguientes etapas[32]:

- *Exploración*: El tester busca conocer el sistema que se está analizando. Esto incluye tratar de determinar qué software está en uso, qué endpoints existen, qué actualizaciones están instaladas, etc. Esto también incluye buscar en el sistema el contenido oculto, el contenido con vulnerabilidades, y otros indicios de debilidad.
- *Ataque*: El tester intentará explotar las vulnerabilidades conocidas o sospechosas para probar que efectivamente existen.
- *Informe*: El tester informa los detalles de sus pruebas, incluyendo las vulnerabilidades, cómo se explotaron, la dificultad de las mismas, y la gravedad de la vulnerabilidad.

El principal objetivo del pentesting es la búsqueda de vulnerabilidades para que éstas se puedan solucionar. Se puede también verificar si un sistema no es vulnerable a una categoría conocida o a un fallo específico; o, en el caso de las vulnerabilidades que se han notificado como fijas, se puede comprobar que el sistema ya no es vulnerable a ese fallo.

2.2. Estudio de los mecanismos de seguridad

En un articulo[1] se analizaron aproximadamente 500 APIs, de las que el (86 %) requerían *autenticación de usuario* para hacer llamadas a la API. Dentro del estudio se los usuarios deben estar autenticados y autorizados para ejecutar operaciones, mientras que otras operaciones solo requieren autenticación por ejemplo operaciones de datos específicos que involucran a un usuario en particular.

Dentro de esta investigación se ha agregado en el Apéndice C.1 un gráfico donde se muestra la distribución de los mecanismos de autorización y autenticación usados en este articulo.

Cabe mencionar algunos aspectos esenciales de seguridad de un sistema REST basado en HTTP:

- *Los sistemas de REST están diseñados para no tener estado.* REST define el servidor sin estado, lo que significa que almacenar los datos del usuario en sesión después del inicio de sesión inicial no es una buena idea.
- *Recomendable el uso de HTTPS.* En los sistemas REST basados en HTTP, HTTPS debe ser usado para asegurar el canal de comunicación, haciendo más difícil la captura y lectura del tráfico de datos.

2.3. Mecanismos de seguridad sin estado

OAuth 1.0, OAuth 2.0, Digest Auth, y Basic Auth + TSL son los principales métodos de autorización hoy en día. Han sido implementados en todos los lenguajes de programación modernos, y han demostrado ser la elección correcta para el trabajo.

Todos dependen de que el usuario tenga información almacenada en algún tipo de capa de caché en el servidor. Este pequeño detalle, especialmente para los expertos en seguridad, significa que no se puede diseñar un sistema por sesión, porque va en contra de una de las más básicas de las restricciones impuestas por REST: *la comunicación entre el cliente y el servidor deben ser sin estado.*

2.4. JSON Web Token

Un JSON Web Token (JWT) es un contenedor para transportar datos entre las partes involucradas en formato JSON. El JWT esta definido por la IETF (Internet Engineering Task Force) en el RFC7519[21], que se define como un medio compacto y seguro de URL para representar *claims* en dos partes. En esencia un JWT es un bloque de información en formato JSON que esta firmada lo cual puede verificar su autenticidad.

Existen muchos usos de los JWT. Entre estos están los servicios de autorización con *OAuth*, *Sesiones de usuario*, *Restablecimiento de contraseñas*, etc.

Algunos de sus características del lado del cliente:

- **Sesiones sin estado**: El principal aspecto de esta aplicación recae en el uso de firmas y cifrados para autenticar y proteger los contenidos de la sesión.
- **Compacto**: pueden enviarse a través de una URL, un parámetro `POST`, `PUT`, `GET`, `DELETE` o dentro de la cabecera HTTP. Además, el tamaño más pequeño significa que la transmisión es rápida.
- **Claims**: Cierto número de información del usuario/entidades empaquetadas en el JWT.

La comunicación entre cliente y servidor mediante JWTs se diferencia de la respuesta y el formato dado por el servidor como se muestra en la figura 2.1. En este caso el cliente envía sus credenciales para autenticarse y el servidor responde con un JWT que le devuelve al cliente. El cliente ahora tiene la autorización para hacer solicitudes. En este caso es de tipo `GET`.

Un JSON Web Token consta de tres partes separadas por puntos (**.**) como se muestra en la figura 2.2, que son cabecera y/o encabezado (**Header**) en la primera parte color azul, cuerpo (**Body**) como segunda parte de color verde y en la tercera parte se tiene la firma (**Signature**). Cabe destacar que existen sitios web que permiten ver la estructura de un JWT para revisar las partes visibles que componen el JWT[5]. En la figura 2.2 se presenta un ejemplo básico.

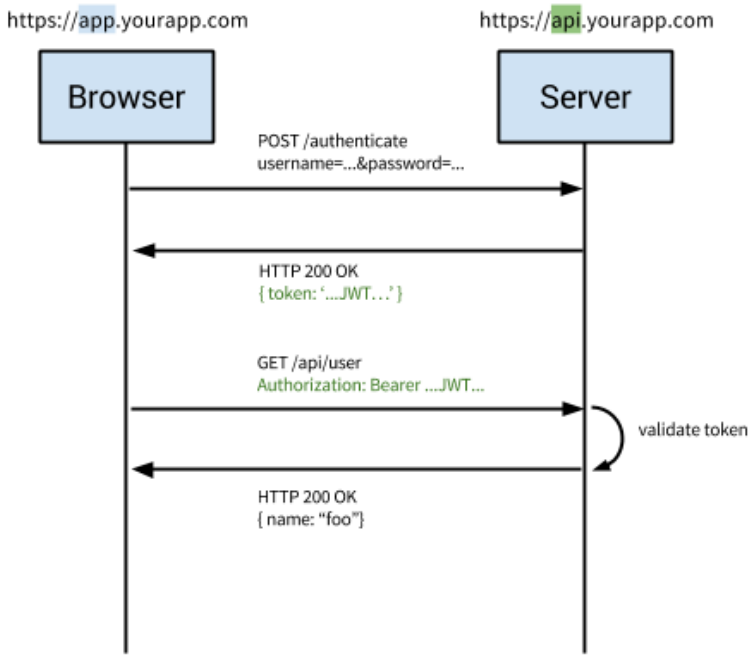

Figura 2.1: *Comunicación implementando JWT*

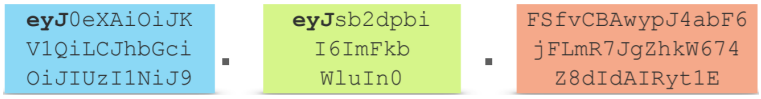

Figura 2.2: *JSON Web Token: header.body.singture*[25]

En la figura 2.2 el JWT está codificado en base *64url*[15], que hace referencia a las primeras tres letras del Token en este caso **eyJ** = **Base64(“ {...} ”)**, para tener una mejor interpretación una de cada conjunto que lo compone como se muestra en la figura 2.3.

Figura 2.3: *JSON Web Token: Base64*[25]

La cabecera normalmente consta de dos partes: el tipo (*typ*) de token, que es JWT, y el algoritmo de hashing que se utiliza, como por ejemplo: HS256, HS512, RSA256, etc. Cada JWT lleva una cabecera con *claims* sobre sí mismo. Los JWT cifrados llevan información sobre los algoritmos criptográficos utilizados para el cifrado de claves y el cifrado de contenido

como se puede observar en la figura 2.4. El único *claim* obligatorio para una cabecera JWT no cifrada es el argumento **alg**: que es el algoritmo principal para firmar y/o descifrar este JWT.

```
{
  "alg": "HS512",
  "typ": "JWT"
}
```

Figura 2.4: *JWT: Header*[5]

El *body* como se muestra en la figura 2.5 es el elemento donde normalmente se añaden todos los datos de usuario que son de interés.

```
{
 "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Figura 2.5: *JWT: Body*[5]

Los *claims* son declaraciones o etiquetas acerca de una entidad (normalmente, el usuario) y sus metadatos adicionales.

A continuación se muestran una lista que contiene la palabra reservada de los *claims reservados*, *públicos*, *privados* así como su descripción correspondiente.

1. *Reserved claims*: Se trata de un conjunto de claims predefinidos que no son obligatorios. Algunos de ellos son:
   - **alg** (algoritmo)
   - **iss** (emisor)
   - **sub** (asunto)
   - **aud** (público)
   - **iat** (fecha y hora de emisión)
   - **exp** (fecha y hora de vencimiento)

- **uid** (identicador del usuario, 1 - 36 caracteres)
- **nbf** (hora actual después del inicio del período de validez de este token)

2. *Public claims*: Para evitar colisiones deben ser definidos en el IANA JSON Web Token Registry o ser definidos como un URI que contiene un espacio de nombres resistente a colisiones.
3. *Private claims*: Son las reclamaciones personalizadas creadas para compartir información entre las partes que acuerdan usarlas.

Ya en la ultima parte del JWT se tiene la firma como se muestra en la figura 2.6. Hay que tomar el header, body, y una clave, así como el algoritmo especificado en el encabezado, y firmarlo. La firma se utiliza para verificar que el remitente del JWT es quien dice ser y para asegurarse de que el mensaje no se ha cambiado en el camino.

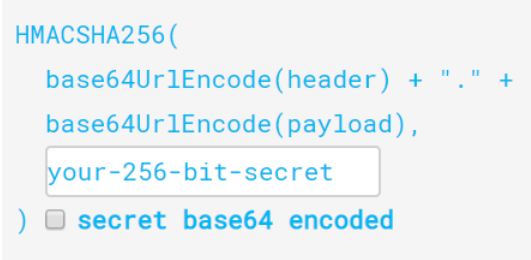

Figura 2.6: *JWT: Signature*[5]

2.5. Arquitectura REST

REST, acrónimo de **RE**presentational **S**tate **T**ransfer, describe un estilo arquitectónico que permite la definición y el direccionamiento de los recursos sin estado, principalmente mediante el uso de Identificadores Uniformes de Recursos (URI) y HTTP. REST es un estilo de arquitectura para el diseño de Servicios Web o Web Services (WS). Este estilo permite aprovechar al máximo los recursos de la web, consiguiendo así ser más ligero y mejorar tanto eficiencia como rendimiento frente a otras opciones como SOAP[73]. Varios estudios señalan que los desarrolladores han pasado de *Simple Object Access Protocol* (SOAP) a los servicios REST, como el medio para que los consumidores usen sus servicios.

Los formatos usados para llevar la comunicación entre el cliente y el servidor se encuentran en XML y JSON. JSON de sus siglas en inglés (*JavaScript Object Notation*)[16], y es una forma de almacenar información de forma organizada y de fácil acceso. En pocas palabras, nos da una colección legible de datos que podemos acceder de una manera realmente lógica, lo caracterizan su sencillez y facilidad de uso, como así también compacto.

La arquitectura REST fue introducida en el año 2000 por el Dr. Roy Thomas Fielding. Él es uno de los principales autores del protocolo HTTP[13], de igual forma es co autor del Apache Web Server[3] que se basan en los principios que respaldan la World Wide Web (WWW) o red de redes[53].

De acuerdo con los principios REST, las interfaces se basan exclusivamente en un URI (*Uniform Resource Indentifiers*)[23] para la detección, interacción de recursos, y transferencia de mensajes generalmente es a través del protocolo HTTP[19]. Un URI de servicio REST solo proporciona la ubicación y el nombre del recurso, que sirve como un identificador del recurso. Los verbos HTTP predefinidos se utilizan para definir el tipo de operación que se debe realizar en el recurso seleccionado.

2.5.1. Principios REST

Según los principios básicos basados en la arquitectura REST del Dr. Roy Thomas Fielding los define de la siguiente forma[8]:

Cliente/Servidor. Establece que el servidor tiene recursos y el cliente desea interactuar con esos recursos.

Stateless. Cada petición del cliente contiene toda la información necesaria para que el servidor responda y esta se trata de forma independiente.

Esto representa varias mejoras para la arquitectura como se muestra en la siguiente lista:

- *Visibilidad*: La supervisión del sistema se simplifica cuando toda la información requerida está dentro de la petición.
- *Escalabilidad*: Al no tener que almacenar datos entre las peticiones, el servidor puede

liberar recursos más rápidamente.

- *Fiabilidad*: El sistema es independiente, dado que puede recuperarse de un fallo.
- *Fácil implementación*: Escribir código que no tiene que manejar datos de estado almacenados a través de múltiples servidores.

Cacheable o no cacheable (reutilizables). Para mejorar la escalabilidad, mientras tanto el cliente tiene capacidad de almacenamiento local. Al almacenar las respuestas, tiene beneficios que se añaden a la arquitectura de lado del servidor. Como por ejemplo interacciones a una base de datos.

Interfaz Uniforme

- *Re direccionamiento de recursos*: cada servicio REST tiene una URI[23] como una única dirección.
- *Manipulación de recursos a traves de representaciones*: un cliente tiene suficiente información para manipular un recurso.
- *Mensajes autodescriptivos*: las peticiones y respuestas deben ser autodescriptivas, por ejemplo usar los operaciones con HTTP para manipular recursos.
- *Hypermedia As The Engine Of Application State* (HATEOAS): los mensajes del servidor deben proporcionar enlaces a recursos relacionados.

Sistema en Capas. Requiere la capacidad de agregar o quitar intermediarios en tiempo de ejecución sin alterar el sistema mejorando su funcionalidad.

Código bajo demanda. Permite que la lógica dentro de cliente (como navegadores web) se actualice independientemente de la lógica del servidor utilizando un código ejecutable enviado por los proveedores del servicio a los consumidores.

2.5.2. Recursos

Los principales bloques de construcción de una arquitectura REST son los recursos. Cualquier cosa que pueda ser nombrada como (una página web, una imagen, una persona, un informe del servicio meteorológico, etc.) se considera un recurso. Los recursos definen

de qué tratarán los servicios, el tipo de información que será transferida y sus acciones relacionadas.

2.5.3. Representaciones

Una representación es un conjunto de bytes, y algunos meta-datos que describen estos bytes. Un solo recurso puede tener más de una representación.

A continuación se muestra una lista donde se muestra un informa de un día nublado en la cual regresaría la siguiente información.

- Fecha a la que se refiere el informe.
- Temperatura máxima del día.
- Temperatura mínima del día.
- Unidad de temperatura a utilizar.
- Porcentaje de humedad.
- Código que indica cuándo esta nublado en el día(por ejemplo, alto, medio, bajo).

Ahora la estructura del recurso está definida, aquí hay algunas posibles representaciones del mismo recurso:

Representación JSON

```
{"reportDay":
    {
      "date": "2020-10-25",
      "max_temp": "25.5",
      "min_temp": "10",
      "temp_unit": "C",
      "percent_hum": "75.5",
      "prom_cloudy": "25.5"
    }
}
```

Representación XML

```
<?xml version="1.0" encoding="UTF-8"?>
<reportDay>
    <fecha></fecha>
    <max_temp></max_temp>
    <min_temp></min_temp>
```

```
    <temp_unidad></temp_unidad>
    <percent_unit></percent_unit>
    <percent_hum></percent_hum>
    <percent_cloudy></percent_cloudy>
</reportDay>
```

2.5.4. Acciones

La implementación de una arquitectura REST también dependen de la noción de un conjunto de operaciones limitadas que tanto el cliente como el servidor entienden desde el comienzo de la comunicación.

En el protocolo HTTP, las operaciones seis operaciones más utilizadas se muestran en la tabla 2.1.

Operación	Descripción
GET	Devuelve la información que se haya identificado mediante el URI de solicitud
PUT	Solicita que la entidad adjunta se almacene en el URI de solicitud suministrado.
POST	Solicita que el servidor de origen acepte la entidad adjunta en la solicitud como un nuevo subordinado del recurso identificado por el URI de solicitud.
DELETE	Solicita que el servidor de origen elimine el recurso identificado por el URI de solicitud.

Tabla 2.1: *Operaciones sobre HTTP*

Las aplicaciones que siguen la arquitectura REST y emplean en el protocolo HTTP comprenden y siguen estas reglas rigurosamente. Estas acciones podrían ser cualquier cosa, al igual que el tipo de recursos que maneja el sistema. Aún así, hay un conjunto de acciones comunes que cualquier sistema que sea orientada a recursos debería ser capaz de proporcionar: acciones CRUD (*create*, *retrieve*, *update*, *delete*) al español se definirian como (crear, recuperar, actualizar y borrar) .

2.5.5. Códigos de estado en HTTP

Los códigos de estado HTTP describen de forma abreviada la respuesta HTTP del servidor al cliente. Estos códigos están especificados en el RFC7231[20]. Cabe mencionar de igual forma que el Internet Assigned Numbers Authority[12] (IANA) mantiene el registro oficial de los códigos de estado HTTP.

El primer dígito del código de estado especifica uno de los 5 tipos de respuesta, el mínimo para que un cliente pueda trabajar con HTTP es que reconozca estas 5 clases como se muestra en la siguiente lista:

1. **1xx**: Informativo y sólo definido bajo HTTP/1.1.
2. **2xx**: La petición fue exitosa, se recibe el contenido solicitado.
3. **3xx**: El recurso no existe o fue trasladado a otro lugar.
4. **4xx**: La fuente de la solicitud ocurrió un error.
5. **5xx**: El servidor se caído debido a algún error en su código.

2.6. OWASP

La OWASP (Open Web Application Security Project)[51], surgió en internet el 1 de diciembre de 2001 y se estableció como una organización benéfica sin ánimo de lucro en los Estados Unidos el 21 de abril de 2004 para asegurar la disponibilidad y el apoyo continuo en la seguridad del software.

La OWASP es una comunidad abierta dedicada a permitir que las organizaciones conciban, desarrollen, adquieran, operen y mantengan aplicaciones en las que se pueda confiar. Todas las herramientas, documentos, foros y capítulos de la OWASP son gratuitos y están abiertos a cualquier persona interesada en mejorar la seguridad de las aplicaciones[46]. Su misión se basa en enfocar la seguridad como un problema de las *personas*, los *procesos* y la *tecnología*, porque los enfoques más efectivos para la seguridad de las aplicaciones incluyen mejoras en todas estas áreas.

2.7. OWASP Cheat Sheet Series Project

OWASP Cheat Sheet Series[43] es una colección de documentos[42] proporcionados por la OWASP con alto valor técnico en temas específicos de seguridad en aplicaciones web. Este proyecto está dirigido por Jim Manico[26] y Eliee Saad[9]. Dentro de ésta existe una serie de medidas de seguridad relativas a la seguridad REST[50], aplicado al uso del protocolo HTTP.

2.7.1. REST Security Cheat Sheet

REST Security Cheat Sheet[36] es un documento en el que se publican algunas estrategias y técnicas de implementación y desarrollo de aplicaciones REST, que pueden aplicarse a cualquier proyecto, mejorando la calidad en seguridad de las aplicaciones.

2.8. Estudio de OWASP API Security Project

El OWASP API Security Project[38] busca aportar valor a desarrolladores de software y pentesters, subrayando los riesgos potenciales en las API inseguras e ilustrando cómo estos riesgos pueden mitigarse[49].

El proyecto está diseñado para colaborar con un gran número de organizaciones, que estén desarrollando e implementando las API potencialmente sensibles. Estas API se utilizan para tareas internas y/o para interactuar con terceros, muchas veces sometiéndose a las rigurosas pruebas de seguridad que las protegerían contra ataques.

2.8.1. OWASP API Security Top 10

La idea de "OWASP API Security Top 10"[44] es crear y mantener un documento que permitirá orientar aquellos que se involucran en el desarrollo y mantenimiento de las API por ejemplo desarrolladores, diseñadores, arquitectos, administradores y organizaciones. Dentro de este proyecto se remarca que en los últimos años, la arquitectura de las aplicaciones ha cambiado significativamente.

Según en este documento se recopilaron, revisaron y analizaron datos disponibles públicamente sobre incidentes de seguridad en las APIs. Categorizando por un grupo de expertos en seguridad.

Para llevar cabo el análisis de riesgos se menciona el uso de OWASP Risk Assessment Framework[37], en la cual a través de los resultados estadísticos se generó una lista de cuatro componentes como se muestra en la figura 2.7. Y se establecen cuatro niveles de impacto en una API, que va desde la *explotabilidad*, *debilidad prevalente*, *debilidad detectable*, *impacto técnico* e *impacto en el negocio*, dado que todos siguen distintos sectores de negocio y sus actividades y la naturaleza de los datos es distinto a cada uno, se han definido distintos valores que dependerá del sector en el que este involucrado, de igual manera como la cantidad de datos gestionados.

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impact	Business Impacts
API Specific	Easy: **3**	Widespread **3**	Easy **3**	Severe **3**	Business Specific
	Average: **2**	Common **2**	Average **2**	Moderate **2**	
	Difficult: **1**	Difficult **1**	Difficult **1**	Minor **1**	

Figura 2.7: *Riesgos de seguridad en una API según la OWASP*

Como sugerencia de seguridad aplicable a cualquier ambiente de producción, se menciona que no siempre se aplica a los detalles técnicos, siendo este un factor de vulnerabilidad particular. Llevando a cada organización a tomar la importancia sobre la seguridad de riesgos en sus aplicaciones.

Este proyecto fue dado a conocer en mayo de 2019 en el evento **OWASP Global AppSec Tel Aviv**[46], desde entonces, ha estado disponible en GitHub para discusión pública y futuras contribuciones[45].

2.8.2. OWASP API Security Top 10 2019

Dentro de este proyecto se enumeran diez vulnerabilidades de seguridad en una API, con el nivel de impacto que tiene sobre una API y de igual forma se han elegido los dos primeros

niveles de seguridad vulnerables; entre los que encuentra la *autorización* y *autenticación* como se muestra en la tabla 2.2.

Nivel de Seguridad	Escenario
API1:2019 – Autorización	Las APIs tienden a exponer los endpoints que manejan los identificadores de los objetos, creando un problema de control de acceso a nivel de la superficie de un ataque. Los controles de autorización a escala de un objeto deben considerarse en cada función para acceder a una fuente de datos utilizando una entrada del usuario.
API2:2019 – Autenticación	Los mecanismos de autenticación suelen aplicarse incorrectamente, permitiendo a los atacantes comprometer los tokens de autenticación o explotar los fallos de implementación para asumir las identidades de otros usuarios temporalmente o permanentemente. Comprometiendo la capacidad del sistema para identificar al cliente/usuario, y en general la seguridad de la API en general.

Tabla 2.2: *OWASP API Security*

2.8.3. Análisis de autorización

En esta sección se explican de forma particular los posibles escenarios que pueden enfrentarse una API en terminos de *autorización* según el OWASP API Security Top 10, en la cual se define a detalle las generalidades que comprometen al sistema.

Nivel de autorización comprometido

La autorización a nivel de un objeto es un mecanismo de control de acceso que generalmente se implementa a nivel de código para validar que un usuario puede acceder a los recursos que debería tener acceso. Cada *endpoint* de la API recibe un ID de un objeto y realiza cualquier tipo de acción sobre este, por lo que debe implementar comprobaciones de autorización a nivel de objeto. Estas comprobaciones deben validar que el usuario conectado tiene acceso para realizar la acción solicitada en el objeto solicitado. Los errores en este mecanismo generalmente conducen a la divulgación, modificación o destrucción de información no autorizada de todos los datos expuestos.

A continuación se muestran dos escenarios posibles de un ataque y de los mecanismos en el que se pueden prevenir este tipo de vulnerabilidades:

Ejemplos de ataque:

Escenario 1.

Una plataforma de comercio electrónico de tiendas en internet proporciona un listado con las tablas de ingresos de sus tiendas. Al inspeccionar las solicitudes del navegador, un atacante puede identificar los *endpoints* de la API utilizados como fuente de datos para esos gráficos y su patrón (p. ej */shops/shopName/revenue_ data.json*). Usando otro endpoint de la API, el atacante puede obtener la lista de todos los nombres de las tiendas.

Con un simple script puede manipular los nombres en la lista, reemplazando `shopName` en la URL, el atacante obtiene acceso a los datos de ventas de miles de tiendas de comercio electrónico.

Escenario 2.

Al escanear el tráfico de la red, un dispositivo portátil hace la petición de `PATCH HTTP` llama la atención de un atacante debido a la presencia de una cabecera `HTTP` personalizada como `X-User-Id`: `54796`. Reemplazando el valor `X-User-Id` con `54795`, el atacante recibe una respuesta `HTTP` exitosa y puede modificar los datos de la cuenta de otros usuarios.

Prevención de ataques

- Implementar un mecanismo de autorización adecuado que se base en las políticas y la jerarquía de los usuarios.
- Usar un mecanismo de autorización para verificar si el usuario conectado tiene acceso para realizar la acción solicitada.
- Es recomendable utilizar valores *aleatorios* e impredecibles como Identificador Único Global, (***Globally Unique Identifier, GUID***) por cada **ID** de registros.

2.8.4. Análisis de autenticación

Los *endpoints* y los flujos de autenticación son activos que deben protegerse. "*Olvidar la contraseña o restablecerla*" debe tratarse de la misma manera que los mecanismos de autenticación.

Una API es vulnerable sí:

1. Permite el relleno de credenciales mediante el cual el atacante tiene una lista de nombres de usuario y contraseñas válidas.
2. Permite a los atacantes realizar un ataque de fuerza bruta en la misma cuenta de usuario, sin presentar un mecanismo de bloqueo de cuenta/captcha.
3. Permite contraseñas débiles.
4. Envía detalles confidenciales de autenticación, como tokens de autenticación y contraseñas en la URL.
5. No valida la autenticidad de los tokens.
6. Acepta tokens JWT sin firmar, con signo débil ("alg" : "none") o no valida su fecha de vencimiento.
7. Utiliza texto sin formato, contraseñas no cifradas o con hash débil.
8. Utiliza claves de cifrado débiles.

Ejemplos de ataque

Escenario 1

El relleno de credenciales (usando diccionarios/lista de nombres de usuario y contraseñas conocidas) es un ataque común. Si una aplicación no implementa protección automatizada contra amenazas o relleno de credenciales, la aplicación se puede usar como un generador de contraseñas (probador) para determinar si las credenciales son válidas.

Escenario 2

Un atacante inicia el proceso de recuperación de contraseña emitiendo una solicitud POST a */api/system /verification-codes* y proporcionando el nombre de usuario en el cuerpo

de la solicitud. Seguido de esto, se envía un token vía SMS con 6 dígitos al teléfono de la víctima. Debido a que la API no implementa una política de limitación de velocidad (*rate limitations*), el atacante puede probar todas las combinaciones posibles utilizando un *script* de subprocesos múltiples, contra el punto final */api/system/verification-codes/smsToken* para descubrir el token correcto en unos pocos minutos.

Prevención de ataques

Para prevenir el ataque de autenticación del usuario se proponen las siguientes recomendaciones desde el punto de vista del nivel de acceso.

1. Se debe de conocer todos los flujos posibles para autenticarse en la API, como enlaces web profundos que implementan la autenticación con un solo clic, etc.
2. Es recomendable instruirse sobre los mecanismos de autenticación. El desarrollador debe entender qué y cómo se usan.
3. Se deben aplicar las tecnologías existentes en autenticación que sean más convenientes para la generación de tokens y almacenamiento de contraseñas.
4. Los *endpoints* de recuperación de credenciales deben tratarse como *rutas* de inicio de sesión en términos de fuerza bruta, limitación de velocidad y protecciones de bloqueo.
5. Es recomendable usar la *Cheat Sheet* de autenticación por la OWASP[39].
6. Se recomienda implementar la autenticación multifactor.
7. Como buena práctica de lado del servidor se recomienda implementar un mecanismo de bloqueo/captura de cuentas para evitar ataques de fuerza bruta contra usuarios específicos.
8. Las claves de una API no deben usarse para la autenticación de usuarios, sino para la autenticación de la aplicación y/o proyecto del cliente.

Capítulo 3

Descripción del entorno

"Qui desiderat pacem, praeparat bellum. (Quien quiera paz, que se prepare para la guerrra.)". – VEGECIO, siglo IV d.C

En éste capítulo se realiza una descripción del entorno de trabajo. El entorno de trabajo se definió de acuerdo a los elementos que interactúan en la comunicación de la API, así como las herramientas y/o tecnologías que se emplearon tanto en el desarrollo y la auditoría de la API creada.

3.1. Diagrama de comunicación IoT

En esta sección se lleva a cabo una introducción relativa a los componentes que interactúan desde la adquisición de los datos hasta su almacenamiento.

En la figura 3.1 se muestran los elementos que integran la comunicación con la API.

Los clientes principales son Node RED, Firefox, Postman. Node RED tiene la función principal de recibir los datos del SensorTAG y enviarlos por peticiones POST autorizado con un JWT hacia el servidor.

Mientras tanto Firefox y Postman funcionan como herramientas auxiliares de desarrollo y pruebas de funcionamiento. La aplicación y la base de datos se encuentran en el lado del servidor con sus esquemas correspondientes.

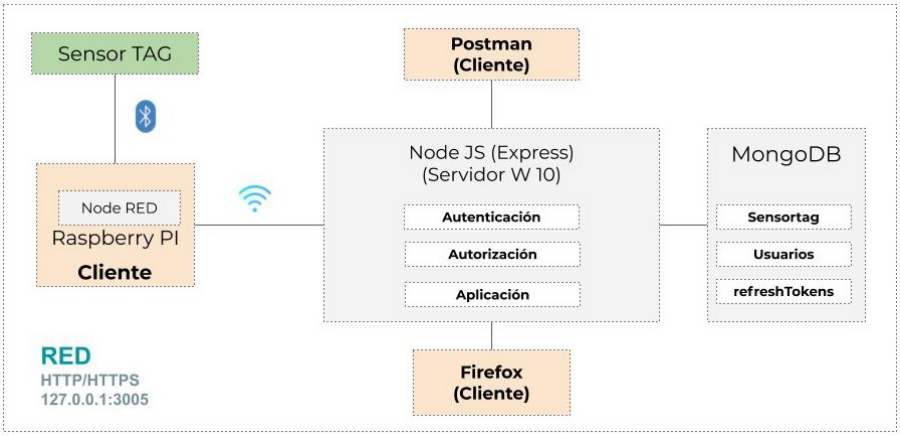

Figura 3.1: *Diagrama de comunicación de la API implementada*

3.2. Aspectos hardware

El ordenador Lenovo T460s fue usado para el desarrollo, auditoría e implementación de la API. Cuenta con una RAM de 20 GB, CPU Intel (R) Core (TM) i7-6600U 2.60 a 2.81 GHz, con un sistema operativo Windows 10, compilación 1909.

De igual forma se hizo uso de una Raspberry Pi 3, la cual recibe los datos del sensotag CC2650 vía BLE.

3.2.1. SensorTAG CC2650

El Sensortag CC2650 es un sensor creado por la empresa *Texas Instruments* para entornos IoT. Este dispositivo integra tecnología inalámbrica Bluetooth Low Energy (BLE), que permite enviar y recibir datos. Este nodo sensor funciona con bajo consumo eléctrico, lo cual garantiza un buen funcionamiento a largo plazo.

Sus características principales se muestran en la siguiente lista:[65]

- Integra 10 sensores de bajo consumo.
- Nodo programable en C, permite la integración de otros sensores y/o actuadores.
- Múltiples estándares de comunicación soportados: 6LowPAN, BLE, Zigbee.
- Acceso seguro a la nube.

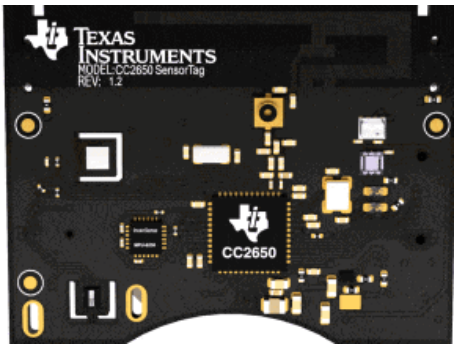

Figura 3.2: *SensorTag CC2650: Placa PCB*[65]

3.2.2. Raspberry Pi

Una placa Raspberry Pi es un ordenador de bajo coste y consumo energético construida por la *Raspberry Pi Foundation*[68]. Este dispositivo permite el desarrollo de proyectos enfocados a la robótica, automatización, IoT, etc.

El modelo a usar en este Trabajo Fin de Máster es una Raspberry Pi 3[67], la cual integra Wi-Fi, Bluetooth, Bluetooth Low Energy (BLE), un puerto Ethernet 10/100, 1 GB de memoria RAM, un CPU de 4 núcleos, así como una salida HDMI. Una vista de este dispositivo se muestra en la figura 3.3. El sistema operativo que ejecuta la *Raspbian* esta basado en Debian.

Figura 3.3: *Raspberry Pi 3 Model B*

3.2.3. Bluetooth Low Energy (BLE)

Es una tecnología inalámbrica de bajo consumo energético y de corto alcance. BLE también conocido como Bluetooth Smart es parte del desarrollo de Bluetooth 4.0. No está diseñado para garantizar alta velocidad, pero sí para un bajo consumo. BLE originalmente fue un proyecto desarrollado por Nokia.

3.3. Aspectos software

Para el desarrollo de la API en el lado del servidor, se ha usado Node.js[55] en su versión v12.13.1. y el administrador paquetes de node (*npm*) en su versión 6.12.1.

Se utilizó Postman[52] en su versión 7.31.1, para realizar las pruebas en el lado del cliente. Tanto en las peticiones y respuesta del servidor.

3.3.1. Node.js

Node.js fue creado en 2009 por Ryan Dahl[54] y patrocinado por Joyent, la empresa para la que Dahl trabajó. Node.js utiliza el motor Google V82 para ejecutar código JavaScript en el lado del servidor.

En términos de desarrollo las particularidades de Node.js, que constituye algo más que un lenguaje de programación se definen en la siguiente lista:

1. *Programación asíncrona*: Ésta es una de las grandes virtudes de Node.js.
2. *E/S asíncrona*: Aplicado a las aplicaciones de entrada/salida de gran volumen.
3. *Simplicidad*: Node.js tiene soporte específico para construir servidores web.
4. *Administrador de paquetes de Nodo: npm, (node package manager)*[31].
5. *Integración*: Permite la interacción con sistemas o servicios en formato JSON, así como con otras API, bases de datos; etc.

3.3.2. Sistema de base de datos NoSQL

Para la implementación de la API en un sistema IoT, es de gran importancia el almacenamiento de los datos para administrarlos. En el presente trabajo se propone el desarrollo de una API que administre los datos de los clientes, los sensores y controlar el acceso de los clientes, por lo que es necesario llevar un registro adecuado al comportamiento y flujo de datos en el lado del servidor y sobre el cliente.

Para tomar en cuenta la selección de la base de datos se presenta la siguiente lista en la que se hace un descripción general o características que conlleva elegir una base de datos.

1. *Desarrollo rápido*. El proceso para ir de manera rápida.
2. *Facilidad al cambio de esquemas*. Es recomendable ajustarlo a posibles cambios para eliminar o agregar campos.
3. *Capacidad para manejar las relaciones entre entidades*. Esto significa almacenamiento de claves/valores.
4. *Integración fluida entre el código de las entidades y la representación de los datos en la base de datos*.

MongoDB

En base de datos NoSQL se almacenan los datos en documentos o colecciones en formato JSON. Este tipo de base de datos tiene el potencial de guardar gran cantidad de este tipo de colecciones. MongoDB es de uso abierto o gratuito lo cuál no requiere uso de licencias. Este tipo de base se datos es adaptable a sistemas distribuidos, lo cuál garantiza la alta fiabilidad y disponibilidad, así como también su escalabilidad horizontal, se pueden distribuir en Clústers a través de Internet una red global, lo cual permite mantener backups en tiempo real.

Adaptado a lo antes mencionado se presentan las características principales destacadas de MongoDB:

1. **Integración rápida**: En MongoDB se tiene Mongoose.js[28], que permite abstraer el código del motor, simplificando las tareas a la hora de definir los esquemas, validacio-

nes, etc.

2. **Facilidad al cambio de esquemas**: MongoDB es la elección perfecta para los proyectos dinámicos, dado que al hacer un cambio en el esquema es tan simple como hacer los cambios en el código, y en éste no se requiere ninguna migración.
3. **Manipulación de relaciones con entidades**. MongoDB tiene principalmente la característica de los subdocumentos.

MongoDB es una alternativa versátil a la hora de crear un prototipo y algo nuevo, algo que puede cambiar durante el proceso de desarrollo muchas veces. De esta manera existe la seguridad adicional de que si necesitamos cambiar algo, como adaptar nuestro modelo de datos a una nueva estructura, podemos hacerlo fácilmente y con un impacto menor.

3.3.3. ZAP

El Zed Attack Proxy (ZAP) es una herramienta de prueba de pentesting gratuita y de código abierto que se mantiene bajo la Open Web Application Security Project (OWASP). ZAP está diseñado específicamente para probar aplicaciones web. ZAP es lo que se conoce como un "man-in-the-middle proxy" el funcionamiento básico se muestra en la figura 3.4. ZAP se encuentra entre el navegador del tester y la aplicación web para que pueda interceptar e inspeccionar los mensajes enviados entre el navegador y la aplicación web, modificar el contenido si es necesario, y luego enviar esos paquetes a su destino. Puede ser usado como una aplicación independiente, o como un proceso de demonio.

Figura 3.4: *ZAP: main-in-the-middle proxy*

ZAP ofrece una serie de funcionalidades para un amplio rango de niveles de destreza, desde desarrolladores, pentesters principiantes, pasando por especialistas en seguridad.

ZAP tiene versiones para cada uno de los sistemas operativos y Docker. Las funcionalidades adicionales están disponibles gratuitamente en una variedad de extensiones en ZAP Marketplace, accesibles desde el cliente ZAP.

ZAP es open-source. Su código fuente puede ser examinado con el fin de ver exactamente cómo se implementa la funcionalidad. Cualquier miembro puede ser voluntario para trabajar en ZAP para corregir errores, añadir características, crear requerimientos de acceso para obtener correcciones en el proyecto, y/o diseñar complementos para apoyar situaciones especiales.

Requerimientos de ZAP

ZAP tiene instaladores para Windows, Linux y Mac OS/X. También hay imágenes de Docker disponibles en el sitio de descarga[41].

ZAP requiere Java 8+ para funcionar. El instalador de Mac OS/X incluye una versión apropiada de Java si bien es necesario instalar Java 8+ por separado para las versiones de Windows, Linux y Cross-Platform. Las versiones Docker no requieren la instalación de Java.

Interfaz gráfica de ZAP

La interfaz gráfica por la cual esta compuesta ZAP se muestra en la figura 3.5. Los elementos a continuación se describen:

1. *Menú*: Permite el acceso a las herramientas automatizadas y manuales.
2. *Barra de herramientas*: Incluye botones que proporcionan fácil acceso a las funciones más utilizadas.
3. *Árbol de sitios o* `scripts`: Muestra el árbol de los sitios y el árbol de las scripts.
4. *Ventana de trabajo*: Muestra las peticiones, sus respuestas y scripts que le permiten editarlos.
5. *Estado de los procesos*: Muestra detalles de las herramientas automáticas y manuales.
6. *Estado de los servicios*: Muestra el estado de las alertas encontradas de las principales herramientas automáticas.

De igual forma en la figura 3.7 se muestra la *Venta de trabajo*. En esta se encuentran los botones para el escaneo automático y manual.

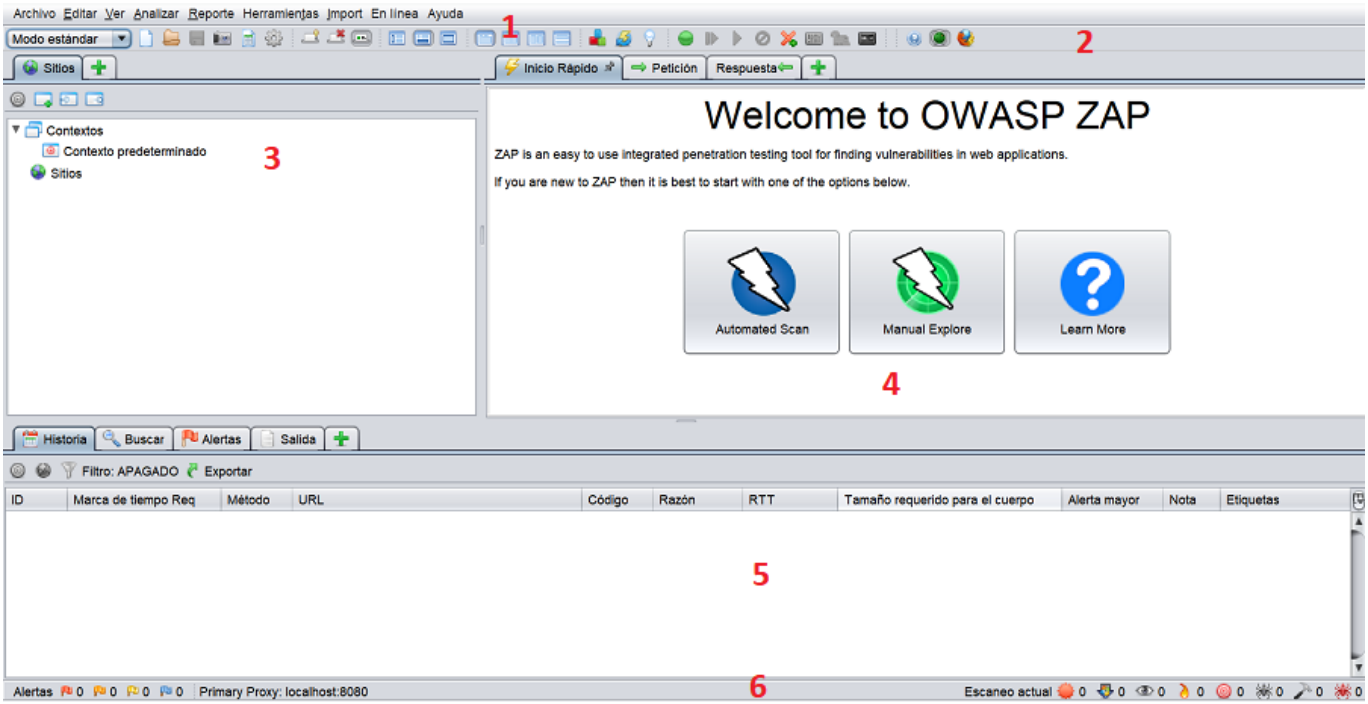

Figura 3.5: *ZAP: Interfaz gráfica*

ZAP proporciona dos métodos para el rastreo de aplicaciones web. Esto se puede observar en la figura 3.6 como *Automated Scan* y *Manual Explore*. El método tradicional de ZAP descubre los enlaces examinando el HTML en las respuestas de la aplicación web. Éste método es más rápido, pero no siempre es efectivo cuando se explora una aplicación web hecha con AJAX, estás generan enlaces usando JavaScript.

Para las aplicaciones AJAX, usando el método de AJAX en ZAP es más efectiva. Esté explora la aplicación web invocando a los navegadores que luego siguen los enlaces que se han generado. El método con AJAX es más lento que el proceso tradicional y requiere una configuración adicional para su uso en un entorno "headless".

ZAP escanea de forma pasiva todas las peticiones y respuestas que le son enviadas a través de ella. Sin embargo, ZAP sólo realiza escaneos pasivos de la aplicación web. El escaneo pasivo no cambia las respuestas de ninguna manera y se considera seguro. El escaneo también

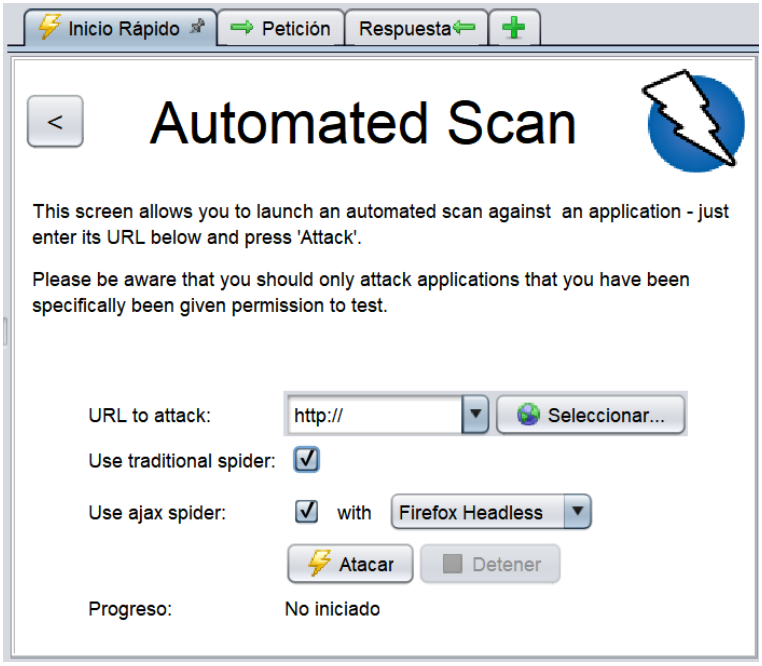

Figura 3.6: *Escaneo automático*

se realiza en segundo plano para no retardar el proceso de la exploración. La exploración pasiva es buena para encontrar algunas vulnerabilidades y como una forma de obtener el estado básico de seguridad de una aplicación web y localizar dónde es posible que se requiera un estudio más profundo.

El escaneo activo, sin embargo, intenta encontrar otras vulnerabilidades utilizando ataques conocidos contra los objetivos seleccionados.

Escaneo automático

Para generar un escaneo automático se hace clic en la pestaña *Inicio Rápido*. Inicio rápido es un complemento de ZAP que se incluye automáticamente al instalar ZAP.

Para ejecutar el *Escaneo automático* se realizan los siguientes pasos:

1. Ejecutar ZAP y hacer clic en la pestaña *Inicio rápido* en la ventana de trabajo.
2. Se hace clic en el botón **Automated Scan**.
3. En el espacio de la **URL** a la que se va a atacar, se debe introducir la IP o nombre de dominio de la aplicación web.

4. Dar clic en el botón **Atacar**

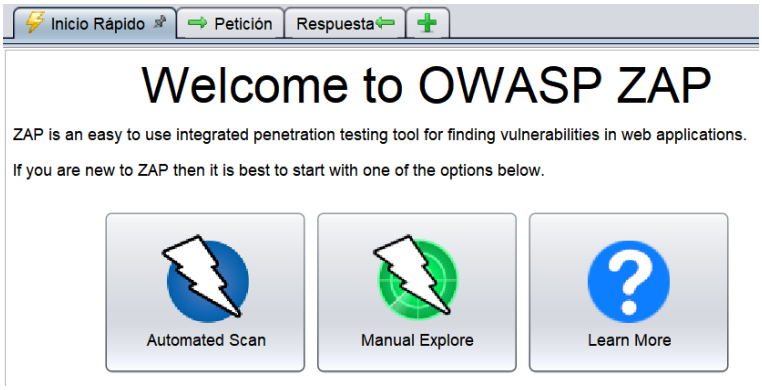

Figura 3.7: *ZAP: Escaneo automático y escaneo manual*

ZAP procederá a rastrear la aplicación web y escaneará pasivamente cada página que encuentre. Después ZAP usará el escaneo activo para atacar todas las páginas descubiertas, las funciones y parámetros.

Explorar una aplicación manualmente

El escaneo pasivo y la opción de ataque automatizado es una buena herramienta para comenzar a evaluar las vulnerabilidades de las aplicaciones web, pero tiene algunas limitaciones. Entre ellas están:

- Una página protegida por login no se puede descubrir durante un escaneo pasivo porque, a menos que haya configurado la funcionalidad de autenticación de ZAP, ZAP no gestionará la autenticación solicitada.
- En un escaneo pasivo no se tiene mucho control sobre la secuencia de exploración o el tipo de ataques que se realizan en un análisis automatizado. ZAP ofrece muchas opciones adicionales para la exploración y los ataques fuera del escaneo pasivo.

Los métodos son una gran manera de explorar el sitio web, pero deben combinarse con la exploración manual para ser más eficaces.

ZAP escanea de forma pasiva todas las peticiones y respuestas hechas durante la exploración de vulnerabilidades. Seguido de esto se continúa construyendo el árbol del sitio y

registra alertas de posibles vulnerabilidades encontradas durante la exploración. En la figura 3.8 se muestra la *URL a escanear*, el navegador en el que se van hacer la exploración, y el HUD(Heads Up Display) en caso de querer ejecutar ZAP mediante una interfaz gráfica en el navegador.

ZAP examina cada página de la aplicación web, ya sea que esté enlazada a otra página o no, en busca de vulnerabilidades.

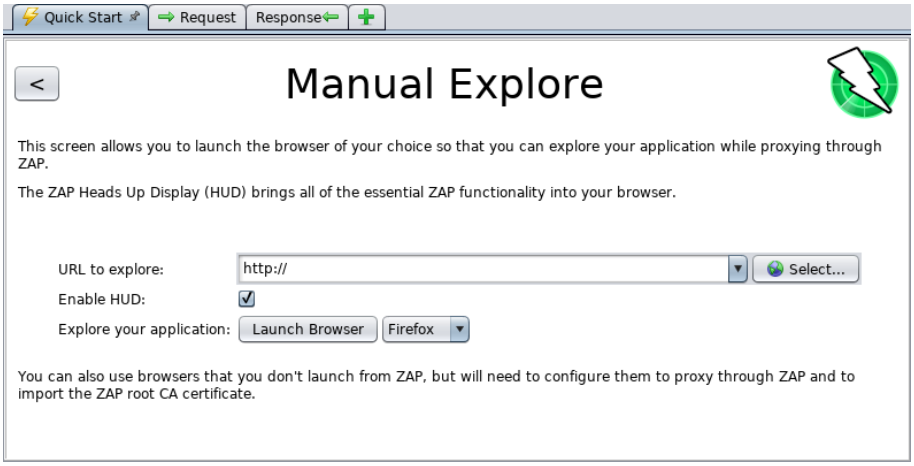

Figura 3.8: *Búsqueda manual de vulnerabilidades*

3.3.4. Helmet

Helmet[2] es un middleware usado para proteger las cabeceras de la aplicaciones desarrolladas en Nodejs basadas en el framework *Express*. Ésta contiene once funciones que protegen la cabecera de una petición o solicitud hacia un endpoint. Estas funciones ayudan a mitigar las vulnerabilidades de seguridad conocidas y asegurar las cabeceras de una petición y/o respuesta de una aplicación.

Entre las once funciones que caracterizan a Helmet, `app.use(helmet());` se encarga de incorporar estas las funciones por defecto.

Los principales middleware que se pueden configurar en la aplicación están relacionadas con `Content-Security-Policy` que ayuda a mitigar ataques *Cross-Site Scripting* y `X-XSS-Protection`.

3.3.5. Node-RED

Node-RED[34] es una herramienta visual y de código abierto desarrollada en Node.js creada por IBM en conjunto con OpenJS Foundation. Permite añadir dispositivos a una red y gestionarlos sin necesidad de tener grandes conocimientos de programación. Además, se trata de una herramienta muy ligera, pudiéndose ejecutar en dispositivos con recursos limitados como una Raspberry Pi.

Node-RED proporciona una interfaz basada en HTML que permite crear flujos de eventos e interconectarlos todos ellos a través de un ligero entorno como se muestra en la figura 3.9. La interfaz muestra visualmente las relaciones y las funciones, pudiendo añadir o eliminar nodos y conectarlos entre sí con el fin de hacer que se comuniquen entre ellos.

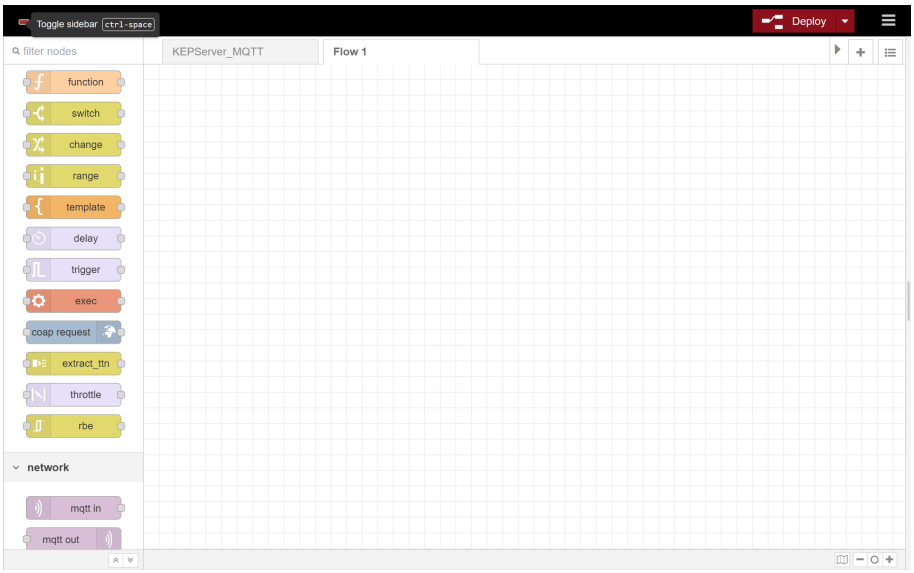

Figura 3.9: *Node-RED*

Node-RED está compuesto principalmente por una paleta de nodos y una ventana de edición, así como su panel de ayuda que contiene el *dashboard*, *debug messages*, *help*, *information*.

De igual forma NodeRED trabajará de la mano del Sensortag. El flujo de NodeRED se ejecutará en la versión 1.1.13.

3.3.6. OpenSSL

OpenSSL[35] es una suite que permite crear, firmar y gestionar certificados, tanto en el lado del servidor como del lado del cliente (sistema embebido). La idea es comenzar creando una Autoridad de Certificación propia, lo cual permite firmar y gestionar los certificados.

Una vez que la Autoridad Certificadora está definida, los certificados que sean necesarios, tanto para el servidor como para cada uno de los clientes se puede crear. Al crear una Autoridad de Certificadora se deben tener a mano algunos datos.

- Se debe elegir un algoritmo para realizar el hash, por ejemplo MD5, SHA-1, SHA-256, etc.
- Se debe elegir un algoritmo para firmar el digest, por ejemplo RSA, DSA, ECC. Al seleccionar uno de estos algoritmos (RSA, ECC, etc) toda la cadena de certificados se firma con el algoritmo elegido.
- Para construir el certificado son necesarios los siguientes datos:
    1. Tamaño de la clave (pe. 2048).
    2. Algoritmo de hash (pe. SHA256).
    3. Tiempo de validez del certificado en días (pe. 3650).
    4. País.
    5. Nombre de la organización.
    6. Nombre del certificado.
    7. Correo electrónico.

Se ha seleccionado esta herramienta open-source para asegurar las comunicaciones entre el cliente y el servidor, así como para firmar los JWT que se generan en la autenticación del cliente.

3.3.7. OpenAPI Initiative

La OpenAPI Initiative (OAI)[66] ha desarrollado la OpenAPI Specification (OAS)[61] que involucra a consumidores, desarrolladores, proveedores y clientes de las API con el objetivo

de definir una interfaz estándar, independiente del lenguaje de programación para las API REST. Google, IBM, PayPal, Intuit, SmartBear, Capital One, Restlet, 3scale y Apigee se involucraron en la creación de la OAI.

La OAS define una interfaz estándar independiente del lenguaje de programación para las API REST que permite que tanto a los humanos como las computadoras descubrir y comprender las capacidades del servicio sin acceso al código fuente, documentación o mediante la inspección del tráfico de la red.

Las herramientas de generación de documentación pueden usar una definición de OpenAPI para mostrar la API, herramientas de generación de código para generar servidores y clientes en varios lenguajes de programación, herramientas de prueba y muchos otros casos de uso.

Versiones y formato

OpenAPI Specification administra sus versiones con base a Semantic Versioning[70]. La versión de `openapi` define la estructura en general de una API a documentar. OpenAPI 3.0 usa versiones semánticas[69] con un número de versión de tres partes.

A continuación se muestra en la figura 3.10 la versión 3.0, así como las secciones principales que la componen.

Cabe destacar que un desarrollador puede escribir las definiciones OpenAPI en YAML o JSON, lo cual da una facilidad y comprensión rápida.

3.4. Adquisición de los datos en Node RED

En Node RED se obtienen los valores de la temperatura que son enviados cada 5s, humedad cada 9s, luminosidad cada 3s y presión cada 8s. En la figura 3.11 se muestra el flujo que permite la conexión BLE con Node RED y el SensorTAG. El flujo que envía los datos se identifican en color crema y las peticiones hacia la API en color café.

Mientras tanto en la figura 3.12 se muestra una función en JavaScript. ´Ésta permite

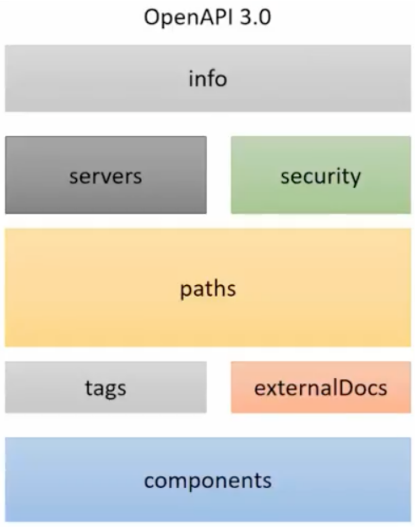

Figura 3.10: *Mapa: OpenAPI 3.0*[66]

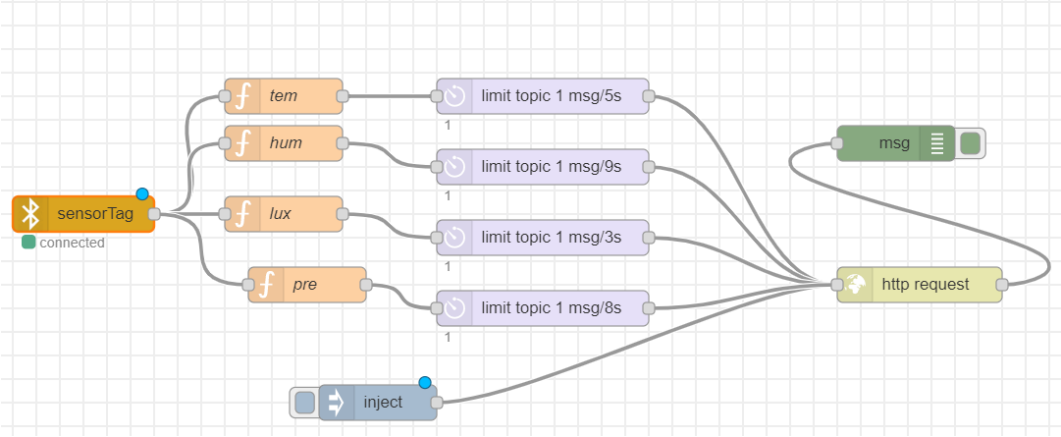

Figura 3.11: *Flujo datos del SensorTag*

obtener la *temperatura* y construir la petición en formato JSON. Los campos que forman el cuerpo de la petición son: *sensorTag*, *valor*, *sensor*, *fecha*, y el *topic*.

En este flujo se configuran los parámetros para realizar las peticiones usando el método `POST`, al endpoint `/api/v1/sensortag`, así como el tipo de seguridad. Para autenticar las peticiones que se añadió de tipo *bearer authentication* y *token*. En la figura 3.13 se muestra el resultado para configurar.

```
1  if (msg.topic === "sensorTag/humidity")
2  {
3      var d = new Date();
4      var tem = msg.payload.temperature;
5      var data = {payload: {
6          "sensorTag":"cc_2650_home",
7          "valor": tem,"sensor":"tem",
8      "fecha":d}, topic: "Temperatura"};
9      return data;
10 }
11 else
12 {
13    return null;
```

Figura 3.12: *Código JS: obtención de la temperatura*

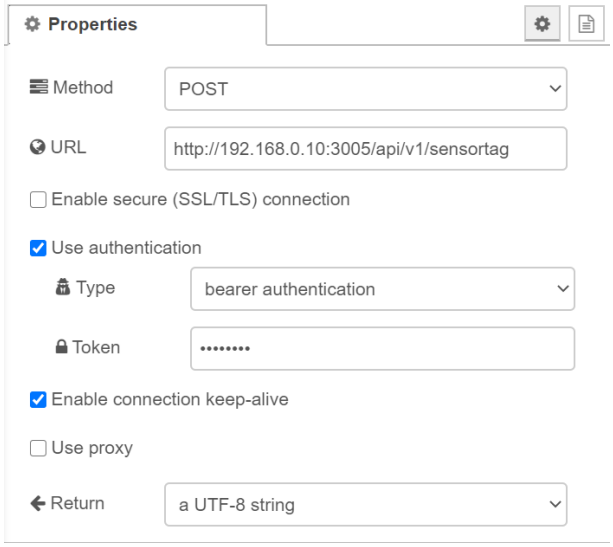

Figura 3.13: *Node RED: configuración de una petición POST*

Capítulo 4

Arquitectura de la API

"La manera más superficial de influir en los demás es mediante la conversación que no tiene nada real detrás de ella. La influencia así producida siempre es insignificante". – I Ching, China, circa siglo VIII a.C.

.

Las APIs (*Application Programming Interface*) constituye un pilar esencial de nuestro mundo conectado. El software utiliza estas interfaces para comunicarse desde aplicaciones en teléfonos inteligentes hasta servidores en *backend*.

En el presente capítulo se muestra la arquitectura de la API de autoría propia de este Trabajo Fin de Máster. Está API tiene objetivo funcional integrar distintas herramientas y/o módulos para su funcionamiento. La API implementada fue desarrollada en Nodejs y el framework *Express*[64]. Cuenta aproximadamente con 1051 líneas de código.

4.1. Contexto general de la API

Se desarrolló una API que en su conjunto permite gestionar *peticiones*, *respuestas*, administra *endpoints* de autenticación, autorización que funcionan como *Middleware* y de datos. También cuenta con un endpoint para conocer el funcionamiento de la API haciendo referencia a la *Documentación*.

En particular se presenta una API de autoría propia que permite el envío de datos de un SensorTAG CC2650 hacia la API aplicación que está interconectada a una base de datos

NoSQL. Del mismo modo permite la autorización y autenticación de usuarios.

Un usuario/cliente puede realizar peticiones al servidor siempre y cuando este autenticado. Lo cual tiene reserva de hacer peticiones de acuerdo a su rol, siendo que el rol de administrador puede tanto *borrar*, *escribir* y *leer* recursos. Mientras tanto un usuario o *guest* solo puede leer.

La autenticación del usuario/cliente registrado es mediante el correo electrónico y una contraseña. Al identificarse en la aplicación, el servidor envía un *refreshToken* que permite el cliente usarlo para generar *accessTokens* y enviarlos en la cabecera de la petición.

4.1.1. Endpoints de la API

La API desarrolla gestiona peticiones y respuestas que escuchan en el puerto 3005 en HTTPS y en el puerto 3006 en HTTP. Los endpoints están divididos en tres esquemas relativos al *usuario*, control de acceso *autenticación*, autorización y los datos del *sensor*.

En la figura 4.1 se muestran las rutas parametrizadas encerradas en un rectángulo rojo, así como también todos los endpoints creados.

```
// Apartado Usuario
api.get('/v1/users', auth.isAuthAdmin, UserCtrl.getUsers)
api.get('/v1/user/:userId',auth.isAuthAdmin, UserCtrl.getUser)
api.post('/v1/signup', UserCtrl.singUp)
api.post('/v1/login', UserCtrl.login)
api.delete('/v1/logout', UserCtrl.logout)

//Crea un nuevo acessToken
api.get('/v1/refreshTokens', UserCtrl.getrefreshTokens)
api.delete('/v1/refreshTokens/:refreshTokenId', auth.isAuthAdmin, UserCtrl.deleterefreshToken)
api.post('/v1/token', UserCtrl.createnewTokenbyRefreshToken)


// Apartado Sensor
api.get('/v1/sensortag', auth.isAuthAdmin, SensorCtrl.getSensors)
api.get('/v1/sensortag/:sensortagId' auth.isAuthNormal, SensorCtrl.getSensor)
api.post('/v1/sensortag', auth.isAuthAdmin,  SensorCtrl.saveSensor)
api.put('/v1/sensortag/:sensortagId', auth.isAuthNormal, SensorCtrl.updateSensor)
api.delete('/v1/sensortag/:sensortagId', auth.isAuthAdmin, SensorCtrl.deleteSensor)
```

Figura 4.1: *Rutas parametrizadas*

4.1.2. Control de acceso en la API

El uso de middleware es muy común para la autenticación, el registro, manejo de la sesión, etc. En la figura 4.2 se muestran tres funciones que administran la autorización y la autenticación para usuarios con roles como administrador y/o usuario de la API creada.

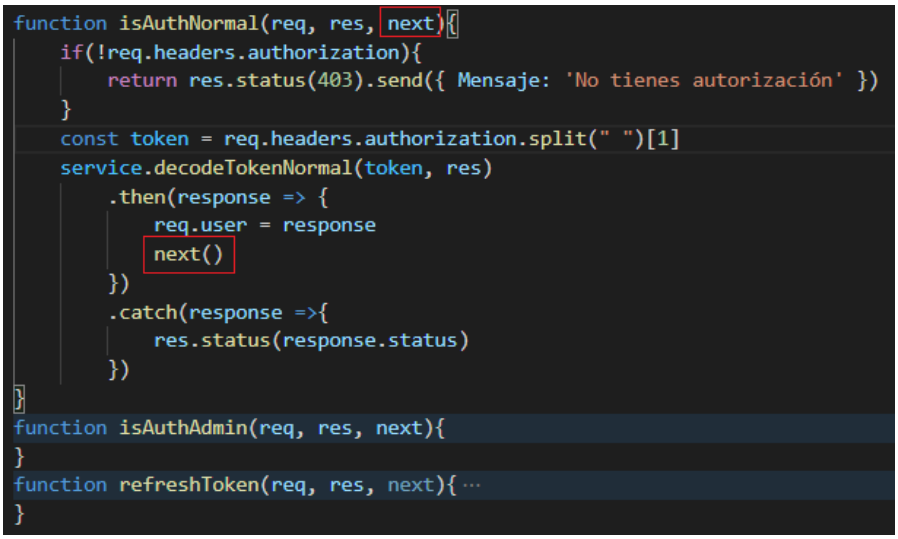

Figura 4.2: *Middleware: autorización*

La función *isAuthNormal* e *isAuthAdmin* en cada petición revisan la cabecera. Ésta debe contener la etiqueta *"authorization"* seguido de un JWT válido para hacer continuar con el proceso de la petición ya sea de escritura y/o lectura.

4.1.3. Documentación de la API

Mantener la documentación actualizada de la interfaz de esta API es crucial. Se tiene como objetivo documentar al sistema y agilizar la interpretación del usuario final.

En el Apéndice C.4 se listan algunos de los módulos analizados que permiten llevar cabo la documentación en una API. Entre los más usados se encuentra *Swagger*[61] que implementa OpenAPI Specification[66]. En este Trabajo Fin de Máster se ha optado por implementar OAS dado su facilidad de uso, la tendencia de implementación como tecnología emergente y su facilidad y adaptación a diferentes lenguajes de programación.

4.1.4. Validación de respuesta y petición en la API

En la API se validan las respuestas y peticiones en formato JSON con el objetivo de asegurar que la ejecución del sistema no se vea afectada por una petición inválida. De igual forma en el Apéndice C.4 se describen algunos módulos relativos a la validación del formato JSON.

4.2. Modelo Vista Controlador de la API

El *Modelo Vista Controlador* sigue una estructura distinta para cada desarrollo. En este desarrollo se describe de forma breve cada una de sus partes.

1. *Controlador*: Maneja las solicitudes y pide a los modelos que tomen acciones adicionales.
2. *Modelos*: Contiene la lógica principal de la API. Consulta la base de datos. Por ejemplo, MongoDB proporciona diferentes mecanismos para añadir comportamientos extra a los modelos.
3. *Vista*: La vista se incrustará dentro del código del modelo en forma de un método que traduce los detalles de un modelo en JSON que puede ser devuelto al cliente.

4.2.1. Estructura del proyecto desarrollado

La API desarrollada tiene el objetivo de administrar peticiones autorizadas y autenticadas. En este contexto la figura 4.3 muestra las carpetas que integran el proyecto. En Node.js la carpeta que contiene cada uno de los frameworks o herramientas de trabajo es `node_modules`, mientras que los archivos de configuración y los datos de desarrollo de la API se muestran en el archivo `packagejson`.

Para la firma de los JWT se tiene la carpeta `asymmetric` que contiene las llaves pública y privada. Los endponits escritos se encuentran en la carpeta `routes`. Para la creación de los JWT se creo la funcón *createToken* y para la validación se tienen las funciones *decodeTokenNormal* y *decodeTokenAdmin* que se encuentra en la carpeta `services`. Los

archivos que contienen la especificación OpenAPI se encuentran en la carpeta `swagger`.

El proyecto se intenta adaptar al *Modelo Vista Controlador*. En la figura 4.3 se presentan los elementos anteriormente mencionados.

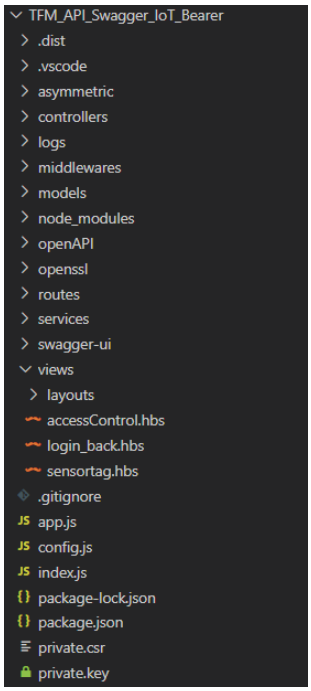

Figura 4.3: *API: carpetas del proyecto*

4.2.2. Configuración del servidor

La configuración hace referencia a las variables correspondientes al servidor. Aquí se encuentran las claves para firmar los JWT y los puertos. La figura 4.4 muestra las variables de configuración.

4.2.3. Modelos de la API

Los modelos hacen referencia a los esquemas creados en MongoDB. En la figura 4.5 se muestran los campos correspondientes del esquema usuarios, así como sus validaciones correspondientes a ésta.

También se tiene el esquema de *sensortag* el cual amacena los datos de los sensores de

```
'use-strict'
const fs = require('fs')
const path = require('path')

const PRI_RSA_KEY = path.join(__dirname, './asymetric/key.pub')
const PUB_RSA_KEY = path.join(__dirname, './asymetric/key.pem')

console.log(PRI_RSA_KEY)

module.exports = {
    PORT_HTTPS: process.env.PORT || 3005,
    PORT_HTTP: process.env.PORT || 3006,
    db: process.env.MONGODB || 'mongodb://localhost:27017/api-rest-iot',
    //Configuración HMAC
    KEY_TOKEN: 'miC',  // 'miCl@vEToK#9'
    REFRESH_KEY_TOKEN: 'miCro',
    REFRESH_TOKENS: [],
    //Configuración Asimetrica
    PRIVATE_RSA_KEY: fs.readFileSync(PRI_RSA_KEY, 'utf-8'),
    PUBLIC_RSA_KEY: fs.readFileSync(PUB_RSA_KEY, 'utf-8')

}
```

Figura 4.4: *API: variables de configuración del servidor*

```
const userSchema = new Schema ({
    email: {type: String, trim: true, select: true, unique: true, index: true,
      lowercase: true, "default": "", validate: [validateLocalStrategyProperty,
      "Ingresa tu email"], match: [/.+\@.+\..+/, "Ingresa un email valido"]},
    name: {type: String, unique: true, select: true, index: true, lowercase: true,
      required: "Completa tu nombre de usuario",
      trim: true, match: [/^[\w][\w\-\ \@]*[\w]$/, "Ingresa un nombre de usuario valido"]},
    pa (property) validate: [(password: any) => boolean, string]
      validate: [validateLocalStrategyPassword, "La contraseña debe ser más grande"]},
    singupDate: { type: Date, default: Date.now()},
    lastLogin: Date,
    role: {type: String, enum: ['admin','user', 'guest']},
    stateAccount: {type: Boolean, default: false},
    accessAttempts: {type: Number, default: 0},
})
```

Figura 4.5: *API: modelo del usuario*

temperatura, humedad, intensidad lumínica y presión. Este modelo se presenta en la figura 4.6, con los campos y datos respectivamente.

Para cada usuario que se autentica, el servidor devuelve un JWT de refresco (*refresh-Token*) como se muestra en la figura 4.7. Este esquema almacena la asignación del JWT de refresco de los clientes autenticados.

```
const SensorTagSchema = Schema ({
    id_user: {type: String, unique: true, select: true, index: true, trim: true},
    sensorTag: String,
    valor:{type: Number, default: 0},
    sensor: {type: String, enum: ['tem','hum','lum','pre']},
    fecha: Date
})

module.exports = mongoose.model('Sensor', SensorTagSchema)
```

Figura 4.6: *API: modelo del sensor*

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema


const refreshTokenSchema = new Schema ({
    id_user: {type: String, unique: true, select: true, index: true, trim: true},
    refreshToken: {type: String, unique: true, select: true, index: true, trim: true}
})

module.exports = mongoose.model('refreshToken', refreshTokenSchema)
```

Figura 4.7: *API: modelo del JWT de refresco*

4.2.4. Vistas de la API

Las vistas tienen como propósito identificar específicamente al usuario, y darle la autorización correspondiente de acuerdo a lo que desea hacer en los *endpoints*. Para crear las vistas en la API se ha usado *hbs*[30] como motor de plantillas. La carpeta que contiene las vistas se muestra en la figura 4.8. Todos los archivos administran sentencias HTML y JavaScript.

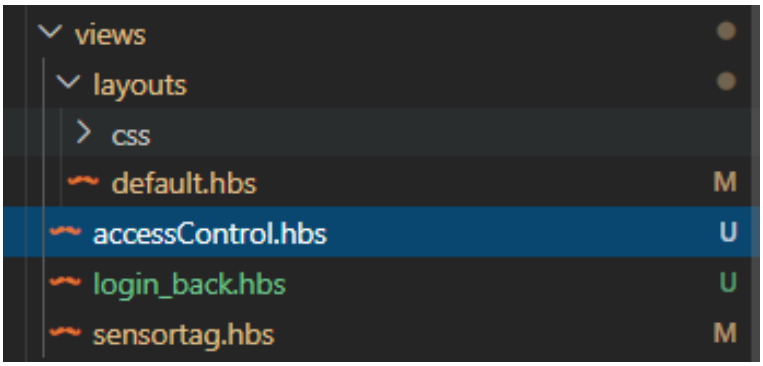

Figura 4.8: *API: vistas implementadas*

La vista que permite autenticar y autorizar al cliente se muestra en la figura 4.9. E stos

elementos se encuentra en el endpoint */accessControl* que son accesibles desde el navegador.

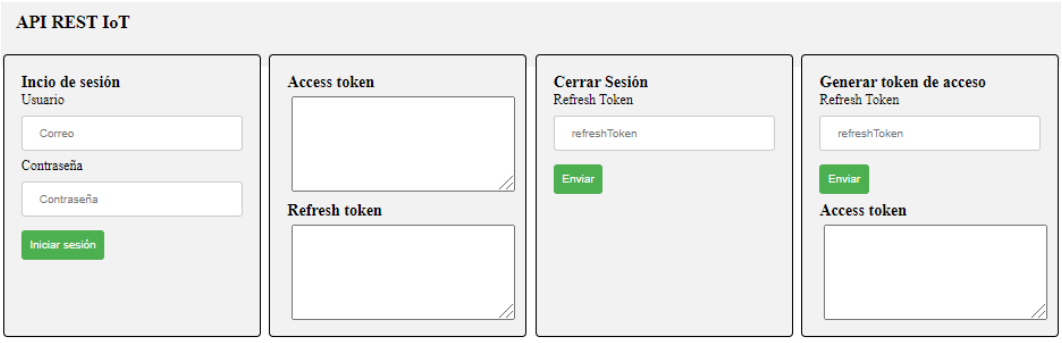

Figura 4.9: *API: vista de autenticación y autorización*

4.2.5. Controladores

En esta sección se describen los controladores correspondientes al usuario y el sensor. Por lo que cada *endpoint* tiene asignado método REST.

En la figura 4.10 se muestra la función *getSensor*. Esta devuelve los valores del registro con **_id** específico. Y así sucesivamente para cada ruta se opera de distinta forma.

En el apartado del usuario es donde se presentan un mayor número de operaciones. Por ejemplo para obtener todos los usuarios registrados se emplea la función *getUsers*, para hacer esta petición el usuario debe ser administrador y así poder obtener los datos. En la figura 4.11 se muestran las operaciones antes mencionadas y en su conjunto las del usuario.

4.3. Especificación de la API

A continuación se describe cada módulo que integra la API desarrollada como complemento de lo anteriormente descrito:

1. *Autorización*: La API permitirá a los clientes hacer peticiones dependiendo del rol asignado.
2. *Autenticación*: La API permitirá al cliente dar acceso a los clientes identificados en la API.

```
'use-strict'

const Sensor = require('../models/sensor')

function getSensor (req, res){···
}

function getSensors(req, res){···
}

function saveSensor(req, res){···
}

function updateSensor(req, res){···
}

function deleteSensor(req, res){···
}

module.exports = {
    getSensor,
    getSensors,
    saveSensor,
    updateSensor,
    deleteSensor
}
```

Figura 4.10: *API: Controlador del sensor*

```
'use strict'

const User = require('../models/users')
const RefreshTokenValue = require('../models/token')


const Service = require('../services/index')
const db = require('../config')
const bcrypt = require('bcrypt-nodejs')

function getUsers (req, res){···
}
function singUp (req, res){
}
function getUser (req, res)···
}
function login (req, res, next){ // función de inicio de sesión···
}
function getrefreshTokens(req, res){···
}
function deleterefreshToken(req, res)···
}
function createnewTokenbyRefreshToken (req, res, next){···
}
function logout (req, res){···
}
```

Figura 4.11: *API: Controlador del usuarios*

3. *Almacenamiento*: La API se encarga de proporcionar la información a todas las entidades, así como de interactuar directamente con *MongoDB*.
4. *Sensores*: Permite la monitorización de los sensores.

En la tabla 4.1 se muestran los recursos necesarios para integrar la API, así como su descripción de cada uno de éstos:

Esquema	**Parámetro**	**Descripción**. *Todos los valores en rojo son requeridos al hacer las peticiones.*
Sensortag	ID Usuario * SensorTag * Valor * Sensor Fecha	Ésta es la entidad principal.
Usuario	* Email (Correo electrónico) * Name (nombre) * password (Contraseña) singupDate lastLogin * role (role) stateAccount accessAttempts	Información de contacto del usuario.
RefreshToken	id_user (ID del usuario) refreshToken	Información que se genera al iniciar sesión.

Tabla 4.1: *Esquemas implementados*

Las operaciones para cada *endpoint* se encuentran en la tabla 4.2. Cada una contiene la ruta, método y su descripción correspondiente de la operación.

4.3.1. Diagrama relacional de la API

Se ha adoptado por integrar un diagrama que permita tener el contexto general del esquema *usuarios*, *sensor* y el esquema de autorización *refreshToken*. En la figura 4.12 se muestra los campos presntes en a los esquemas usados en la API y la relación entre estos.

Ruta (*endpoint*)	**Método(s)**	Descripción
`/v1/sensortag`	`GET POST DELETE PUT`	Se administran los datos del sensor.
`/v1/users`	`GET`	Obtiene todos los usuarios registrados en la API.
`/v1/signup`	`POST`	Permite crear un usuario
`/v1/login`	`POST`	Se autentica el usuario, este devuel un *accessToken* y un *refreshToken*
`/v1/logout`	`DELETE`	Elina el *refreshToken*.
`/v1/refreshTokens`	`GET`	Muestra todos los JWT de refresco, con el *id_user*.
`/v1/token`	`POST`	Este genera nuevos token de acceso (*accessTokens*).

Tabla 4.2: *Lista de endpoints y métodos HTTP*

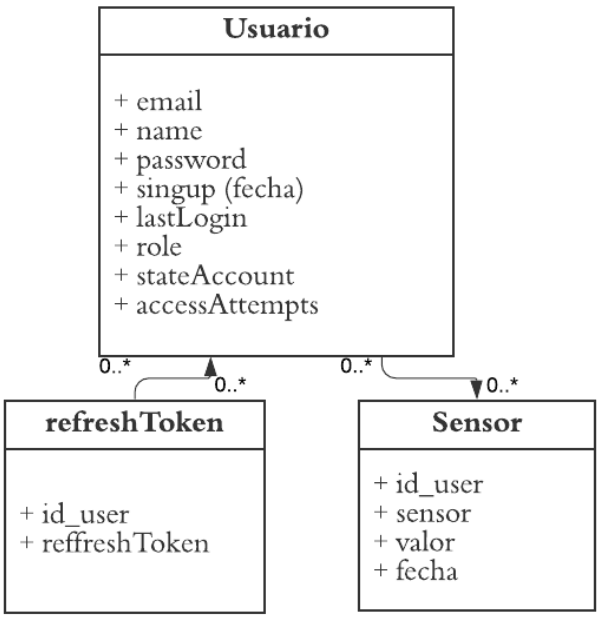

Figura 4.12: *Relación entre esquemas*

Capítulo 5

Swagger

"Solitudinem faciunt pacem appellant. (Donde crean un desierto lo llaman paz.)". – TÁCITO, 55 d.C. – 120 d.C.

El correcto funcionamiento y ejecución de sistema tiene origen en el manejo de sus recursos a través de fuentes o elementos que permitan guiar y entenderlo. La documentación es el medio que permite conectar a los desarrolladores, diseñadores, pentester, etc, para llevar lo acabo.

En el presente capítulo se presenta Swagger como medio para documentar la API en Node.js. Swagger implementa la sintaxis de la *OpenAPI Specification* (OAS).

Cada uno de los elementos antes mencionados serán incorporados a la API desarrollada con el fin de obtener una mejor interpretación del funcionamiento de la API y sus características que envuelven al sistema.

5.1. Swagger.io

Swagger es un *framwork* que implementa OpenAPI Specicification (OAS)[24], lo que permite el desarrollo en todo el ciclo de vida de la API[29], desde el diseño, documentación, a las pruebas e implementación.

Las principales herramientas de Swagger incluyen: Swagger UI[60], Swagger Editor[58], Swagger Codegen[57] y Swagger Hub[62].

OpenAPI tiene la capacidad para describir su estructura por bloques siendo esto una ventaja. Una vez especificada una API, la OpenAPI Specification y las herramientas de Swagger pueden impulsar el desarrollo de la API.

5.1.1. Swagger Editor

Swagger Editor se puede usar en internet, o haciendo uso de un contenedor en Docker, o llevar un instalación de dependencias y usarlo localmente el repositorio en Github[59]. Swagger Editor se ha usado como herramienta para escribir la sintaxis OAS en la API desarrollada.

La interfaz gráfica de Swagger Editor se muestra en la figura 5.1 la cual está compuesta por la *Barra de herramientas*. En ésta se puede descargar las declaraciones OAS en un archivo **YAML** o **JSON**, o de igual forma se pueden importar estos formatos localmente o desde una URL.

Mientras tanto en el *Panel de Edición* se muestra el contenido de las especificaciones en formato YAML. Aplicado OAS, la visualización se muestra en *UI Panel Docs*, que tiene la capacidad de surtir los cambios en el panel de edición cada vez que se agregue o elimine contenido en el panel de edición.

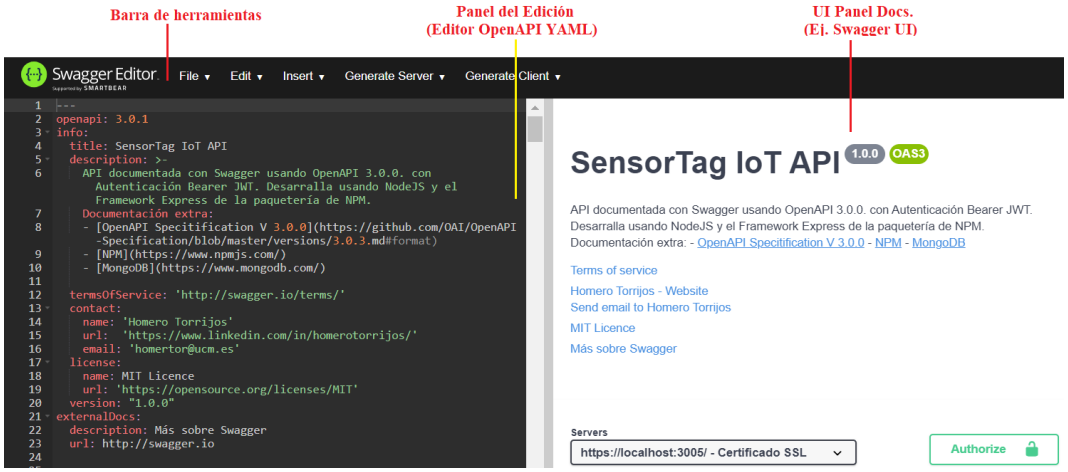

Figura 5.1: *Swagger Editor: Diseño y Especificación OpenAPI*

Para crear un endpoint `/api/v1/signup` como se muestra en figura 5.2 se debe declarar dentro de *paths*. La *Actualización Docs. UI* en Swagger Editor contiene las siguientes seccio-

nes como son la *operación*, *parámetros*, *cuerpo de la solicitud* `application/x-www-form-uriencoded` o `application/json`, así como la sección de *respuestas*. Es posible hacer pruebas de funcionamiento pulsando en `Try It out`.

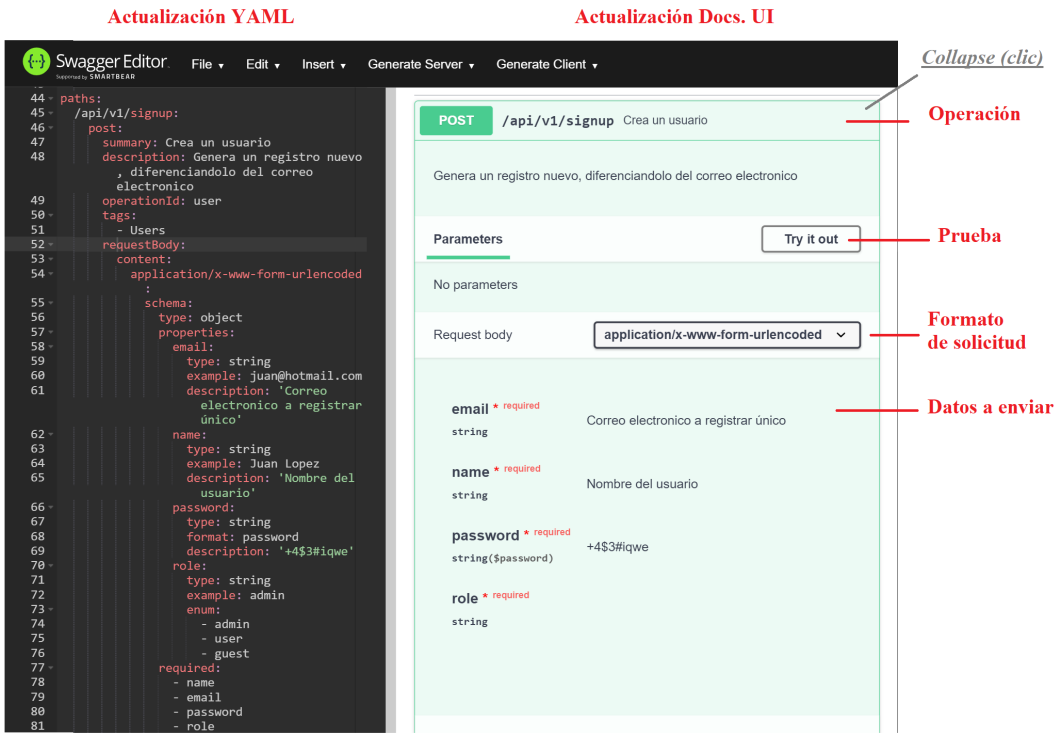

Figura 5.2: *Swagger Editor: cuerpo de la solicitud*

En la sección *responses* se tiene la opción de elegir el valor del *media type*. Por ejemplo en la figura 5.3 se muestran los *códigos de estado*, el formato de la *repuesta* que devuelve el servidor, así como la descripción de cada respuesta.

5.1.2. Swagger UI

El resultado de implementar OAS se ve reflejado en la visualización de todos los endpoints, peticiones, respuestas, datos, etc logrando así la integración en conjunto con la API usando Swagger UI. `swagger-ui-express` es un módulo usado en Node.js. Éste se puede visualizar en la figura 5.4. Para poder observar la documentación se creó el endpoint `/api-docs` en la API desarrollada.

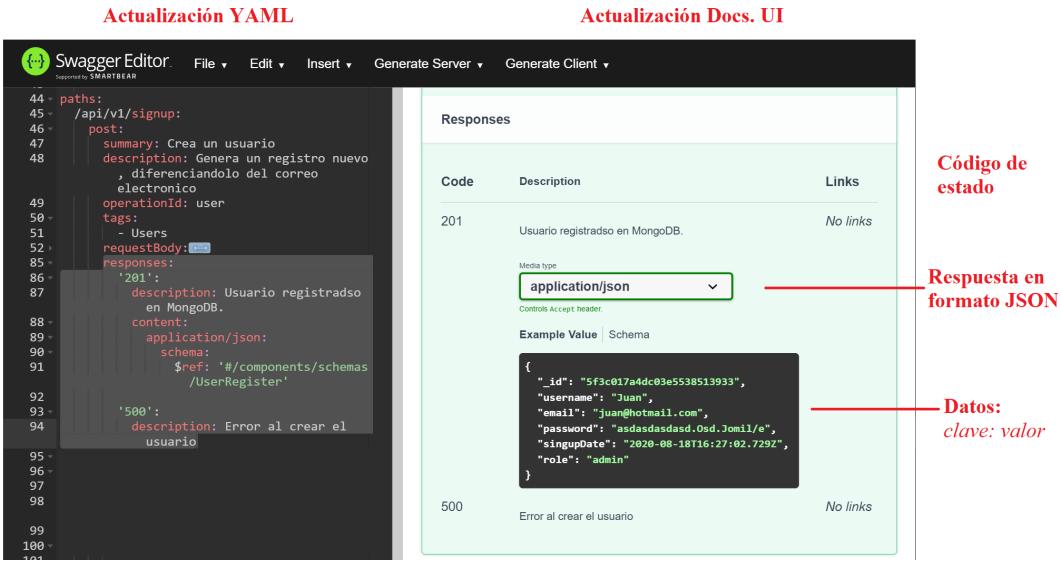

Figura 5.3: *Swagger Editor: respuestas del servidor*

Con esto se tiene acceso a la documentación correspondiente.

```
const YAML = require('yamljs')
const swaggerUi = require('swagger-ui-express')
const swaggerJsDocument = require('swagger-jsdoc')
const swaggerDocument = YAML.load('./swagger-ui/openapi.yaml')

app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocument))
```

Figura 5.4: *Swagger UI: Nodejs*

`yamljs` es el módulo para la lectura de la sintaxis OAS. La lectura se hace través de un archivo YAML. Mientras tanto, *swagger-jsdoc* tiene como uso principal leer la sintaxis OAS en formato *JSON*. Para cargar las reglas en la API se ha declarado una variable `swaggerOptions` que permite leer la estructura OAS en formato JSON como se puede observar en la figura 5.5.

Las operaciones que se definen en el esquema *Users* se muestra en la figura 5.6.

De igual forma en el esquema *Sensor* se muestran las operaciones REST definidas en la figura 5.7.

Para tener una mejor interpretación y uso de la API, los *schemas* se usan para conocer

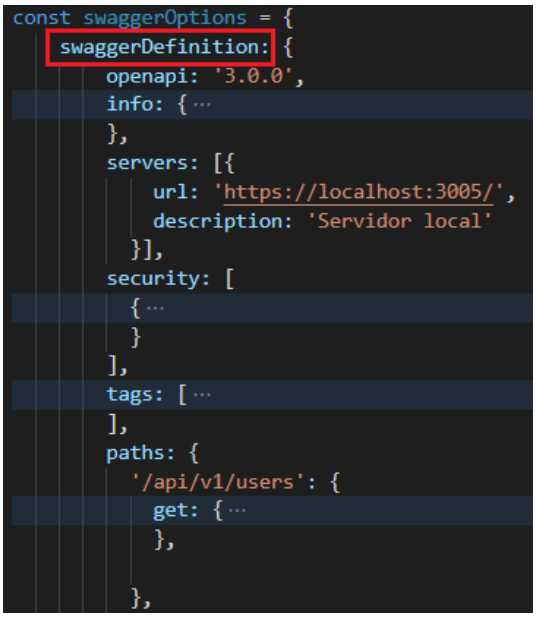

Figura 5.5: *Swagger UI: OpenAPI en formato JSON*

Figura 5.6: *Swagger UI: Usuario*

los tipos de datos que se manejan en las peticiones y respuestas para cada endpoint. Un ejemplo de esto se muestra en la figura 5.8. Mientras tanto en la figura 5.9 se puede que campos son necesarios para crear un usuario y los tipos de datos para esta petición.

Los esquemas de seguridad soportados por Swagger UI son: *autenticación HTTP*, *API keys* (ya sea como una cabecera, parámetro de cookie o como parámetro de consulta), y flujos *OAuth2* (implícitos, contraseña, credenciales de cliente o código de autorización). Los

Figura 5.7: *Swagger UI: Sensor*

Figura 5.8: *Swagger UI: Esquemas*

Figura 5.9: *Swagger UI: esquema crear usuario*

esquemas de seguridad (*securitySchemes*) y la palabra reservada `security` se utilizan para describir los métodos de autenticación utilizados en la API.

En la figura 5.10 se ha creado el esquema de seguridad `bearerAuth` de tipo `http`, esquema `bearer`, así como el formato `bearerFormat` de tipo `JSON Web Token`.

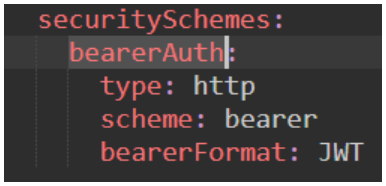

Figura 5.10: *securitySchema: nombre asignado bearerAuth*

Para enviar el JWT del cliente en cada petición que se hace a la API desde el endpoint */api-docs* se agregó el componente `security` con nombre *bearerAuth*. Con esto permite identificarse correctamente el cliente y poder escribir o leer los datos. Cada endpoint creado en la API se agregó en la documentación con sus respectivos tipos de respuestas, peticiones, campo de seguridad, etc. En la figura 5.11 se muestra el ejemplo del campo de seguridad.

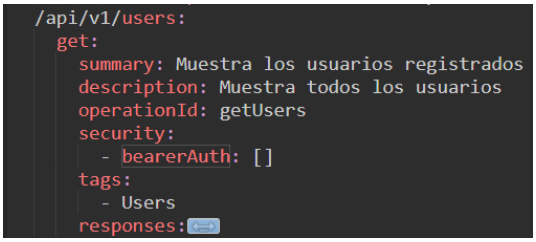

Figura 5.11: *securitySchema: aplicado a las rutas*

Capítulo 6

Auditoría de la API

"Caos: donde nacen los sueños brillantes". – I Ching, China, circa siglo VIII a.C

En este capítulo se abordarán los escenarios vulnerables y seguros sobre la API desarrollada, teniendo como objetivo mejorar la seguridad en los niveles de transporte, integridad de los datos, autorización y autenticación de la aplicación. Para esto se pretende abarcar el uso de ZAP y Wireshark, así como también de mostrar las buenas prácticas propuestas en un desarrollo con Nodes.js.

Por consiguiente se aplicaran herramientas y técnicas de testeo para intentar vulnerar y asegurar la API y con base a las buenas prácticas OWASP API Security Project Top 10.

6.1. Auditoría con ZAP

En esta sección se describe el uso de ataques de *fuerza bruta*, *man in the middle*, *escaneo de cabeceras* de la API, *explotación de JWT* como método de autorización, así como también las credenciales de autenticación. Se buscará hacer una auditaría en seguridad sin el objetivo de perjudicar la integridad de los sistemas REST.

6.1.1. Escenario inseguro: testeo de cabeceras

Para este tipo de pruebas se utilizó ZAP, herramienta que permite escanear posibles vulnerabilidades que tiene la API en términos lógicos del cliente y en la aplicación en su conjunto.

Con esta herramienta se escanearon las cabeceras. ZAP permite hacer de forma automatizada leer los datos de los formularios implementados en esta API. Por consiguiente en la figura 6.1 se muestra un ataque sobre la dirección del servidor `192.168.0.9:3005`.

Figura 6.1: *ZAP: Escaneo de vulnerabilidades*

En la API se identifican tres tipos alertas que van desde bajo, medio y alto riesgo. Se ha considerado hacer los cambios necesarios sobre la API de acuerdo a cada parte que integra la cabecera de la API. En la figura 6.2 se muestra la lista de alertas de la API que deben cambiarse o configurar.

Figura 6.2: *ZAP: Alertas de falta de configuración*

La falta de configuración o parámetros también es un error que puede vulnerar la API. Al aplicar el escaneo sobre el servidor la alerta que arroga esta relacionado con la falta de configuración del *Cache-Control* y *HTTP Pragma* como muestra en la figura 6.3.

El parámetro *Cache-Control* especifica directivas (instrucciones) para almacenar temporalmente (caching) tanto en peticiones como en respuestas. Mientras tanto *HTTP Pragma* es una cabecera HTTP/1.0 y funciona de igual forma que *Cache-Control*.

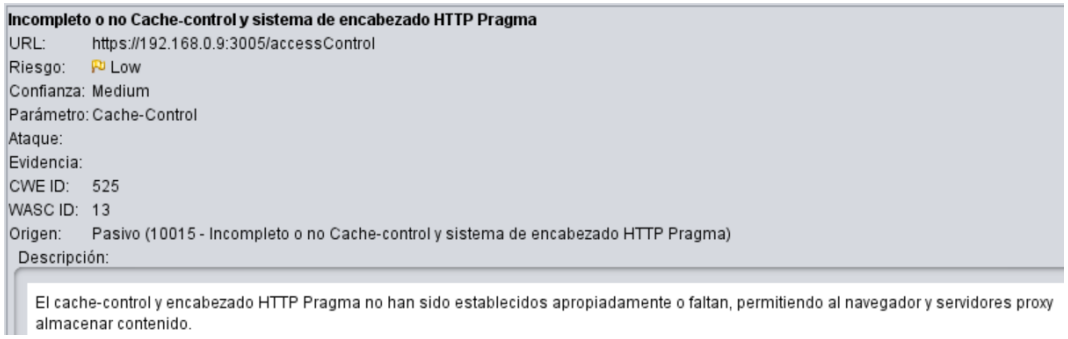

Figura 6.3: *ZAP: no Cache-Control*

Esta vulnerabilidad consiste en ingresar información confidencial en la aplicación. Por ejemplo al cerrar sesión pasando a dar clic en el botón *Atrás* en el navegador para verificar si se puede acceder a la información confidencial mostrada o enviada anteriormente mientras no se está autenticado.

Este escenario se puede aplicar a la autenticación. Por ejemplo, cuando un usuario ingresa su dirección de correo electrónico para suscribirse a un boletín, esta información podría recuperarse si no se maneja adecuadamente.

Mientras tanto el `Content-Type` es la propiedad de cabecera (header) usada para indicar el *media type* del recurso. `Content-Type` indica al cliente qué tipo de contenido será retornado.

Las peticiones que hace el cliente al servidor deben especificarse. Establecer esta cabecera evitará que el navegador interprete los archivos como un tipo MIME diferente al especificado en la cabecera HTTP de tipo `Content-Type`. Al escanear la API nos muestra esta vulnerabilidad. En la figura 6.4 se muestra una descripción de los posibles ataques y el entorno vulnerable sobre la aplicación.

Las respuestas por parte del servidor llevan consigo la propiedad `X-Power-By` que identifica al framework o componentes usados para desarrollar la aplicación. Al escanear la API lo muestra de bajo riesgo, en la figura 6.5 se muestra como falta de configuración aplicado a la API

El parámetro `X-Frame-Options` de la cabecera tiene que ser establecido dado que valida

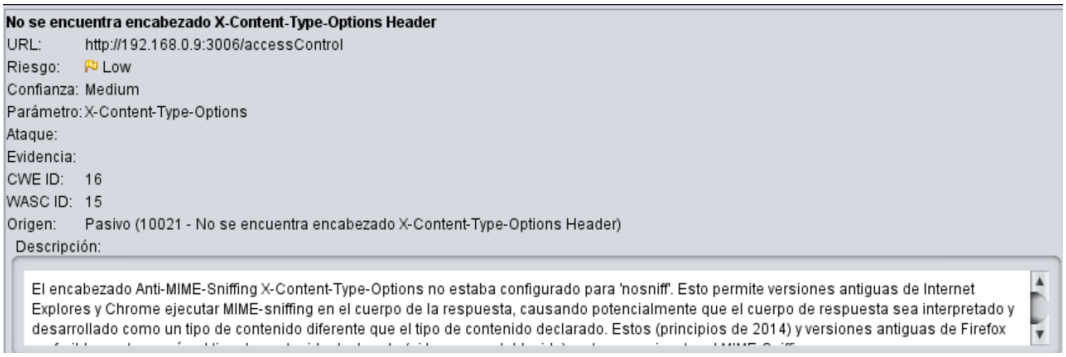

Figura 6.4: *ZAP: Content-Type*

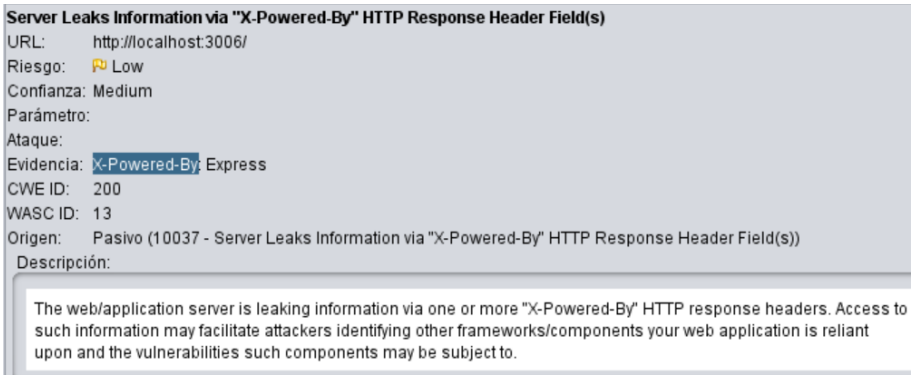

Figura 6.5: *ZAP: X-Power-by*

el origen de la peticiones como se muestra en la figura 6.6. En este caso se describe vulnerable a ataques de tipo *ClickJacking*. Este tipo de ataque, ya sea solo o junto con otros ataques, podría enviar comandos no autorizados o revelar información confidencial mientras la víctima interactúa con páginas web aparentemente inofensivas, todo esto mediante la interfaz gráfica de navegación mediante botones, referencias, imágenes, etc.

6.1.2. Escenario seguro: testeo de cabeceras

Cabe destacar y tener en cuenta las recomendaciones de la OWASP en su REST Security Cheat Sheet descrita en el Capitulo 2. En ésta se debe considerar limitar el uso de los verbos HTTP para cada ruta como buena práctica.

Los parámetros de la cabecera mencionados en la sección anterior y adicionales a estos se muestran en la siguiente lista:

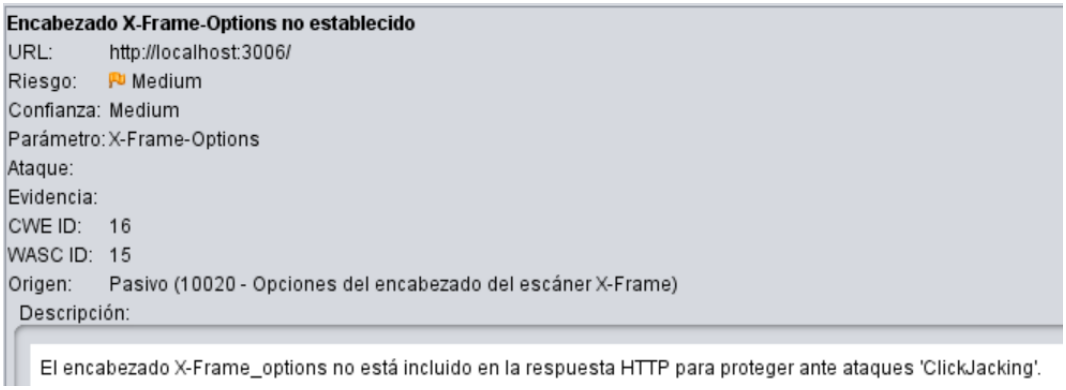

Figura 6.6: *ZAP: X-Frame-Options*

- `Content-Security-Policy`
- `Content-Type`
- `Cache-Control`
- `X-Frame-Options`
- `Access-Control-Allow-Origin`
- `Access-Control-Allow-Headers`
- `Access-Control-Allow-Methods`

De acuerdo a Express[33] como framework de desarrollo, Helmet funciona como herramienta de seguridad que ayuda a proteger la configuración de la cabecera de la aplicación siguiendo las vulnerabilidades conocidas aplicadas a la API. En la figura 6.7 se aplicó una configuración en función las riesgos que fueron encontrados con ZAP.

```
app.disable('x-powered-by')
app.use(function (req, res, next) {
  res.removeHeader("X-Powered-By")
  next()
})

app.use(cors())
app.use((req, res, next) => {
  res.setHeader("Content-Security-Policy", "script-src 'self' https://apis.google.com")
  res.setHeader("Content-Type", "application/json")
  res.setHeader('Cache-Control', 'no-store, no-cache, must-revalidate')
  res.setHeader('X-Content-Type-Options', 'nosniff')
  res.setHeader('X-Frame-Options', 'SAMEORIGIN')
  res.setHeader('Access-Control-Allow-Origin', 'true')
  res.setHeader('Access-Control-Allow-Headers', 'Authorization, X-API-KEY, Origin, X-Requested-With, Content-Type, Accept, Access-Control-Allow-Request-Method')
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE')
  res.setHeader('Allow', 'GET, POST, PUT, DELETE')
  next()
})
```

Figura 6.7: *API: Configuración apropiada de las cabeceras*

6.1.3. Escenario inseguro: explotación HTTP

Para llevar a cabo el escaneo de la red se utilizó *Wireshark*, siguiendo los parámetros correspondientes al protocolo HTTP y el puerto definido en la API. El tráfico capturado con Wireshark fue filtrado por peticiones *POST*, así como el protocolo HTTP y el puerto como se muestra en la figura 6.8. En ésta petición se envían datos al endpoint */api/v1/login* y muestra un `media type` *application/x-www-form-urlencoded*, así como una respuesta del servidor con código de estado 200.

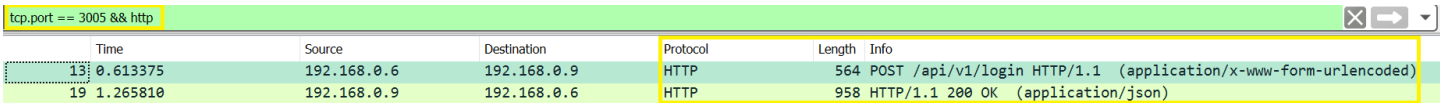

Figura 6.8: *HTTP*

Se identificó el paquete donde se encuentran los datos en texto plano de la petición. En este paquete se identificaron los datos de inicio de sesión que devuelve el servidor. En la figura 6.9 se muestra el correo electrónico y la contraseña del cliente que hizo la petición.

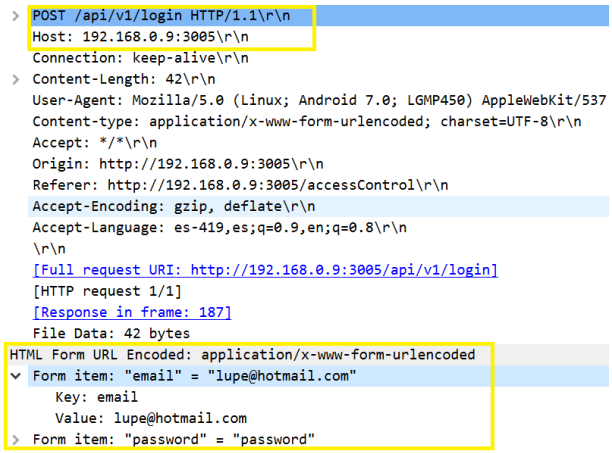

Figura 6.9: *HTTP: Datosde autenticación del usuario en texto plano*

6.1.4. Escenario seguro: uso de HTTPS

El uso de HTTP sin cifrar no tiene una ventaja con respecto a la confidencialidad, integridad de las comunicaciones seguras.

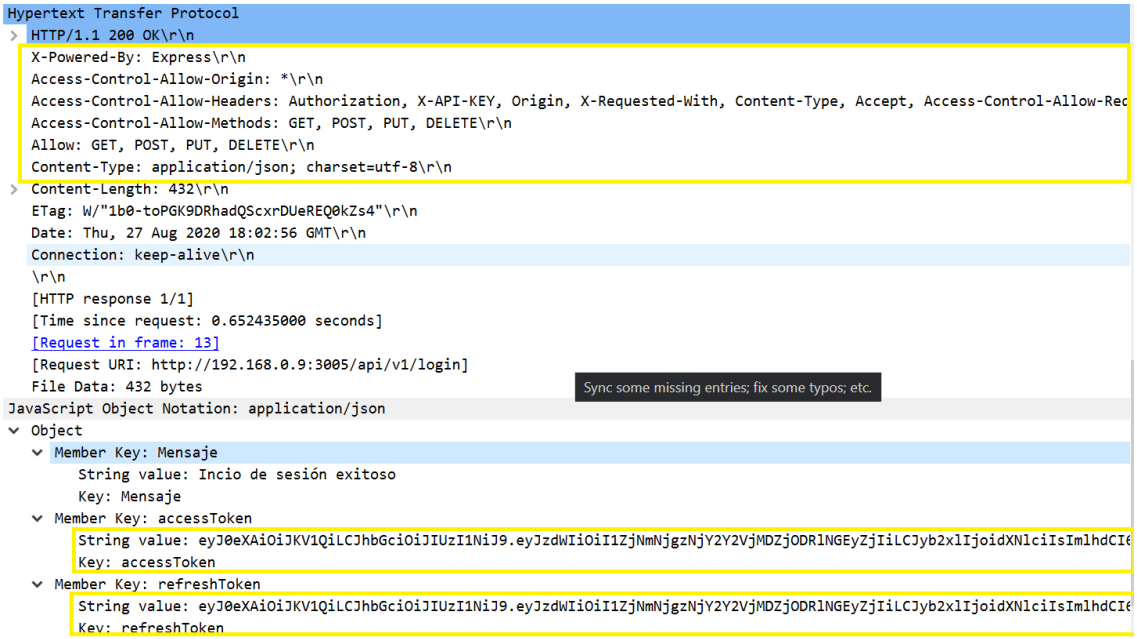

Figura 6.10: *HTTP: Datos en texto plano*

Para asegurar la comunicación entre el cliente y el servidor se requiere hacer uso de certificados digitales y claves. La implementación de herramientas como OpenSSL[35], son una alternativa para generar certificados y claves.

En el siguiente comando de OpenSSL se muestra cómo crear un certificado y la clave privada. En este comando se puede definir el tipo de certificado, la extensión de caducidad en días, el tipo de cifrado a implementar y de igual forma las entradas de configuración del certificado que puede ir en texto plano.

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout private.key -out private.csr -config request.txt
```

Con el comando anterior se generó una clave privada *private.key* con cifrado RSA de 2048 bits con una caducidad de 365 días, seguido del certificado como *private.csr*. Una vez obtenidos los archivos y la *passphrase* se cargaron a la aplicación como se muestra en la figura 6.11. Esto permite iniciar el servidor con una comunicación cifrada entre el cliente y el servidor.

Una vez iniciado el servidor la comunicación se encuentra cifrada. Por lo que no se puede

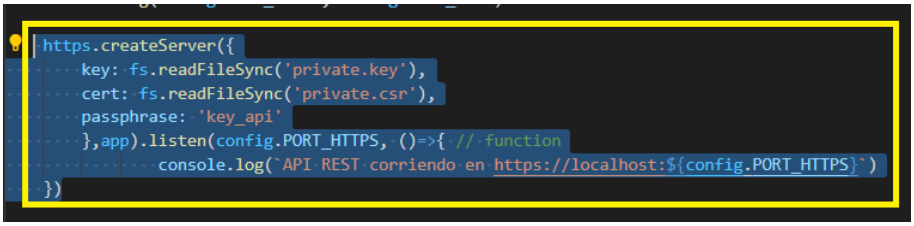

Figura 6.11: *HTTPS: Clave y certificado*

saber el puerto en el que se ejecuta y los tipos de peticiones que se hacen hacia el servidor. En la figura 6.12, se muestra la captura de un paquete del cliente. En este caso se muestra los datos se encuentran cifrados.

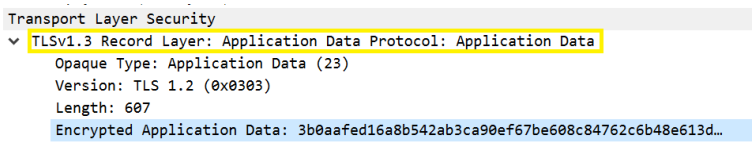

Figura 6.12: *HTTPS: Datos cifrados usando certificados*

6.1.5. Escenario inseguro: explotación almacenamiento local

El objeto *Storage* (API de almacenamiento web), permite almacenar datos de manera local en el navegador, sin la necesidad de realizar alguna conexión a una base de datos. *localStorage* (almacenamiento local) es una propiedad que accede al objeto *Storage*. *localStorage* almacena la información de forma indefinida o hasta que se decida limpiar los datos del navegador.

En la implementada API se tiene el endpoint `/api/v1/sensortag` que implementa el uso del *localStorage*. Haciendo uso de un navegador web como cliente de la API, y accediendo al endpoint */accessControl* con las credenciales correctas de un usuario registrado en la API, se puede tener acceso a los datos almacenados del sensortag en el endpoint `/api/v1/sensortag`.

Para ejemplificar el uso de *localStorage* se creó una vista *accessControl* que se puede acceder en el endpoint */accessControl* que tiene como resultado una interfaz de autenticación

como se muestra en la figura 6.16.

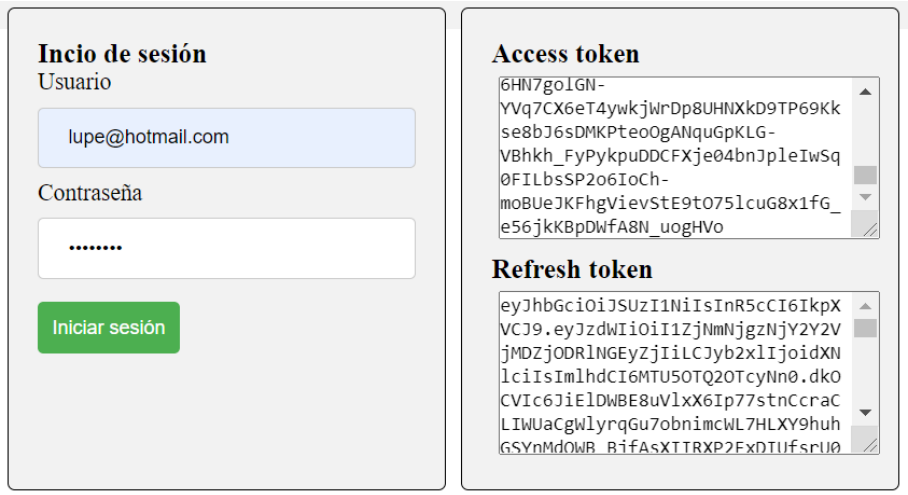

Figura 6.13: *API: autenticación y autorización del cliente*

Esta misma vista contiene el código JavaScript para almacenar el JWT de acceso en el navegador. En la figura 6.14 se muestra la función *fetch*, donde se hacen llamadas al endpoint */api/v1/login* para obtener los JWT de acceso y de refresco que envía el servidor. Las credenciales se obtienen desde el formulario de esta vista. La instrucción usada para almacenar el JWT de acceso se muestra en la parte inferior de la figura siguiendo la sintaxis `localStorage.setItem(‘‘accessToken’’, data.accessToken)`.

```
fetch('/api/v1/login', {
    method: 'POST',
    headers: {
        "Content-type": "application/x-www-form-urlencoded; charset=UTF-8"
    },
    body: params
})
.then(res => res.json())
.then(data =>{
    }else{
        document.getElementById("acctoken").innerHTML  = ''
        document.getElementById("reftoken").innerHTML  = ''
        document.getElementById("acctoken").innerHTML  = data.accessToken
        document.getElementById("reftoken").innerHTML = data.refreshToken
        localStorage.setItem('accessToken', data.accessToken)
    }
```

Figura 6.14: *Implementación: localStorage*

Para acceder a la propiedad *localStorage* en el navegador Firefox o Google Chrome es

preciso hacer clic derecho *Inspeccionar* y en el menú de opciones *Consola*. En la figura 6.15 se muestra la ejecución del comando *localStorage* lo cual muestra sus propiedades.

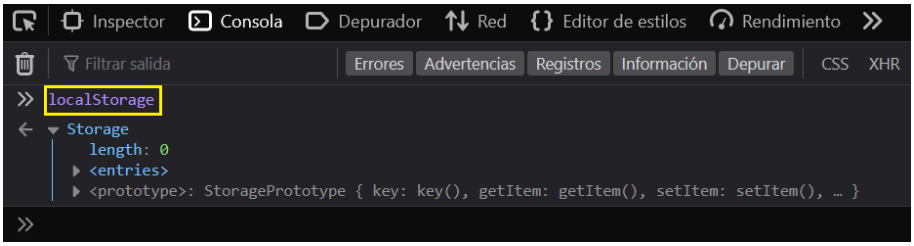

Figura 6.15: *Navegador: localStorage*

Seguido este proceso se puede obtener el JWT en la consola del navegador mediante la ejecución de *localStorage* una vez que el usuario se ha autenticado. En la figura 6.16 se muestra el nombre la variable *accessToken* y su valor que almacenado. El formato que devuelve es en JSON.

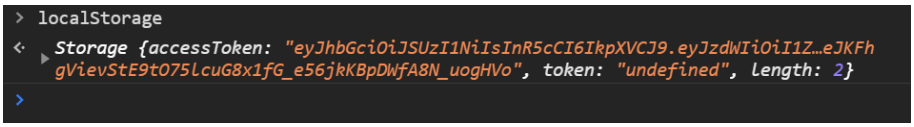

Figura 6.16: *Explotación del token*

Esto representa una vulnerabilidad al almacenar los datos de autorización en el lado del cliente. Es posible extraer de un JWT el algoritmo de cifrado, la fecha en la que expira, datos relativos al usuario, descifrar la clave con la cuál fue firmado y/o obtener la clave cifrado asimétrico.

Se plantean dos escenarios de vulnerabilidad de un JWT. El primer escenario es cuando el cliente almacena el token de acceso en el *localStorage* del navegador, y el segundo cuando es capturado en una línea de comunicación con el servidor, así como también la perdida o copia de la firma digital del tokens.

Los datos a obtener son tanto la clave como el algoritmo usado para crear el token. Para hacer esto se realizó un ataque de fuerza bruta. La herramienta usada lleva como nombre `c-jwt-cracker`[6] programada en `C`. Esta herramienta se ejecuta en un sistema GNU/Linux.

Para la obtención de la clave se realizaron dos procedimientos: 1) compilar el programa en `C` y 2) ejecutarlo. El procedimiento de ejecución se realiza dentro de la carpeta donde se almacenaron los archivos de compilación con la siguiente sintaxis: *./jwtcrack*, seguido de la cadena del JWT. Esta ejecución se puede observar en la figura 6.17. El tiempo para obtener la clave depende de las características de la computadora a usar, así como el formato o caracteres usados en la clave. Para la obtención de esta clave llevo aproximadamente 3 minutos. Al usar caracteres especiales se extiende el tiempo para la obtención de la clave entre 5 y 8 horas aproximadamente.

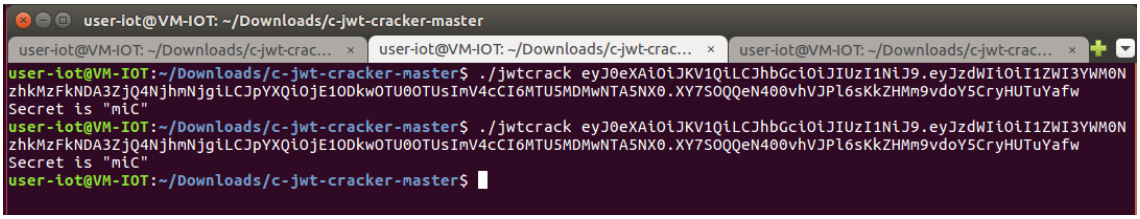

Figura 6.17: *Obtención de la clave mediante fuerza bruta*

6.1.6. Escenario seguro: explotación de almacenamiento local

Para corregir este escenario es indispensable el uso de cifrado asimétrico mediante una clave privada que se encargaría de firmar los JWT de acceso y de refresco, y mediante la clave pública se decodificaría la integridad del JWT usado. Para generar la clave pública o privada se puede hacer a través de OpenSSL y o en línea[71]. En este sentido se ha implementado las claves de ambas formas.

Se generaron las claves públicas y privadas con un algoritmo de cifrado RSA256. Se tiene como resultado una mejora en la seguridad en la comunicación entre el cliente y el servidor en términos de confidencialidad, integridad y disponibilidad de los datos. En la figura 6.18 se muestra la importación de las claves para crear el JWT y decodificarlos. Estas son usadas por el *middleware* de autorización y autenticación discutidos en el Capítulo 4.

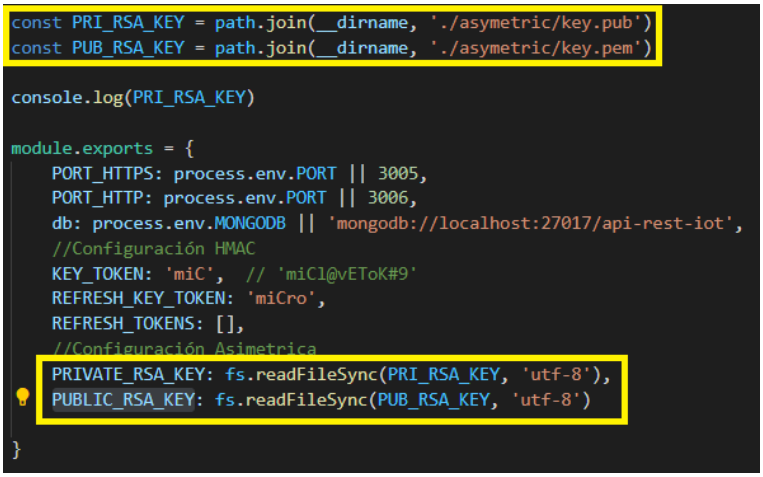

Figura 6.18: *Cifrado: con llave pública y privada*

6.1.7. Escenario seguro: Helmet

Helmet[2] es una aplicación que ayuda a proteger las APIs desarrolladas basadas en *Express*. Los parámetros de la API escaneados con ZAP pueden configurarse automáticamente usando llamando a `app.use(helmet())`, como se muestra en la figura 6.19, o en su caso individualmente a cada parámetro que compone la cabecera. Esto permite agilizar la seguridad de la aplicación dado que es adaptada de acuerdo a la vulnerabilidades existentes en el framework Express.

Figura 6.19: *Helmet configuración de seguridad en Express*

Helmet es también una colección de funciones middleware que establecen cabeceras de

respuesta HTTP relacionadas con la seguridad:

- CSP (Content-Security-Policy): establece la Política de Seguridad de Contenido para ayudar a prevenir los ataques Cross-site Scripting y otras inyecciones de sitios.
- ieNoOpen: establece X-Download-Options solo para Internet Explorer 8+. Instrucción que le dice al navegador no abrir directamente las descargas por lo contrario la única opción es no permitirla.
- Frameguard: establece la cabecera de X-Frame-Options para proporcionar protección contra el clickjacking[40].
- xssFilter: establece X-XSS-Protection para habilitar el filtro de Cross-site scripting (XSS) en los navegadores web más recientes.

6.2. Auditoría de seguridad según la OWASP

En la siguiente sección se introducirán las buenas prácticas según la *OWASP Top 10 Security Project* visto en el capítulo 2, lo cual abarcará la *autenticación* y *autorización* como puntos principales de explotación de un sistema. Esto se aplicará a la API desarrollada.

6.2.1. Autorización

En esta sección se abarcará la *autorización por niveles*, y el registro por identificador único. Este es asignado por el gestor de base de datos NoSQL (**MongoDB**), aplicado a los esquemas usuarios y sensores.

Como se puede observar en la figura 6.20, se le asigna automáticamente un *"_id":* que es único por cada registro que se haga sobre este esquema "sensortags".

Cabe resaltar en el esquema aplicado a los usuarios que este cuenta con elementos más importantes que son validados desde su entrada como son el correo electrónico, contraseña, y la longitud, así como la presencia de caracteres especiales. Como se muestra en la figura 6.21 cada usuario registrado debe cumplir con el uso de caracteres especiales.

De igual forma para asegurar el acceso a los recursos solicitados por el usuario o cliente se

```
"sensortags": [
    {
        "valor": 0,
        "_id": "5eb61e2ed1f5760c104559a3",
        "sensorTag": "sen_cc2650_home",
        "sensor": "tem",
        "fecha": "2021-09-04T22:00:00.000Z",
        "__v": 0
    },
```


Figura 6.20: *Registro de un sensor*


```
{
    "email": "lupe@hotmail.com",
    "password": "$2a$10$v7uBiQc1/fHvxVW//ki6y.zlVIDNIbgbGsM8WOKFwoK7UoWnU6BZS",
    "singupDate": "2020-08-21T06:22:23.201Z",
    "_id": "5f3f683666cec06c84e4a2f2",
    "name": "lupe",
    "role": "user",
    "__v": 0,
    "stateAccount": false,
    "accessAttempts": 3
},
```


Figura 6.21: *Registro de un usuario*

creó un esquema con nombre *refreshTokenV* como se muestra en la figura 6.22 que permite almacenar identificar el *_id* del usuario, a partir de que se autentica. Por consiguiente se genera un refreshToken que es almacenado y el cliente lo pueden usar para generar tokens de acceso, mecanismo usado para hacer peticiones hacia el servidor. Los refreshTokens se generan sólo una vez. Mientras tanto, estos dos niveles de seguridad aportan gran ventaja al acceso de recursos.

El esquema de un usuario está definido por los siguientes campos: *email* (correo electrónico), *name* (nombre), *password* (contraseña) y *role* (rol), entre sus principales, se validan los datos de entrada; tamaño o rango de caracteres, el tipo de dato. Esto se puede observar en la figura 6.23.

Cabe resaltar que para cada *endpoint* se aplica un tipo de autorización, donde el rol de administrador corresponde a **auth.isAuthAdmin** el cual puede realizar todas las opera-

```
"refreshTokenV": [
  {
    "_id": "5f516caeec9a8369f0f9b487",
    "id_user": "5f3f683666cec06c84e4a2f2",
    "refreshToken": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1ZjNmNjgzNjY2Y2VjMDZjODRlNGEyZjIiLCJy
    b2xlIjoidXNlciIsImlhdCI6MTU5OTE3MTc1OH0.UlIwYO8ZYHX8lLyfdfBO1RlQKsbAlEZvvJe8VSAY1YGcL0H4SkdCDRO9A2dIbr
    PmNbMnXdnoaYWigWy8FZzMhzwN2S1MASmeO7gsPlq5uKVoIgXEaBW1VP3Yz-hP4v-I-
    ZOrB6AaYYTRdtoiLI_mSzXzpcwyS84DJ8HfEm-
    VnZthwUprE4qFxML2Sg1XyvFoL9ek_O9xXdJRWr0REtGllOdxFhxNeTSqXKYpqRB2G62HOVE1o-O8oWII5yBMxx-
    jCHqegZsAUhZc4nksYJ4TrF1j2gNc69K3PC_z1QGP5p9f9yByTp7eiZrPUDZY4dLslAKFUprwBuqMaEbPtPRTa1EMDFbu4DFS1ZVB0
    y2dPOZIHdhkxjZcDkDJwTCI6vw8uXgz9rsQjKGchkHHsMI539b_rka7QJNibigLrP7AWkT5ZhlV49CLeu49X3gpZVvvmAl8w9qdUz_
    67FKleu3gBwsdTJl1CO9lViTffVTOEe0FEe6HGLeyo1r_eKoxsv7PtQS6q-2gTAdxq-mOO6mbELwjmC-Ma-
    bn0B1ZqUcuFpXOrrt24toiqGqHddtv_z8agaoRvV-
    pz7d_OS6QR9R7VoQ5l1Sm_sSymobWAzw_u7Z5ljftFKIKolaXf0q6NzXvIXACjTdc5okoS-VVIhed7AHHxGeqypBv2jCuxMxRlOE",
    "__v": 0
  }
]
```


Figura 6.22: *Generación de un refreshToken*

```
const userSchema = new Schema ({
    email: {type: String, trim: true, select: true, unique: true, index: true,
      lowercase: true, "default": "", validate: [validateLocalStrategyProperty,
      "Ingresa tu email"], match: [/.+\@.+\..+/, "Ingresa un email valido"]},
    name: {type: String, unique: true, select: true, index: true, lowercase: true,
      required: "Completa tu nombre de usuario",
      trim: true, match: [/^[\w][\w\-\._\@]*[\w]$/, "Ingresa un nombre de usuario valido"]},
    password: {type: String, "default": "",
      validate: [validateLocalStrategyPassword, "La contraseña debe ser más grande"]},
    singupDate: { type: Date, default: Date.now()},
    lastLogin: Date,
    role: {type: String, enum: ['admin','user', 'guest']},
    stateAccount: {type: Boolean, default: false},
    accessAttempts: {type: Number, default: 0},
})
```

Figura 6.23: *Aplicación de autorización por niveles*

ciones permitidas mientras tanto en el rol usuario o invitado **auth.isAuthNormal**. Esto se puede observar en la figura 6.24.

```
// Apartado Sensor
api.get('/v1/sensortag', auth.isAuthAdmin, SensorCtrl.getSensors)
api.get('/v1/sensortag/:sensortagId', auth.isAuthNormal, SensorCtrl.getSensor)
api.post('/v1/sensortag', auth.isAuthAdmin,  SensorCtrl.saveSensor)
api.put('/v1/sensortag/:sensortagId', auth.isAuthAdmin, SensorCtrl.updateSensor)
api.delete('/v1/sensortag/:sensortagId', auth.isAuthAdmin, SensorCtrl.deleteSensor)
```

Figura 6.24: *Autorización en las rutas*

6.2.2. Autenticación

En esta sección se toma en consideración lo propuesto por la OWASP API Security, en términos de autenticación.

En primera instancia se presenta el cifrado de la contraseña en la creación de un usuario, así como también a la hora de autenticarse. Seguido esto se presenta el estado de la cuenta o registro del usuario después de 3 intentos para autenticarse este pasaría a bloquearse.

En la figura 6.25 se muestra la función de *userSchema.pre*, que permite obtener el dato correspondiente a la contraseña *hasheado* 10 veces para crear una y almacenarla. Cabe mencionar que durante la creación de un usuario este devuelve los datos y la contraseña firmada.

```
userSchema.pre("save", function(next)
{
    var user = this

    if (!this.isModified("password"))
    {
      return next()
    }
    return bcrypt.genSalt(10, function(err, salt)
    {
      if (err) {
          return next(err)
        }

        return bcrypt.hash(user.password, salt, null, function(err, hash) {
        if (err) {
          return next(err)
        }
        user.password = hash
        return next()
      })
    })
})
```

Figura 6.25: *Cifrado de la contraseña en el registro*

En el momento de la autenticación cuando el usuario envía su contraseña por método POST, la comparación se hace con ambas contraseñas firmadas. Esto se muestra en la figura 6.26. En la respuesta de servidor cuando el usuario se autentica la función *isMatch* devuelve un error durante el proceso, en el caso de que no sea la contraseña correspondiente a la almacenada en el registro.

Para llevar a cabo la autenticación y tomar en cuenta que la API puede ser atacada con

```
userSchema.methods.comparePassword = function(pwd, cb) {
    bcrypt.compare(pwd, this.password, function(err, isMatch) {
      if (err) {
        return cb(err)
      }
      cb(null, isMatch)
    })
  }
```

Figura 6.26: *Autenticación: se cifra la contraseña por el cliente y se con la del servidor*

el uso de diccionarios desde *JavaScript*. Se ha definido que la API puede bloquear la cuenta del usuario con dos posibles variables a tomar en cuenta, 1) el número de intentos mayor a tres, 2) y, desde el mismo punto que el cliente ya no puede acceder a esta. Cabe resaltar que las pruebas se hicieron desde un cliente creado en HTML y JavaScript, con motor de vista *hbs*[30]. En consecuencia, se puede mostrar que a la hora de la autenticación se decidió tener dos variables que controlen el numero de intentos *accessAttemps* y el estado del usuario *stateAccount* como se muestra en la figura 6.27. Estas variables se encuentran en el esquema del usuario.

```
if(user.stateAccount != false){

    console.log(user.stateAccount)
          any
    user.comparePassword(req.body.password, function(err, isMatch) {
        if (isMatch && !err) {···
        } else if(user.accessAttempts < 3){···
        }else if(user.accessAttempts === 3){
            user.updateOne({ stateAccount: false}, function(err, res) {
                if (err) {
                    res.send(err)
                } res.status(403).send({
                    Mensaje: "Usuario bloqueado",
                })

            })
        }
    })
}else{
    console.log('Usuario bloqueado')
    return res.status(403).send({
        Mensaje: 'Usuario bloqueado',
        Usuario: '3'
    })
}
```

Figura 6.27: *Validación del usuario*

Un usuario bloqueado no puede acceder al recurso que solicita lo cual recibe un código de

estado 403, y un mensaje del servidor que identifica al usuario en un nivel que hace referencia que esta bloqueado, como se muestra en la figura 6.28. En esta se muestra un *alert*, aun sí la contraseña es correcta, no se puede acceder directamente al servicio correspondiente.

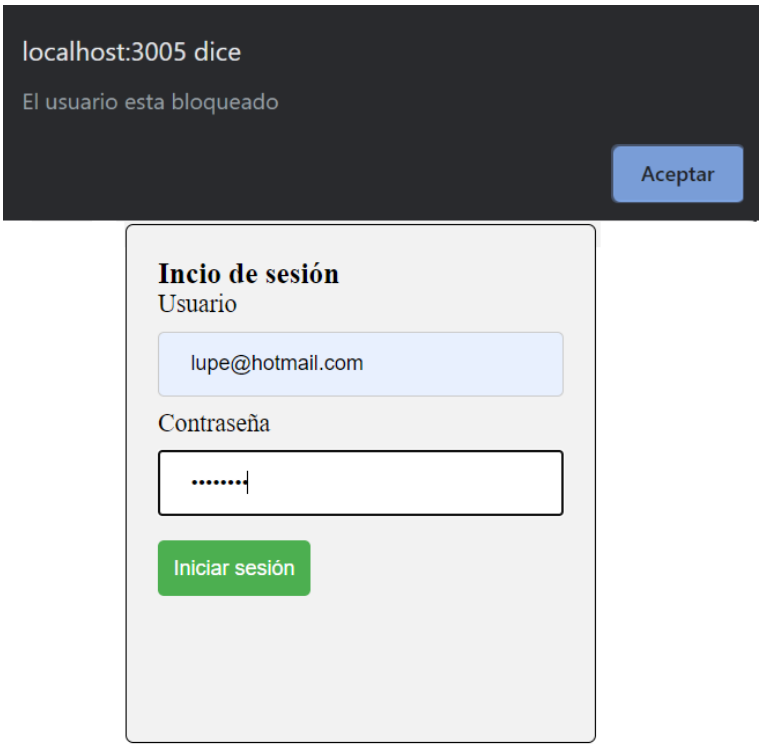

Figura 6.28: *Vista del cliente: usuario bloqueado*

Capítulo 7

Conclusiones y Trabajo Futuro

"En la guerra, la pusilanimidad es mil veces más perniciosa que el arrojo, dado un grado igual de inteligencia.". – CARL VON CLAUSEWITZ, 1780 – 1831.

En el presente trabajo se ha desarrollado completamente la infraestructura propuesta inicialmente, y se han llevado a cabo todas las tareas hechas para alcanzar el objetivo principal. En este capítulo se intenta recapitular, los logros obtenidos, y las líneas en las que se puede orientar en términos de auditoría en seguridad y desarrollo de software.

7.1. Conclusiones

A lo largo de este Trabajo Fin de Máster se estudiaron diferentes tecnologías para llevar a cabo el desarrollo de la API y que fueron la base para construir la infraestructura implementada. En particular se desarrolló una API que permite administrar recursos de un sensor genérico que envía peticiones REST autorizadas hacia el servidor y se controla el acceso de los clientes y/o usuarios por medio de la autenticación, permitiendo acceder a los datos almacenados del nodo sensor. El objetivo principal de este Trabajo Fin de Máster fue auditar la API de acuerdo a las buenas prácticas de las OWASP Foundation y aplicándolas en el software escrito en Node.js.

El tránsito seguro de los datos del cliente al servidor es de suma importancia dado que se debe tener el canal de comunicación cifrado. Esto conlleva una mejora en la confidencialidad

e integridad de los datos. Las características del entorno en el que se está trabajando influyen muy poco dado la escalabilidad y la distribución del sistema, pero en el contexto de desarrollo y/o en producción es donde se encuentran todo tipo de vulnerabilidades que puedan afectar al rendimiento del sistema. En este caso se han llevado a cabo las pruebas de seguridad, de desarrollo y almacenamiento de los datos de forma local.

La integración de dispositivos y tecnologías en el desarrollo de esta solución confirman la interoperabilidad en un despliegue IoT. Las APIs están ya muy estudiadas en entornos IoT, dado que son altamente escalables en sistemas y servicios tecnológicos. La opción de integrar aplicaciones nativas o servicios web que puedan consumir los datos lo hacen sumamente completo e integrable un sistema basado en API.

Relacionado con el proyecto una base de datos NoSQL aporta flexibilidad al sistema ya que permite integrar múltiples dispositivos, que generen diferentes esquemas y/o modelos de datos sin necesidad de volver a diseñar la base de datos.

La detección de vulnerabilidades en el desarrollo de esta API posibilita anteponerse a posibles riesgos de seguridad y fallos que puedan afectar al rendimiento del sistema. Identificar de manera prematura que una API va a sufrir ataques permite asimilar y reaccionar de manera automática y eficiente.

La auditoría de seguridad llevada a cabo se realizó en dos fases. La primera en las buenas prácticas recomendadas por la OWASP descritas en su manual de seguridad OWASP API Security Project Top 10. En este manual se desarrollo las recomendaciones relacionadas a la autorización y autenticación, éstas se aplican a nivel de desarrollo de software lo cuál fue de manera exitosa. De igual forma en conjunto con REST Cheat Sheet se integraron las buenas prácticas en la API desarrollada. En la segunda fase se usó ZAP como herramienta de pentesting de alto valor llegando a una configuración segura de la API, y de ésta manera se evaluaron los riesgos posibles en los parámetros de la cabecera.

Adicionalmente se evaluaron los posibles problemas de seguridad que existen al almacenar los datos de autorización en el lado del cliente usando código JavaScript, y los riesgos

que conlleva como son el descifrado del contenido del mensaje y la obtención de claves de seguridad. Se implementó una forma más segura de firmar los JWT, usando cifrado asimétrico lo cual aumenta el nivel de seguridad en los JWT.

Por último, en el sector TIC existe gran cantidad de software para documentar las API. En el presente trabajo se integraron las especificaciones OAS dado que es una tecnología en tendencia para llevar a cabo la documentación. Swagger se usó como framework que interpreta la especifican OAS y llevándolo así a la adaptación con el trabajo desarrollado en Node.js. Swagger es una herramienta de gran ayuda para mejorar los procesos del desarrollo e implementación de la API, lo que se traduce en un mejor rendimiento y uso del sistema. Swagger se adapta a un gran número de lenguajes de programación.

7.2. Trabajo futuro

A partir de la labor desarrollada este Trabajo de Fin de Máster y tras la auditoría de la API, la línea de trabajo futuro inmediato debe ser la ampliación del sistema a más nodos sensores empleando la arquitectura REST. Si bien se ha visto y verificado el correcto funcionamiento en una ambiente local, es de gran importancia verificar y alternarlo a un ambiente en producción real. Los sistemas distribuidos de igual forman son adaptables a esta arquitectura REST, dado que se pueden separar los servicios de autorización, autenticación, y de bases de datos. Las tecnologías de desarrollo empleadas permiten la escalabilidad por lo que funciona como base para auditar y documentar futuras arquitecturas REST.

Se tiene en consideración aplicar el abanico de posibles formatos de autorización como OAtuh[22] implícita, por contraseñas o credenciales cliente. Permitiendo así ajustar mejores resultados en la seguridad de la API. Cabe resaltar que hacer uso de túneles en las comunicaciones permite reforzar aún más la seguridad entre el cliente y el servidor.

Finalmente el uso del formato JSON ha ido en ascenso como tecnología preferida en el transporte de los datos. Por ello se ha considerado tomar en cuenta el uso del JSON Web Encryption[17] y el JSON Web Signature[18] permitiendo agregar una capa más a nivel de

confidencialidad de los datos.

Bibliografía

[1] N. Laranjeiro A. Neumann and J. Bernardino. *An Analysis of Public REST Web Service APIs.* IEEE Transactions on Services Computing.DOI: 10.1109/TSC.2018.2847344, 2018.

[2] Adam Baldwin. Helmet. https://helmetjs.github.io/, 2020. Accesado: 2020-11-26.

[3] Apache. Apache http server project. http://httpd.apache.org/, 2000. Accesado: 2020-10-26.

[4] Arabian Industry. Big data explosion: 90 % of existing data globally created in the past two years alone. https://www.arabianindustry.com/technology/comments/2013/nov/18/big-data-explosion-90-of-existing-data-globally-created-in-the-past-two-years-alone 2020. Accesado: 2020-05-30.

[5] Auth0. Jwt.io. https://jwt.io/, 2013. Accesado: 2020-03-26.

[6] Brenda Rius. Jwt cracker. https://github.com/brendan-rius/c-jwt-cracker/, 2020. Accesado: 2020-11-13.

[7] Cisco. The internet of things: How the next evolution of the internet is changing everything. https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf, 2020. Accesado: 2020-05-30.

[8] Fernando Doglio. *Pro REST API Development with Node.js.* Apress, primera edition, 2015.

[9] Eliee Saad. Application security engineer. https://github.com/ThunderSon, 2020. Accesado: 2020-10-30.

[10] GitHut. GitHut 2.0, A Small Place to Discover Languages in Github. https://madnight.github.io/githut/#/pull_requests/2020/1, 2020. Accesado: 2020-05-30.

[11] Homero Torrijos. Tfm api swagger iot bearer. https://github.com/homerotorrijos/TFM_API_Swagger_IoT_Bearer_v1, 2020. Accesado: 2020-11-23.

[12] Internet Assigned Numbers Authority (IANA). Hypertext transfer protocol (http) status code registry. https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml, 2018. Accesado: 2020-10-26.

[13] Internet Engineering Task Force. Hypertext transfer protocol – http/1.1. https://www.ietf.org/rfc/rfc2616.txt, 1999. Accesado: 2020-10-26.

[14] Internet Engineering Task Force. (extensible markup language) xml-signature syntax and processing. https://tools.ietf.org/html/rfc3275, 2002. Accesado: 2020-10-16.

[15] Internet Engineering Task Force. The base16, base32, and base64 data encodings. https://tools.ietf.org/html/rfc4648, 2013. Accesado: 2020-03-26.

[16] Internet Engineering Task Force. The javascript object notation (json) data interchange format. https://tools.ietf.org/html/rfc7159, 2014. Accesado: 2020-05-16.

[17] Internet Engineering Task Force. Json web encryption (jwe) draft-ietf-jose-json-web-encryption-26. https://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-26, 2014. Accesado: 2020-06-10.

[18] Internet Engineering Task Force. Json web signature (jws). https://tools.ietf.org/html/rfc7515, 2015. Accesado: 2020-06-10.

[19] Internet Engineering Task Force. RFC2616 - Hypertext Transfer Protocol HTTP/1.1. https://tools.ietf.org/html/rfc2616, 2020. Accesado: 2020-05-30.

[20] Internet Engineering Task Force. RFC7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. https://tools.ietf.org/html/rfc7231, 2020. Accesado: 2020-05-30.

[21] Internet Engineering Task Force. RFC7519 - JSON Web Token (JWT). https://tools.ietf.org/html/rfc7519, 2020. Accesado: 2020-05-30.

[22] Internet Engineering Task Force. The OAuth 2.0 Authorization Framework. https://tools.ietf.org/html/rfc6749, 2020. Accesado: 2020-05-30.

[23] Internet Engineering Task Force. RFC2396 - Uniform Resource Identifiers (URI). https://www.ietf.org/rfc/rfc2396.txt, 2020. Accesado: 2020-05-30.

[24] Linux Foundation. OpenAPI OAS. http://spec.openapis.org/oas/v3.0.3/, 2020. Accesado: 2020-06-30.

[25] Louis Nyffenegge: PentestingLab. Attacking json web tokens. https://owasp.org/www-pdf-archive//20190222--Nyffenegger-JWAT.pdf, 2019. Accesado: 2020-06-10.

[26] Manicode Security. Jim manico. https://github.com/jmanico, 2020. Accesado: 2020-10-30.

[27] Medium. What went wrong? https://medium.facilelogin.com/what-went-wrong-d09b0dc24de4, 2020. Accesado: 2020-05-30.

[28] Mongoose on OpenCollective. Mongoosejs. https://mongoosejs.com/, 2020. Accesado: 2020-11-12.

[29] NodeJS. Swagger nodejs. https://github.com/swagger-api/swagger-node, 2020. Accesado: 2020-11-12.

[30] NPM. hbs. https://www.npmjs.com/package/hbs, 2020. Accesado: 2020-10-30.

[31] NPM. Node package manager. https://www.npmjs.com/, 2020. Accesado: 2020-11-12.

[32] Open Web Application Security Project. Owasp zap 2.9 getting started guide. https://www.zaproxy.org/pdf/ZAPGettingStartedGuide-2.9.pdf, 2020. Accesado: 2020-12-29.

[33] OpenJS Foundation. Production best practices: Security. https://expressjs.com/en/advanced/best-practice-security.html, 2020. Accesado: 2020-11-13.

[34] OpenJS Foundation. Node red. https://nodered.org/, 2020. Accesado: 2020-11-13.

[35] OpenSSL Software Foundation. Openssl. https://www.openssl.org/, 2008. Accesado: 2020-10-30.

[36] OWASP. Rest security cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html, 2015. Accesado: 2020-10-30.

[37] OWASP. Owasp risk assessment framework. https://github.com/OWASP/RiskAssessmentFramework, 2018. Accesado: 2020-10-30.

[38] OWASP. Owasp api security project. https://owasp.org/www-project-api-security/, 2019. Accesado: 2020-10-30.

[39] OWASP. Authentication cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html, 2019. Accesado: 2020-10-30.

[40] OWASP. Clickjacking. https://www.owasp.org/index.php/Clickjacking, 2020. Accesado: 2020-11-13.

[41] OWASP Flagship project. Zed attack proxy (zap). https://www.zaproxy.org/, 2020. Accesado: 2020-11-13.

[42] OWASP Foundation. Documents: Owasp cheat sheet series. https://cheatsheetseries.owasp.org/bundle.zip, 2015. Accesado: 2020-10-30.

[43] OWASP Foundation. Owasp cheat sheet series. https://cheatsheetseries.owasp.org/, 2015. Accesado: 2020-10-30.

[44] OWASP Foundation. OWASP API Security Top 10. https://github.com/OWASP/API-Security/blob/master/2019/en/dist/owasp-api-security-top-10.pdf, 2019. Accesado: 2020-05-30.

[45] OWASP Foundation. OWASP API Security Top 10. https://github.com/OWASP/API-Security, 2019. Accesado: 2020-05-30.

[46] OWASP Foundation. Owasp global appsec tel aviv community of innovation. https://telaviv.appsecglobal.org/, 2019. Accesado: 2020-10-30.

[47] OWASP Foundation. Cross Site Scripting (XSS). https://owasp.org/www-community/attacks/xss/, 2020. Accesado: 2020-10-16.

[48] OWASP Foundation. Cross Site Request Forgery (CSRF). https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html, 2020. Accesado: 2020-05-30.

[49] OWASP Foundation. OWASP API Security Project. https://owasp.org/www-project-api-security/, 2020. Accesado: 2020-05-30.

[50] OWASP Foundation. REST Security Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html, 2020. Accesado: 2020-05-30.

[51] OWASP Foundation, Inc. Openweb application security project. https://www.owasp.org/, 2004. Accesado: 2020-10-30.

[52] Postman, Inc. Postman. https://www.postman.com/downloads/, 2020. Accesado: 2020-11-13.

[53] Roy Thomas Fielding. Architectural Styles and the Design of Network-based Software Architectures. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000. Accesado: 2020-05-30.

[54] Ryan Dahl. Git hub - ry. https://github.com/ry/, 2020. Accesado: 2020-11-12.

[55] Ryan Dahl. Node. https://nodejs.org/en/, 2020. Accesado: 2020-11-12.

[56] Prabath Siriwardena. *Advanced API Security: OAuth 2.0 and Beyond*. Apress, segunda edition, 1994.

[57] SmartBear Software. Swagger Codegen. https://github.com/swagger-api/swagger-codegen, 2020. Accesado: 2020-06-30.

[58] SmartBear Software. Swagger Editor. https://swagger.io/swagger-ui/, 2020. Accesado: 2020-06-30.

[59] SmartBear Software. Swagger editor. https://github.com/swagger-api/swagger-editor/, 2020. Accesado: 2020-11-12.

[60] SmartBear Software. Swagger UI. https://editor.swagger.io/, 2020. Accesado: 2020-06-30.

[61] SmartBear Software. Swagger.io. https://swagger.io/, 2020. Accesado: 2020-10-26.

[62] SmartBear Software. Swagger hub. https://swagger.io/tools/swaggerhub/, 2020. Accesado: 2020-11-12.

[63] Stack Overflow. Most Polular Technologies, (Programing, Scripting, and Markup Languages). http://shorturl.at/twDK0, 2020. Accesado: 2020-05-30.

[64] StrongLoop, IBM. Express. https://expressjs.com/, 2020. Accesado: 2020-11-16.

[65] Texas Instruments. Simplelink multi-standard cc2650 sensortag. https://www.ti.com/tool/TIDC-CC2650STK-SENSORTAG, 2020. Accesado: 2020-11-15.

[66] The Linux Foundation. Openapi iniciative. https://www.openapis.org/, 2020. Accesado: 2020-11-12.

[67] The Raspberry Pi Foundation. Raspberry pi 3 model b. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, 2020. Accesado: 2020-11-18.

[68] The Raspberry Pi Foundation. Raspberry pi model b v1.2. https://www.raspberrypi.org/about/, 2020. Accesado: 2020-11-18.

[69] Tom Preston Werner. Semantic Versioning. https://semver.org/, 2020. Accesado: 2020-06-30.

[70] Tom Preston-Werner. Semantic versioning 2.0.0 (semver). https://semver.org/spec/v2.0.0.html, 2020. Accesado: 2020-11-14.

[71] Travis Tidwell. Jsencrypt. https://travistidwell.com/jsencrypt/demo/, 2020. Accesado: 2020-11-13.

[72] Wipro. Big data: Catalyzing performance in manufacturing. https://www.wipro.com/content/dam/nexus/en/industries/process-and-industrial-manufacturing/latest-thinking/2606-Big%20Data%20-%20Copy.pdf, 2020. Accesado: 2020-05-30.

[73] World Wide Web Consortium. Simple object access protocol (soap) 1.1. https://www.w3.org/TR/2000/NOTE-SOAP-20000508/, 2013. Accesado: 2020-10-16.

Apéndice A

Introduction

"Aut non tentaris, aut perficie. (Do not attempt it, or take it to the end.)". – OVIDIO, 43 a.C. – 17 d.C.

A.1. Background

API adoption at the enterprise level has exceeded expectations as the proliferation of these was noted in almost all industries. It is not an exaggeration to say that a business without an API is like having a computer without internet access[56]. One of the key aspects behind the success of the Internet of Things (IoT) is the API.

Being that APIs are the key to creating communication channels on the Internet of Things. According to a report by the Wipro Council for Industry Research[72], a six-hour flight in a Boeing 737 from New York to Los Angeles produces 120 terabytes of data that is collected and stored on the plane. With the explosion of sensors and devices taking over the world, there needs to be a proper way to store, manage and analyze data. In 2014 it was estimated that 4 Zettabytes of information were stored worldwide, and it is estimated that by 2020 this number will increase to 35 Zettabytes[4]. The most interesting thing is that 90 % of the data we have is generated only during the last few years. APIs role in the context of the Internet of Things is basically the way that connects devices to other devices and/or to the cloud.

In an article published by Cisco[7] about the Internet of Things, it is estimated that during 2008, the amount of things connected to the Internet exceeded the number of people on Earth.

This world is more connected now than ever before. We share photos on our social networks like Instragam, Facebook, Twitter, etc. In consequence, all this is only possible thanks to the APIs, which have proliferated in the last few years. eBay, Salesforce, Expedia and many other companies generate a great deal of revenue annually with the use of APIs. It is also worth noting the media or interaction profiles that APIs have. For example a Single-Page Applications (SPA), Native Mobile Applications (NMA) from Android or iOS, and Browser-Less Applications, etc.

API security has changed a lot in the last few years. The growth of standards for securing APIs has been enormous. JSON plays an important role in API communication. Most APIs developed today only support JSON, not XML[14].

Business APIs have become the standard way to expose business functions to the outside world via the Internet. Exposing functionality is convenient, but of course it carries a risk of migration. As with any software system, developers often ignore the security element during the design stage of the API. It is only at implementation or integration time that they begin to worry about security.

Security levels that are at risk of transition include authentication and authorization, by which personal data is vulnerable both inside and outside the application, in the case of customers or users. The same happens in the channels through which the data flows, which in turn are constantly attacked.

Finally, to face this problem, we present this final work in which a brief introduction to the REST architecture is integrated. As well as the development of an API of own authorship that will be documented for its optimal operation. Likewise, a security audit will be carried out at the authorization and authentication levels. This will allow the corresponding vulnerabilities to be identified and the API to be corrected. To do this, the pentesting,

documentation and development technologies will be implemented, as well as the technical advice for development at the security level.

A.2. Goal

Developing an API adapted to an IoT environment, as well as auditing its overall performance in terms of safety and documentation.

For this purpose, an infrastructure has been made in which different technologies have been chosen to speed up the development, documentation and auditing of the API security.

The Figure A.1 below shows a diagram that includes three sections of interaction of the API, which is detailed below its main functionality:

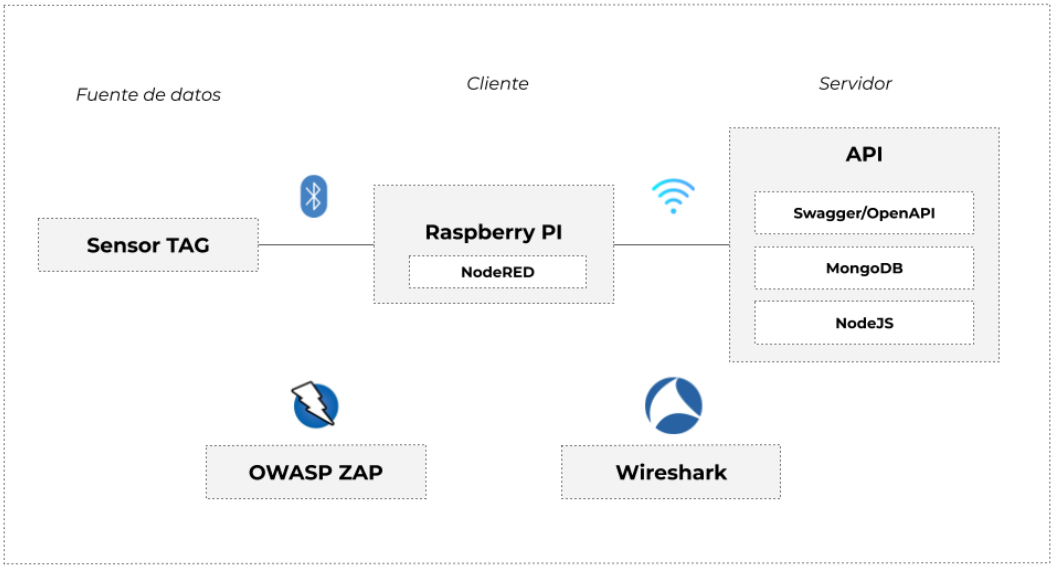

Figura A.1: *Global API audit and communication diagram*

1. *Data source*: The device SensorTAG CC2650 will generate the data related to the *humidity*, *temperature*, *pressure* e *light intensity*. The same data will be sent via BLE to the Raspberry Pi in a given time for each sensor.
2. *Client*: Node RED will work as a client by receiving the data from SensorTAG to the Raspberry Pi and these in turn will be sent by POST requests to the API. The

request header will contain a JWT, so the API will authorize or reject the request. These requests will be sent over Wi-Fi in the local network .

3. *Server*: Manages requests from authorized and authenticated customers to delete, obtain, consult and modify resources. The API is connected to a NoSQL database (MongoDB). Swagger works as a framework to carry out the documentation of the API, which will be implemented in the development. Swagger implements OpenAPI Specification on the server, here are defined the endpoints, schemas, data types, requests and responses of each endpoint.

A.3. Work plan

To carry out the integration of this Final Work in the Figure A.2 shows the activities to be performed. This includes a brief introduction to the REST architecture.

Through the technologies to be implemented. Swagger as documentation technology, Node.js for development, MongoDB to store API data.

To integrate the API, the customer's schemes, sensortag, will be defined. This covers the corresponding data types and validations. Meanwhile, the endpoints will define the type of operations to be performed, as well as their responses that return for each one. This API will have two levels of security, firstly the registered clients must authenticate themselves by email and password, in order to identify themselves and with this the server will respond with a JWT that will allow the authorization of the client's requests and validate them on the server; as a whole this will cover the development and implementation of the API.

ZAP and Wireshark will be used to audit the API already in operation. ZAP will function as a pentesting tool to find the vulnerabilities of the developed API. In this way, we will have vulnerable and secure scenarios. Wireshark will be used to scan the network through which the data will travel, thus allowing us to know the weak points faced by the API, such as the ports, headers and packets that are captured to obtain sensitive API data. In this case, the aim is to break and secure the data in transit between the client and server,

resulting in encrypted communication.

Nombre	Fecha de ini.	Fecha de fin
Arquitectura REST	2/03/20	20/03/20
Antecendentes	2/03/20	11/03/20
Seguridad	12/03/20	20/03/20
Requerimientos	23/03/20	1/05/20
Tecnología de auditoría	23/03/20	1/04/20
Tecnología de documentación	2/04/20	14/04/20
Tecnología de bases de datos	15/04/20	22/04/20
Tecnología de desarrollo	23/04/20	1/05/20
Desarrollo e implementación de la API	4/05/20	26/06/20
Esquemas de la API	4/05/20	15/05/20
Rutas de la API	18/05/20	29/05/20
Autenticación y Autorización de la API	1/06/20	12/06/20
Documentación de la API	15/06/20	26/06/20
Auditoría de la API	29/06/20	24/07/20
Escenarios vulnerables de la API	29/06/20	10/07/20
Escenarios seguros de la API	13/07/20	24/07/20
Conclusiones	27/07/20	31/07/20

Figura A.2: *Work plan activities*

To organize the activities respect to time, the stages and their independent tasks planed to work are presented in Figure A.3.

1. Study the architecture of a REST system.
2. Develop an API in Node.js to manage sensortag resources, manage client security through authorization and authentication adapted to a REST architecture.
3. Document the API using the OpenAPI Specification with the implementation of the Swagger UI framework, integrating it to the developed API.
4. Analyze the different security flaws of a REST architecture, listed by OWASP in the developed API.
5. Analyze and apply the good development practices proposed by the OWASP API Security Top 10 2019 project.
6. Vulnerating and securing the API, by means of brute force attacks, textttman in the middle attack with Wireshark.
7. Secure the communication channels between client and server, by certificates, public

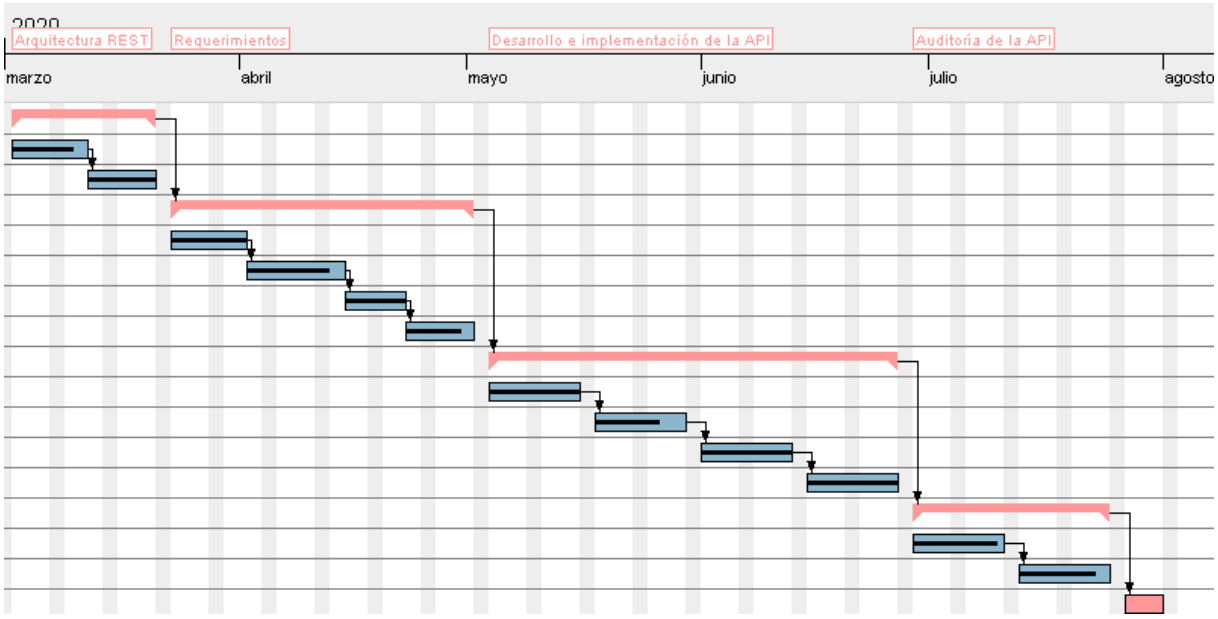

Figura A.3: *Time-based work plan*

and private keys, that allow to sign each message without being altered.

8. Use ZAP as a tool to exploit possible security vulnerabilities in the application and resolve them within the designed system.

Meanwhile in the rest of the document the structure is presented as follows:

- In the **chapter**, 2, a brief introduction of the concepts that involve a REST system's security is made.
- In the **chapter** 3, it describes the set of technologies that are integrated in the development of the API. Aspects of the hardware and software involved in the Final Work are highlighted. As well as the elements that interoperate in the operation of the API.
- In the **chapter**, 4, the entire architecture of the developed API is described.
- In the **chapter** 5, illustrates the general application of OpenAPI Specification in the API through Swagger.
- In the **chapter** 6, the vulnerabilities and safe scenarios are developed for the API.

ZAP is used as a pentesting tool. Best practices according to the OWASP API Security Project Top 10 are also applied.

- In the **chapter** 7, the final conclusions and future work are delimited to extend the technical usability of this project.

For reinforcing the content of the Final Work is presented in the Appendix C the context of the development of the API, which involves the most used security mechanisms, development frameworks, programming languages and some development modules in Node.js and finally is presented in the Appendix D the code of the API developed in Node.js, hosted in GitHub.

Apéndice B

Conclusions and future work

"Solitudinem faciunt pacem appellant. (Donde crean un desierto lo llaman paz.)". – CARL VON CLAUSEWITZ, 1780 – 1831.

In the present work, the infrastructure initially proposed has been fully developed, and all the tasks done to achieve the main objective have been carried out. In this chapter we try to recapitulate, the achievements obtained, and the lines in which it is possible to orientate in terms of audit in security and software development.

B.1. Conclusions

Throughout this Final Project, different technologies were studied to develop an API and were essentially the basis for building the implemented infrastructure. An API was developed to manage resources for a generic sensor that sends authorized REST requests to the server, controls access by clients and/or users through authentication, allowing access to the stored data of the sensor node. In this way, the goal of auditing the API based on OWASP's best practices was achieved and applied to software developed in Node.js.

Secure data transit from the client to the server is extremely important as the communication channel must be encrypted, which improves the confidentiality and integrity of data. The features of the environment in which we are working have very little influence given the scalability and distribution of the system, but it is in the context of development

and/or production where all kinds of vulnerabilities that can compromise the performance of the system are found. In this case, security, development and data storage tests have been carried out locally.

Device and technology integrations during the development of this solution confirm the interoperability in an IoT deployment. APIs are already well known and used in IoT environments, since they are highly scalable in Systems and Technology Services. The option of integrating native applications or web services that can consume the data makes it extremely complete and integrable to the needs of the environment.

A NoSQL database brings flexibility to the system since it allows the integration of multiple devices, which generate different schemes and/or data models without the need to redesign the database.

Vulnerability detection in the development of this API makes it possible to anticipate possible security risks and failures that may affect system performance. By anticipating early on that an API will be attacked, it is possible to assimilate and react automatically and efficiently, which means optimum system performance.

Security audit carried out was based on two parts. The first is the OWASP recommendation best practices described in the OWASP API Security Project Top 10 manual which describes the guidelines for authorization and authentication, which were successfully applied at the software development level. The OWASP Cheat Sheet REST API was also successfully integrated into this development. In the second part, ZAP was used as a high-value pentesting tool, reaching a secure configuration of the API and thus evaluating the possible risks in the header parameters.

At the same time, it was evaluated the possible failures that exist when storing the authorization data on the client side using JavaScript code and the risks involved, such as decryption of the message content and obtaining security tokens. However, a more secure way of signing the JWTs was implemented, using asymmetric encryption, which improves the level of security in the JWTs.

Finally, there is a large amount of software in the IT sector to document the APIs. In the present work, the OAS specifications were integrated as a trend technology to be carried out in the documentation. Swagger was used as a framework for the OAS specifications and thus taking it to the adaptation with the work developed in Node.js. Swagger is a very powerful tool to improve the development and implementation processes of the API, which turns into a better performance and system usability. Swagger adapts to a large number of programming languages.

B.2. Future work

Following the work carried out in this Final Project and after auditing the API, the immediate future line of work must be the extension of the system to more sensor nodes to adapt REST architecture. While its optimum operation has been seen and verified in a local environment, it is very important to verify and alternate it to a real production environment. Distributed systems of the same form are adaptable to this REST architecture. Since authorization, authentication, and database services can be separated. The development technologies employed allow scalability so that it functions as a basis for auditing and documenting future REST architectures.

It is taken into consideration to apply the range of possible authorization formats such as implicit OAtuh[22], by passwords or client credentials. Thus allowing to adjust better results in the API security. It should be noted that the use of tunnels in communication allows further strengthening of security at the level of communication between client and server.

Finally, the use of the JSON format has been on the rise as the preferred technology for data transport. Therefore, the use of JSON Web Encryption[17] and JSON Web Signature[18] has been considered, allowing for the addition of an additional layer of data confidentiality.

Apéndice C

Contexto de las APIs

C.1. Estudio de los mecanismos de seguridad

En la Fig. C.1 se muestra la organización de los mecanismos de autenticación y autorización de las 500 APIs estudiadas. En consecuencia se puede observar que el uso de *API keys* se encuentra como el mecanismo de autenticación más usados mientras *OAuth2.0* se encuentra como el mecanismo de autorización más implementado.

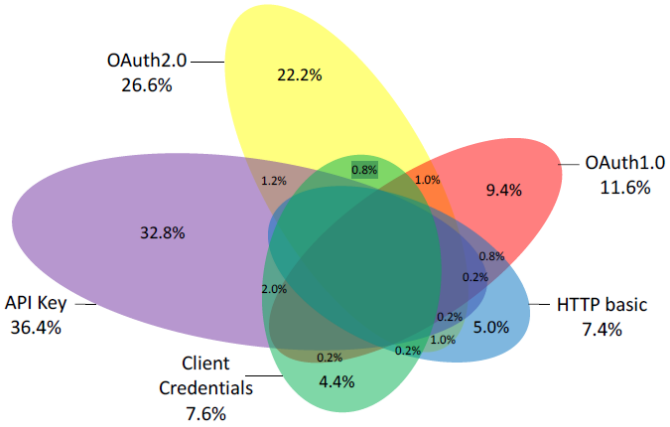

Figura C.1: *Distribución de los diferentes mecanismos de autenticación*[1]

El punto de esta indagación o consulta permite conocer la situación sobre la seguridad aplicada a la mayoría de APIs.

C.2. Frameworks de desarrollo

Dentro de las alternativas de construcción de un API se toma en cuenta varios aspectos relativos tanto a la experiencia en el lenguaje de programación, así como también a la documentación y el soporte continuo disponible al implementar el `Framework`.

En la Tabla C.11 se muestran algunos de los `Frameworks` de desarrollo de APIs más usados.

Lenguaje	Framework	Sitio Web
Java	Spring	https://spring.io/projects/spring-restdocs
NodeJS	Express, Koa, Loopback	https://expressjs.com/, https://koajs.com/, http://apidocs.loopback.io/
Python	Django, Flask	https://www.django-rest-framework.org/, https://flask-restful.readthedocs.io/en/latest/
PHP	Laravel	https://laravel.com/docs/7.x/eloquent-resources
Ruby	Ruby on Rails	https://api.rubyonrails.org/

Tabla C.1: *Frameworks y lenguajes para el desarrollo de una API*

C.3. Lenguajes de desarrollo

En base a un investigación se referencia dos plataformas las cuales se estratifican los distintos lenguajes de programación más usados con distintas métricas de uso. El sistema de control de versiones[10] delimita los lenguajes más usados durante el primer cuatrimestre del año 2020 en la Fig. C.2.

De igual forma se muestra como referencia[63], siendo que en el año 2019 se realizo un estudio en la cual muestrán las tecnologías de programación más usadas por desarrolladores profesionales esto se muestra en la Fig. C.3.

2020 1

# Ranking	Programming Language	Percentage (Change)	Trend
1	JavaScript	18.703% (-1.406%)	
2	Python	16.238% (-1.654%)	
3	Java	10.938% (+0.538%)	
4	Go	9.005% (+0.978%)	
5	C++	7.423% (+0.040%)	
6	Ruby	6.812% (+0.342%)	
7	TypeScript	6.769% (+1.522%)	^
8	PHP	5.127% (-0.458%)	v
9	C#	3.835% (+0.141%)	
10	C	3.181% (-0.203%)	

Figura C.2: *Ranking según Github*

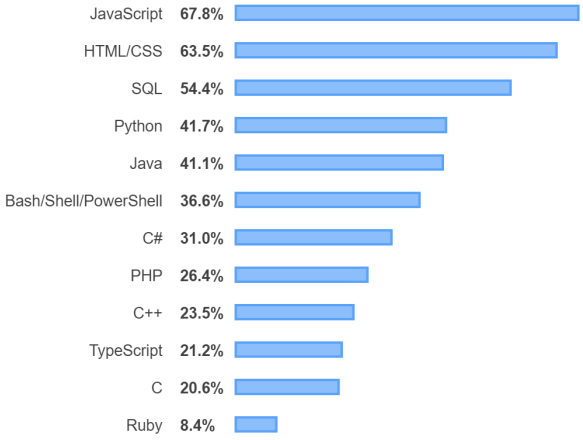

Figura C.3: *Ranking según Stack Overflow*

C.4. Módulos para la documentación

No los compararemos porque no es algo fácil de hacer considerando que algunos módulos sólo manejan una cosa, mientras que otros se encargan de varias cosas. Así que después de repasar entonces, propondré un combinación de estos módulos, pero tendrás suficiente información para elegir una combinación diferente si se ajusta tu problema mejor.

Categoria	Gestor de solicitudes/respuestas, gestor de rutas
Versión Actual	20.0.0
Sitio Web	https://hapi.dev/
Instalación	`Instalación: $ npm i @hapi/hapi`. Para conseguir que la dependencia se añada automáticamente al fichero package.json: `$ npm i -- save @hapi/hapi`

Tabla C.2: *Información del Framework HAPI*

Categoria	Gestor de solicitudes/respuestas, gestor de rutas, middleware
Versión Actual	4.17.1
Sitio Web	https://expressjs.com/
Instalación	`Instalación: $ npm i express`. Para conseguir que la dependencia se añada automáticamente al fichero package.json: `$ npm i -- save express`

Tabla C.3: *Información del Framework Express*

Categoria	Gestor de solicitudes/respuestas, gestor de rutas, middleware
Versión Actual	8.5.2
Sitio Web	https://mcavage.me/node-restify/
Instalación	`Instalación: $ npm i restify`. Para conseguir que la dependencia se añada automáticamente al fichero package.json: `$ npm i -- save restify`

Tabla C.4: *Información del Framework Restify*

Categoria	Gestor de solicitudes y respuestas, middleware, gestión de rutas.
Versión Actual	1.5.0
Sitio Web	https://github.com/deleteman/vatican
Instalación	`Instalación: $ npm i vatican`. Para conseguir que la dependencia se añada automáticamente al fichero package.json: `$ npm i -- save vatican`

Tabla C.5: *Información del Framework vaticam*

Categoria	Instrumento de documentación
Versión Actual	2.1.3
Descripción	Este es un módulo para el Express. Se integra en una aplicación de Express y proporciona las funcionalidades que hace Swagger 3 para documentar las API, que es una interfaz web con documentación de cada método y la posibilidad de probar estos métodos.
Sitio Web	https://github.com/swagger-api/swagger-node-express
Instalación	`Instalación: $ npm i swagger-node-express`. Para conseguir que la dependencia se añada automáticamente al fichero package.json: `$ npm i -- save swagger-node-express`

Tabla C.6: *Información del Framework swagger-node-express*

Categoria	Instrumento de documentación
Versión Actual	0.0.1
Descripción	I/O Docs es un sistema de documentación en vivo diseñado para RESTful APIs. Al definir la API usando el esquema JSON, I/O Docs genera una interfaz web para probar la API.
Sitio Web	https://github.com/mashery/iodocs
Instalación	`Instalación: $ npm i iodocs`. Para conseguir que la dependencia se añada automáticamente al fichero package.json: `$ npm i -- save iodocs`

Tabla C.7: *I/ODocs*

Categoria	Modulo de respuestas hipermedia
Versión Actual	3.0.0
Descripción	Halson es un módulo que ayuda a crear objetos JSON compatibles con HAL, que luego podrá utilizar como parte de la respuesta en su API.
Sitio Web	https://github.com/seznam/halson
Instalación	`Instalación: $ npm i halson`. Para conseguir que la dependencia se añada automáticamente al fichero package.json: `$ npm i -- save halson`

Tabla C.8: *Halson*

Categoria	Modulo de respuestas hipermedia
Versión Actual	1.2.0
Descripción	HAL es una alternativa a HALSON. Proporciona una interfaz más simple pero la misma funcionalidad subyacente: abstrae el formato HAL+JSON y le da al desarrollador una forma fácil de usarlo.
Sitio Web	http://stateless.co/hal_specification.html
Instalación	`Instalación: $ npm i hal`. Para conseguir que la dependencia se añada automáticamente al fichero package.json: `$ npm i -- save hal`

Tabla C.9: *Hal*

Categoria	Validación de la solicitud/respuesta
Versión Actual	0.8.23
Descripción	Este módulo valida la estructura y el contenido de un objeto JSON frente a un esquema predefinido que sigue el formato del Esquema JSON.
Sitio Web	https://www.npmjs.com/package/json-gate
Instalación	`Instalación: $ npm i json-gate`. Para conseguir que la dependencia se añada automáticamente al fichero package.json: `$ npm i -- save json-gate`

Tabla C.10: *Información del módulo JSON-Gate*

Categoria	Validación de la solicitud/respuesta
Versión Actual	1.3.0
Descripción	Este módulo valida la estructura y el contenido de un objeto JSON frente a un esquema predefinido que sigue el formato del Esquema JSON.
Sitio Web	https://github.com/geraintluff/tv4
Instalación	`Instalación: $ npm i tv4`. Para conseguir que la dependencia se añada automáticamente al fichero package.json: `$ npm i -- save tv4`

Tabla C.11: *Información del módulo Tiny Validator (v4 JSON Schema)*

Apéndice D

API Implementada

Los códigos relativos a la implementación del API v1.0 IoT[11] están alojados en la plataforma de Github en la cual podrá ser accesible para futuras consultas y trabajo continuo de este proyecto.