UNIVERSITAT DE BARCELONA

Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica

Universitat de Barcelona

Desenvolupament d’una web segura

Desarrollo de una web segura

Marc Ferré Gras

Director: Raúl Roca Cánovas

Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, 13 de juny de 2023

Resumen

En este trabajo de final de grado, se ha desarrollado, analizado e implementado una aplicación web utilizando las tecnologías Vue y Python, esta última junto con el framework Flask.

El enfoque principal de la aplicación web se centra en la ciberseguridad. Se han identificado y estudiado las principales vulnerabilidades que pueden ocurrir en este tipo de aplicaciones. Se ha investigado cómo funcionan estas vulnerabilidades y las estrategias comúnmente utilizadas para prevenirlas o mitigar su impacto. En la memoria del trabajo se proporciona una explicación detallada de los principales vectores de ataque, se ofrecen ejemplos de cada uno de ellos y se muestra la estrategia que ha sido aplicada para contrarrestarlos en la aplicación desarrollada. Para llevar a cabo esta tarea, se han seguido las recomendaciones y pautas proporcionadas por OWASP, una organización reconocida en el campo de la seguridad informática, que desarrolla una lista de las 10 vulnerabilidades más comunes y proporciona guías sobre cómo enfrentarlas.

Tanto el desarrollo del proyecto como la elección del tema se han realizado con el objetivo de obtener una nueva perspectiva, herramientas y conocimientos que puedan aplicarse en futuros proyectos o trabajos relacionados con el desarrollo de software, dada la importancia de la ciberseguridad en todas las etapas del desarrollo, de forma independiente a la especialización o el ámbito específico del empleo, a causa del contexto actual, con un mundo cada vez más interconectado y digital, haciendo que la protección de datos y sistemas se haya vuelto un tema fundamental.

Resum

En aquest treball de final de grau, s'ha desenvolupat, analitzat i implementat una aplicació web utilitzant les tecnologies Vue i Python, aquesta última juntament amb el framework Flask.

L'enfocament principal de l'aplicació web se centra en la ciberseguretat. S'han identificat i estudiat les principals vulnerabilitats que es poden produir en aquest tipus d'aplicacions. S'ha investigat com funcionen aquestes vulnerabilitats i les estratègies comunament utilitzades per prevenir-les o mitigar-ne l'impacte. A la memòria del treball es proporciona una explicació detallada dels principals vectors d'atac, s'ofereixen exemples de cadascun i es mostra l'estratègia que s'ha aplicat per contrarestar-los en l'aplicació desenvolupada. Per dur a terme aquesta tasca, s'han seguit les recomanacions i pautes proporcionades per OWASP, una organització reconeguda al camp de la seguretat informàtica, que desenvolupa una llista de les 10 vulnerabilitats més comunes i proporciona guies sobre com enfrontar-les.

Tant el desenvolupament del projecte com l'elecció del tema s'han realitzat amb l'objectiu d'obtenir una perspectiva nova, eines i coneixements que es puguin aplicar en futurs projectes o treballs relacionats amb el desenvolupament de software, a causa de la importància de la ciberseguretat a totes les etapes del desenvolupament, de forma independent a l'especialització o l'àmbit específic de la feina, a causa del context actual, amb un món cada cop més interconnectat i digital, de manera que la protecció de dades i sistemes s'ha tornat un tema fonamental.

Resume

In this final degree project, a web application has been developed, analyzed, and implemented using the Vue and Python technologies, with Python being used alongside the Flask framework.

The main focus of the web application revolves around cybersecurity. The primary vulnerabilities that can occur in such applications have been identified and studied. The operation of these vulnerabilities and commonly employed strategies to prevent or mitigate their impact has been investigated. The project report provides a detailed explanation of the main attack vectors, presents examples of each of them, and showcases the strategy that has been implemented to counteract them in the developed application. To accomplish this task, the recommendations and guidelines provided by OWASP, a recognized organization in the field of computer security, have been followed. OWASP develops a list of the top 10 most common vulnerabilities and provides guidance on how to address them.

Both the project development and the choice of the topic have been carried out with the objective of obtaining a new perspective, tools, and knowledge that can be applied in future projects or work related to software development , due to the importance of cybersecurity in all stages of development, regardless of the specialization or the specific job field, given the current context of an increasingly interconnected and digital world. Data and system protection have become fundamental topics.

ÍNDICE

Introducción, motivación y contexto

En el contexto actual, las aplicaciones web desempeñan un papel muy importante en nuestro día a día, ya que están presentes en muchas de las acciones que llevamos a cabo, desde servicios bancarios y compras en línea hasta redes sociales, aplicaciones de empresa o actividades médicas. Con esto aparece también una mayor capacidad e interés por parte de los delincuentes en realizar ataques cibernéticos y, por tanto, resulta crucial desarrollar aplicaciones web seguras que protejan la integridad, la confidencialidad y la disponibilidad de la información.

La falta de seguridad en las aplicaciones web puede tener consecuencias graves debido a que los ciberdelincuentes pueden aprovechar vulnerabilidades en el código o en la configuración de la aplicación para llevar a cabo diferentes tipos de ataque, ataques que pueden tener un impacto devastador tanto para los usuarios de la aplicación como para la reputación de la organización que la desarrolla.

En este contexto, surge la necesidad de desarrollar una web segura que proteja la información y brinde una experiencia confiable a los usuarios. Para lograrlo, es fundamental seguir las mejores prácticas y recomendaciones establecidas por organizaciones de seguridad reconocidas como OWASP (Open Web Application Security Project). Por este motivo, como futuro ingeniero informático y posiblemente desarrollador web, quería profundizar en los posibles ataques a los que me puedo enfrentar en un futuro y así mejorar la seguridad de mis futuras implementaciones.

Objetivos

- Analizar los principios y mejores prácticas de seguridad en el desarrollo web.
- Estudiar las principales vulnerabilidades presentes en el desarrollo web
- Estudiar la guía de OWASP (Open Web Application Security Project) como referencia para asegurar la web.
- Desarrollar una aplicación siguiendo las medidas de seguridad recomendadas por OWASP para mitigar los riesgos y vulnerabilidades comunes.
- Realizar pruebas de seguridad para verificar la robustez de la aplicación y corregir posibles vulnerabilidades.

Planificación

- Planificación prevista

Una vez elegido, como tema del proyecto, el desarrollo de una web segura y dado que en el currículum de estudios la ciberseguridad estaba prevista para el segundo semestre del curso. Ante la necesidad de obtener unas bases de conocimiento previo a partir de las cuales poder empezar a definir el proyecto, se eligió, a propuesta del tutor, la lectura del libro ***Máximo Fernández, Puesta en producción segura. 2022*** [1] con una primera previsión de finalizarla en enero de 2023 y así conseguir que el trabajo previo estuviera finalizado al inicio del segundo semestre (Figura 1.1).

Posteriormente, era imprescindible la búsqueda de información relacionada con el proyecto en el sentido de qué pasos hay que seguir para crear una aplicación de este tipo, tecnologías a aplicar, qué herramientas utilizar, definir el lenguaje de programación más adecuado, donde y como implementar… Se planificó dedicar 1 mes a esta recogida de información y toma de decisiones (Figura 1.2).

Una vez analizada la información y habiendo tomado decisiones al respecto de con qué herramientas y siguiendo que criterios desarrollaría la web, el paso siguiente sería la implementación del código durante los 2 meses posteriores (Figura 1.3), para finalmente realizar la memoria el último mes (Figura 1.4).

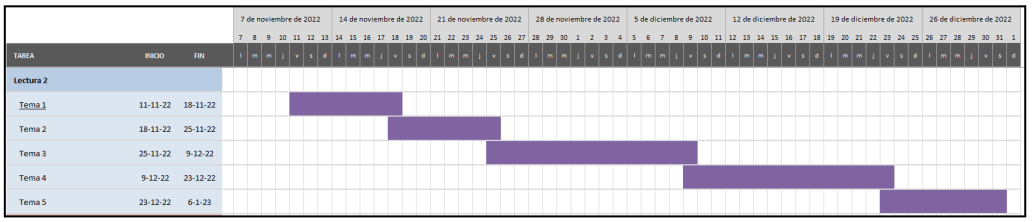


Figura 1.1 *Diagrama de Gantt P1*

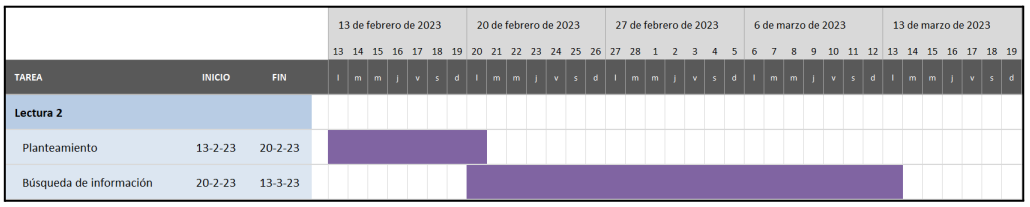


Figura 1.2 *Diagrama de Gantt P2*

Figura 1.3 *Diagrama de Gantt P3*

Figura 1.4 *Diagrama de Gantt P4*

- Calendario real

En cuanto a la primera parte (Figura 1.1) de documentación previa, se cumplieron los plazos establecidos, acabando el libro entero a inicios de año, el segundo bloque (Figura 1.2) costó más de lo previsto, retrasando así la implementación. A causa de esto, la realización de la memoria acabo siendo más tardía de lo esperado, dejando poco margen de tiempo. Además, hubo algunas complicaciones con la puesta en producción del proyecto en Azure.

Tecnologías

Para la realización de este proyecto, las tecnologías que finalmente se han decidido utilizar han sido las siguientes:

Vue.js, es un framework, de JavaScript, utilizado para construir interfaces de usuario interactivas y dinámicas en el lado del cliente [2].

Python, es un lenguaje de programación versátil y de alto nivel que se utiliza ampliamente en el desarrollo de aplicaciones web [3]. Junto a Python se ha usado **Flask** que es un framework diseñado para ser simple y fácil de usar, y se utiliza comúnmente para crear aplicaciones web y APIs [4].

SQL, es un lenguaje de programación diseñado para administrar y manipular bases de datos relacionales que organiza y almacena datos relacionados entre sí mediante claves. Permite realizar diversas operaciones en una base de datos y hay varios tipos de base de datos SQL como MySQL, Oracle, SQLite… [5] En el desarrollo del proyecto se ha usado SQLite en local y PostgreSQL en producción gracias a la utilización de Flask-SQLAlchemy [6].

SQLAlchemy es un ORM de Python. Proporciona una forma intuitiva y flexible de interactuar con bases de datos relacionales utilizando código en lugar de escribir consultas SQL directamente, facilitando la gestión.

Azure, una plataforma en la nube desarrollada por Microsoft que proporciona un conjunto de servicios y herramientas, permitiendo a los usuarios ejecutar aplicaciones y almacenar datos de manera segura. En el proyecto, concretamente, se ha usado para alojar la aplicación en la red, con los recursos que se han necesitado como Azure PostgreSQL Database o App Service [7].

Enlace a la web: https://provamarctfg.azurewebsites.net/

Análisis y desarrollo

Para realizar la implementación de la aplicación web se ha usado como referencia el top 10 de OWASP, ya que **OWASP** es una fundación creada con el objetivo de determinar y evitar las causas que hacen que el software sea inseguro proporcionando proyectos, documentos y herramientas, siendo **OWASP TOP 10** su proyecto más conocido. Este es una lista de las 10 vulnerabilidades de seguridad más comunes en las aplicaciones web y junto a esta lista también proporciona una guía que indica como evitarlas [8], la cual se ha seguido, ya que emplea métodos contrastados por parte de profesionales.

A continuación, se exponen las implementaciones aplicadas al proyecto y la teoría o información relacionada con cada una.

1- Fallas de identificación y autenticación

Es una vulnerabilidad que se produce cuando hay fallas en la identificación y la autenticación que pueden ser aprovechadas por atacantes para acceder a cuentas de usuario sin permiso o para realizar acciones maliciosas en nombre de otros usuarios legítimos.

La **identificación** se refiere al acto de proporcionar una identidad única para distinguir a un individuo o entidad dentro de un sistema y la **autenticación** se refiera al proceso de verificar la identidad de un usuario antes de permitirle acceder a determinados recursos, se basa en la presentación de pruebas o credenciales que demuestren la validez de la identidad a demostrar, normalmente se usa un identificador (nombre de usuario, correo electrónico, etc.) y una contraseña [9].

1.1 Gestión de autenticación: Sesiones y tokens

Hay diferentes enfoques para gestionar la autenticación de usuarios, permitiendo que estos no deban introducir sus credenciales cada vez que se hace una solicitud. Estos métodos se basan en pasar información que permite autenticar una de las solicitudes, y dos de los más usados son el uso de sesiones y el uso de tokens [10] [11].

Sesiones y Cookies

La autenticación basada en sesiones y cookies es la implementación tradicional y es con estado, es decir, al iniciar sesión, después de que un usuario envíe sus credenciales y se validen, se guarda información sobre la sesión del usuario en el lado del servidor y se asocia con una cookie (sesión ID) en el navegador del usuario. De esta forma, no es necesario enviar las credenciales en cada petición y se autoriza comparando la cookie del navegador con la información guardada en el servidor.

Tokens

Con la evolución de la tecnología y la necesidad de realizar integración con aplicaciones de terceros, se han ido desarrollando sistemas más complejos que el tradicional de sesiones. Estos sistemas utilizan criptografía y uno de los más habituales es el uso de tokens. Los tokens pueden tener distintos formatos, pero el más común es JSON Web Token (JWT) que usando objetos en formato JSON, permiten guardar cualquier tipo de metadata, siempre que se trate de un JSON válido.

Este tipo de token consta de tres partes que son: el *header*, que incluye el tipo de token y el algoritmo para firmar, el *payload* que incluye la información y *signature* para verificar la autenticidad e integridad. Con esto, el token se forma firmando el *header* y el *payload* (codificados en base 64), usando el algoritmo del header y una clave privada para así detectar manipulaciones después de enviar el token al cliente.

Comparación entre sesiones y tokens

Los JWT se utilizan ampliamente gracias a su escalabilidad, ya que pueden ser implementados sin estado, con lo que no guardan datos en el servidor, lo que podría ser un problema cuando un gran número de usuarios están accediendo a la aplicación a la vez, es decir, para verificar un token se usa únicamente la clave y el propio token, mientras que para verificar la sesión se debe acceder a la base de datos (o el sistema implementado). La mayor desventaja de la autenticación basada en tokens es el tamaño de los JWT, ya que una cookie de sesión es pequeña comparada con un token

y otra desventaja es que los JWT tienen un tiempo de expiración y en un principio no se pueden invalidar.

Implementación de la gestión de autorización en la aplicación

Analizadas las diferentes posibilidades de autenticación se decide utilizar JWT porque está más enfocado a sistemas modernos actuales, además se tenía experiencia previa utilizando este tipo de tokens. En el proyecto que se desarrolla en este trabajo, los tokens se han implementado teniendo más en cuenta la seguridad y los consejos que OWASP proporciona para estos [12].

Creación del token

Para la creación de tokens es habitual el uso de alguna biblioteca, y, por tanto, requiere elegir cuál de ellas usar, teniendo en cuenta que en muchas existe una vulnerabilidad que deja **modificar el algoritmo de firma**, permitiendo así manipulaciones y tokens falsos [13].

El error consiste en que al intentar verificar el token, se obtiene el algoritmo a usar a partir del header. Este campo se lee antes de realizar ninguna verificación, ya que es el paso previo a esta, permitiendo a un atacante editar este campo y de este modo influir en la verificación posterior.

Esta lógica de verificación genera 2 vías de ataque que se exponen a continuación:

- Aparte de los algoritmos de firma mencionados, existe un algoritmo None que se diseñó para casos en los que ya se ha verificado el token. El problema aparece cuando algunas bibliotecas identifican como válidos los tokens firmados con None como si no hubieran sido protegidos, provocando que un atacante al modificar el algoritmo del header y poner None pueda generar tokens interpretados como válidos.

- La otra vía de ataque, se da cuando se aprovecha el hecho de que los JWT se pueden cifrar con algoritmos simétricos o asimétricos. RSA al ser asimétrico, utiliza una clave privada para firmar y una clave pública para verificar la firma y, por otro lado, HMAC que es simétrico, utiliza una misma clave secreta compartida para firmar y verificar. Con esto en mente, existe una vulnerabilidad en la que los atacantes pueden aprovechar esa característica, de manera que, cuando un servidor está preparado para recibir un token JWT firmado con RSA, pero en su lugar recibe un token firmado con HMAC, al verificar la firma, intentará usar la clave pública como si fuera la clave

secreta de HMAC. La clave pública en el algoritmo RSA es conocida por todos y se utiliza para verificar la autenticidad de la firma, sin embargo, la clave secreta de HMAC debe mantenerse privada, ya que se utiliza tanto para firmar como para verificar la integridad del token. Si el servidor asume que la clave pública es en realidad una clave secreta de HMAC, esto significará que la clave secreta de HMAC se ha expuesto a cualquier persona que tenga acceso a la clave pública. En consecuencia, un atacante malintencionado podría obtener la clave pública y usarla para generar tokens JWT falsificados que el servidor aceptará como válidos.

Estos dos ataques pueden conducir a graves vulnerabilidades de seguridad, como la suplantación de identidad, la manipulación de datos y el acceso no autorizado a recursos protegidos.

La solución para evitar esto es usar una biblioteca que permita especificar de forma explícita el algoritmo a usar al descodificar el token. Con esto el servidor al intentar validar usará el algoritmo indicado en el método en lugar de fiarse del header.

Para desarrollar la web del trabajo se decidió usar la biblioteca PyJWT [14] que permite hacer esta especificación explicita, de modo que al llamar al método decode y encode de esta biblioteca se especifica el algoritmo a usar, HS256 (Figura 2).

```
data = decode(token, secret2, algorithms=["HS256"])
```

Figura 2 *Uso del método decode de PyJWT*

La siguiente decisión era determinar el contenido a incluir en el *payload*. En un primer momento, se pensó añadir únicamente el ID del usuario y el tiempo de expiración, finalmente se optó por añadir también un contexto para dificultar la **apropiación secundaria de tokens**

Se refiere a cuando un token es robado por un atacante y a través de este puede usarlo para hacerse pasar por el usuario, pudiendo así hacer operaciones en su nombre. Una forma de prevenir esto, siguiendo las recomendaciones, es agregar un *"contexto de usuario"* en el token. Este contexto consiste en una cadena aleatoria que se generará durante la autenticación a la vez que el token, y se enviará al cliente como una cookie reforzada. Para generar la cadena (Figura 3) se decide utilizar la biblioteca *secrets* [15] explicada en el apartado de criptografía.

```
contexto_usuario = ''.join(secrets.choice(caracteres) for i in range(20))
```

Figura 3 *Generación de cadena aleatoria*

Una vez generado el secreto, se crea la cookie con las características necesarias para que esta sea segura (Figura 4).

```
response.set_cookie('ctx', contexto_usuario, samesite='Strict', secure=True, httponly=True,max_age=t)
```

Figura 4 *Añadir cookie segura*

Una cookie reforzada se considera igual de segura que un ID de sesión utilizada en el sistema de sesión tradicional. Esto se debe a las características de esta cookie que incluye: *HttpOnly* para indicar que la cookie solo debe ser accesible a través del protocolo HTTP y no mediante scripts del lado del cliente, lo que ayuda a prevenir ataques de XSS, *Secure* que indica que la cookie solo debe enviarse a través de conexiones HTTPS seguras, evitando que se transmita en texto plano y protegiendo contra ataques de interceptación de redes, *SameSite*, que permite controlar si la cookie debe enviarse en solicitudes de terceros (sitios externos), lo que ayuda a prevenir ataques de CSRF y *Max-Age*, para establecer un tiempo de vida máximo para la cookie [16], lo que limita la ventana de oportunidad para un atacante en caso de que la cookie sea interceptada o robada.

Aparte de añadir la cookie, el contexto debe también incluirse en el propio token (Figura 5), pero codificado con un hash SHA256 (Figura 6) para evitar que el atacante lea el valor del contexto y pueda establecer la cookie segura el mismo.

```
token = encode(
    {"id": id,
     "exp": int(time.time()) + expiration,
     "hash_contexto_usuario": contexto},
    secret1,
    algorithm="HS256"
)
```

Figura 5 *Crear token con contexto*

```
hmac.new(current_app.secret_key.encode('utf-8'), contexto_usuario.encode('utf-8'),
                                                        hashlib.sha256).hexdigest()
```

Figura 6 *Crear hash del contexto*

Teniendo el contexto de la cookie y el del token, al verificar este token se añade una comparación entre ambos contextos y si no coinciden se considera como un token inválido, haciendo que se necesite el token junto a la cookie reforzada para poder usar el JWT.

Una vez creado el token también es una buena práctica encriptarlo para evitar la **divulgación de información de tokens** que consiste en que en caso de que un atacante logre obtener el token, este contiene información que se puede leer fácilmente, ya que este contenido está codificado en base64 y, por tanto, es muy sencillo de descodificar, por eso para evitarlo se propone como solución encriptar el propio token antes de enviarlo al cliente usando el algoritmo AES-GCM (Figura 7), recomendado por OWASP.

```
ssgcm = AESGCM(str.encode(secret2))
    nonce = secrets.token_bytes(12)
    ciphertext = aesgcm.encrypt(nonce, token_b, None)
    token_bytes = nonce + ciphertext
    token = token_bytes.hex()
```

Figura 7 *Encriptar token*

Otra de las debilidades que se deben tener en cuenta es que los tokens **no incluyen revocación,** JWT un token solo deja de ser válido al caducar.

Según OWASP una solución a este problema nos la proporciona la implementación mencionada anteriormente, el contexto. Aprovechando la seguridad de la cookie reforzada y 'simulando' un cierre de sesión borrando el JWT del lado del cliente, si el usuario elige cerrar el navegador, tanto la cookie como el almacenamiento de sesión se borrarán automáticamente. Otra forma de protegerse contra esto es implementar una lista de tokens bloqueados, de manera que cuando el usuario cierre sesión se agregue el token de usuario a una lista de bloqueo, lo que resultará en una invalidación inmediata del token para su uso posterior en la aplicación.

En el caso de la implementación de contexto, la parte positiva es que no requiere prácticamente modificaciones, pero realmente el token sigue siendo válido, aunque ya no se encuentre en el cliente, por lo tanto, en caso de que un atacante consiga de algún modo la cookie y el contexto, podría seguir usando el token para autenticarse falsamente hasta que este caduque. En cuanto a la segunda opción, la parte positiva es que guardar los tokens revocados en una lista de bloqueo asegura que en caso de que el usuario cierre sesión, este no se pueda reutilizar aunque se produzca un robo, pero lo malo es que obliga mantener un estado en el servidor.

La decisión final para este trabajo ha sido la de no usar la lista de bloqueados y mantener la recomendación de OWASP del contexto junto al token, que se borra del cliente al cerrar sesión (también al cerrar pestaña o navegador). Esta decisión ha sido tomada para no mantener un estado en el servidor y también porque al intentar la implementación de la lista de bloqueados no se conseguía hacer que al cerrar ventana se bloqueara el token y solo se hacía al cerrar sesión de forma explícita.

Una vez definida la creación del token y qué características tendrá, es importante asegurar el envío y almacenamiento de este.

Enviar y guardar el token

El token se puede enviar al cliente de distintas formas, como por ejemplo en el encabezado de autorización, en una cookie, en el cuerpo de la solicitud o en un parámetro de consulta (Query Parameter).

En el proyecto se decide hacerlo a través del encabezado de autorización, ya que enviarlo en el cuerpo de la solicitud o en el parámetro de consulta de la URL puede hacer que quede expuesto en los registros del servidor, en la barra de direcciones del navegador o en otros sistemas de almacenamiento temporal. Además, la cookie se descarta, debido a que debe estar en un almacenamiento distinto al contexto para evitar las mismas vulnerabilidades y fortalezas. Por lo tanto, la mejor opción en este caso y una de las más habituales es enviar el token a través del header Authorization (Figura 8) que está especialmente diseñado para autorización y autenticación, siendo menos propenso a quedar expuesto.

```
response.headers['Authorization'] = f'Bearer {token}'
```

Figura 8 *Añadir el token al header Authorization*

Después de ser enviado se debe determinar donde se **guardará** para estar disponible en diferentes componentes o páginas del cliente. Las opciones más comunes son en una cookie, en Local Storage o en Session Storage. Como en el caso anterior , la cookie queda descartada y se elige Session Storage (Figura 9), porque aporta una mayor seguridad, eliminando el token al cerrar la pestaña del navegador, mientras que con Local persiste incluso después de que el usuario cierre o reinicie el navegador.

```
sessionStorage.setItem('token', res.headers.authorization.split(' ')[1])
```

Figura 9 *Guardar el token en Session Storage*

Al almacenar el token en Session Storage en lugar de en una cookie reforzada, se expone el token a ser robado a través de XSS, pero con el contexto guardado en la cookie aun robando el token no se podría usar, aumentando la seguridad con la combinación de ambos.

Una vez el token está disponible en el cliente, es enviado al servidor en cada petición (se podría solo incluir en algunas y otras no) usando también el header de Authorization y verificando su autenticidad al ser recibido por el servidor.

Verificación del token

Para poder verificar el token lo primero es desencriptarlo (Figura 10), ya que está encriptado con AESGCM, después se aplica decode (Figura 11) que descodifica el token y realiza la verificación, comprobando la firma del token utilizando la clave secreta y comprobando si ha expirado.

```
aesgcm = AESGCM(k)
token_b = aesgcm.decrypt(nonce, ciphertext, None)
```

Figura 10 *Desencriptar token*

```
data = decode(token, secret2, algorithms=["HS256"])
```

Figura 11 *Usar el método decode*

En caso de que todo sea válido, se pasa a realizar la última comprobación antes mencionada, que consiste en obtener el hash del contexto guardado en el *payload* y el contexto de la cookie, codificando este último y comprobando si los dos *hashes* son iguales. Si esto también es correcto, se usa el ID guardado en el *payload* para identificar al usuario autenticado, lo que permitirá la autorización.

1.2 Pautas generales de autenticación: ID usuario y contraseñas

Uno de los puntos más importantes en la autenticación son las credenciales.

Para iniciar sesión se pide un nombre de usuario y una contraseña que deberán seguir unas pautas para aumentar su seguridad [17].

ID de usuario

En cuanto al ID de usuario (nombre de usuario), debe ser único. Para asegurar esto, al registrar un usuario se comprueba que no exista otra cuenta con el mismo nombre (Figura 12), además también se especifica en la tabla de la base de datos que el valor debe ser único (Figura 13), bloqueando la inserción de usuarios con el mismo nombre.

```
acc1 = AccountsModel.get_by_username(username)
if(not acc1):
```

Figura 12 *Comprobación de usuario con el mismo nombre*

```
username = db.Column(db.String(30), unique=True, nullable=False)
```

Figura 13 *Argumento username de la base de datos*

También es recomendable evitar que distingan entre mayúsculas y minúsculas, aunque en muchos sistemas se consideran diferentes y se tratan como caracteres distintos, puede generar confusiones. Para hacer esto, al crear una cuenta y al iniciar sesión se usa el método *casefold* (Figura 14) que devuelve una cadena de caracteres en minúsculas.

```
username = username.casefold()
```

Figura 14 *Uso de casefold*

Contraseñas

La otra parte fundamental del inicio de sesión son las contraseñas que deben ser secretas y no adivinables, por eso es importante aplicar una política de **contraseñas "fuertes"**. Con el objetivo de dificultar así la adivinación con pruebas manuales o incluso con herramientas automáticas que se pueden combinar con recursos como diccionarios de palabras comunes o listas de contraseñas filtradas.

Para esto, al crear una contraseña, esta debe cumplir ciertos requisitos, como tener una longitud menor de 65, pero superior a 7, donde el tamaño mínimo es necesario, ya que por debajo de 8 caracteres una contraseña se considera débil y aumenta así las probabilidades de ser adivinada, por otro lado, el tamaño máximo es necesario para evitar ataques de denegación de servicio, debido a que se usan algoritmos hash para ocultar las contraseñas al guardarlas y algunos de estos tienen una limitación de largo. Además, aplicar hash a una contraseña muy larga puede llegar a provocar un agotamiento de memoria y CPU.

También es recomendable que las contraseñas puedan estar formadas por cualquier carácter, incluidos unicode (por ejemplo é, ë) y espacios en blanco para así aumentar el número de combinaciones posibles, incluso es común obligar al usuario a usar caracteres diferentes como letra mayúscula, minúscula, número y carácter especial para así asegurar una mayor seguridad.

Siguiendo las premisas comentadas, las características que se ha decidido implementar para la creación de contraseñas en la aplicación desarrollada en este trabajo son las que se muestran en la Figura 15.

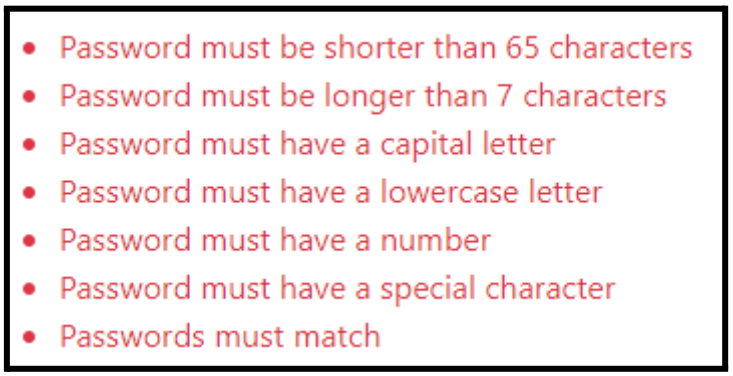


Figura 15 *Requisitos de contraseña*

Para dar más información al usuario sobre su propia contraseña se ha implementado una **barra** (Figura 16) que tiene como función medir el grado de seguridad de la contraseña.

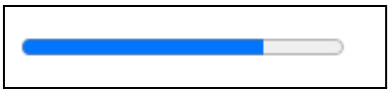


Figura 16 *Barra de fuerza de contraseña*

Finalmente, otra práctica recomendada consiste en evitar el uso de contraseñas que han sido comprometidas. Para hacerlo se usa un servicio externo, *Pwned Passwords*, que contiene un registro de **contraseñas filtradas** (Figura 17).

```
result = pwnedpasswords.check(password, plain_text=True)
if result:
    return {'message': "Password was compromised"}, 401
```

Figura 17 *Uso de Pwned Passwords*

Además de la creación de contraseñas fuertes, se deben aplicar más capas de protección para evitar o dificultar los ataques a contraseñas. En el proyecto se ha añadido una **limitación de repetición** para evitar la repetición excesiva de algunas acciones, debido a que un atacante podría ir probando múltiples contraseñas para ver si en algún caso coincide con la real y para implementarlo se usa un *limiter* (Figura 18) que evita la realización de más de 5 intentos de inicio de sesión por minuto.

```
@limiter.limit("5/minute")
```

Figura 18 *Implementación de un limiter*

También se ha añadido **autenticación de doble factor** [18] que es un método de seguridad que requiere dos formas diferentes de verificar la identidad de un usuario antes de permitir el acceso a una cuenta o sistema, pidiendo, además de la contraseña tradicional, un segundo factor para agregar una capa adicional de protección, mitigando el riesgo de que una cuenta sea comprometida por alguien que haya obtenido o adivinado la contraseña.

Hay diferentes maneras de implementar este método como aplicaciones de autenticación móvil, SMS o tokens de seguridad físicos, pero en la implementación se ha optado por el uso de un código de verificación que llega al usuario por correo electrónico (Figura 19) aprovechando que en el registro es obligatorio introducir uno. Esto obliga al atacante a conocer el correo del usuario y a poder acceder a este para iniciar sesión.

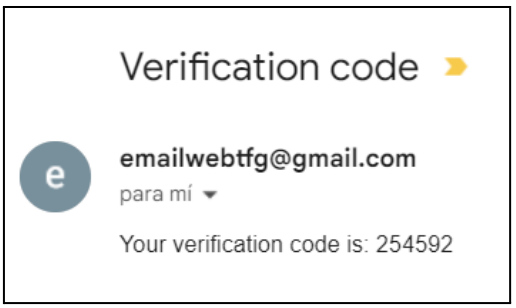


Figura 19 *Ejemplo de código recibido por correo*

Se ha optado por este sistema, ya que los servicios de correo usados habitualmente, como por ejemplo Gmail, están respaldados por empresas muy grandes, que a su vez aplican sus sistemas de seguridad en estos correos. Además, el uso del correo evita que el usuario tenga que descargarse o poseer una herramienta adicional y es un sistema gratuito.

Para generar el código (Figura 20) se decide usar *pyotp* [19], una biblioteca usada para generar y verificar códigos de un solo uso pensados para doble autenticación. Con esta biblioteca se crea un código aleatorio que será guardado en la base de datos, y con ese código se genera otro que será mandado al correo y será válido durante el tiempo indicado. De este modo el código enviado al usuario y el guardado son dos diferentes.

```
def generar_codigo(self, t):
    codigo_verificacion = pyotp.random_base32()
    self.code = codigo_verificacion
    self.save_to_db()
    totp = pyotp.TOTP(codigo_verificacion, interval=t)
    codigo = totp.now()
    return codigo
```

Figura 20 *Generación del código para el doble factor.*

Finalmente, en el registro de fallos (explicado más adelante) se incluyen los intentos de inicio de sesión y la limitación de acceso, que también ayuda a identificar intentos de ataques y así actuar en consecuencia.

Guardar contraseñas

Al almacenar contraseñas [20] en su forma original, existe el riesgo de que si un atacante obtiene acceso no autorizado a la base de datos, pueda leerlas y utilizarlas para acceder a cuentas de usuario. Para evitar eso, se debe aplicar un método que las mantenga ocultas. Tenemos la opción de cifrar y la de codificar, pero el cifrado permite

recuperar el texto original y, por tanto, añade una fuente de inseguridad al permitir a un atacante que accediera a la base de datos y también a la manera de descifrar (obteniendo la llave) obtener la contraseña. Por ese motivo se usa hash que no permite la recuperación del texto, siendo la única manera para un atacante comparar *hashes* creados sabiendo la cadena original.

Para dificultar más esto, se puede aplicar el uso de sal (haciendo que la misma cadena original no tenga el mismo hash) y un número configurable de iteraciones (aumentando el costo computacional del hash).

Al comprobar el hash guardado con la contraseña, el método usado debe incluir una comprobación de longitud de entrada máxima, para protegerse contra ataques de denegación de servicio con entradas muy largas, establecer explícitamente el tipo de ambas variables, para proteger contra ataques de confusión de tipos y regresar en tiempo constante, para protegerse contra ataques de tiempo.

Así, para implementar este hash (Figura 21) se usa la biblioteca *passlib* [21], que incluye tanto iteraciones como sal aleatoria al crear y al comparar aplica 2 de los 3 requisitos, asegura que la contraseña y el hash sean unicode o bytes y regresan en tiempo constante usando un hash ficticio, para simular la verificación de una contraseña de cuenta real.

```
def hash_password(self, password):
    self.password = pwd_context.hash(password)

def verify_password(self, password):
    return pwd_context.verify(password, self.password)
```

Figura 21 *Métodos para generar hash y verificar contraseña con passlib*

En cuanto a la longitud, se realiza la comprobación mediante validación de entradas.

Contraseña olvidada

En caso de olvidar la contraseña, también se debe implementar un método seguro para poder restablecerla [22], para hacerlo se usa el mismo sistema que al hacer la doble autenticación, necesitando el código recibido para validar el cambio de contraseña. También se ha limitado el número de intentos de recuperar la contraseña para evitar que un atacante inunde el sistema en el cual el usuario recibirá el código de recuperación (en este caso el email).

Respuestas de autenticación

En caso de producirse un error al crear la cuenta, iniciar sesión, o restablecer contraseña, la mejor opción según OWASP es responder con un mensaje único para diferentes errores o conflictos como ID de usuario o contraseña incorrecta, la cuenta no existe o ya existe… Aunque en términos de usabilidad se pierde calidad al no informar al usuario sobre el problema exacto que ocurrió en cuanto a seguridad, estos mensajes pueden proporcionar información valiosa a un atacante, permitiéndole, por ejemplo, saber que cuentas existen o no.

La decisión de implementar mensajes de error genéricos o específicos depende del contexto y los requisitos de seguridad de la aplicación. Es importante evaluar los riesgos y equilibrar la seguridad con la usabilidad y la experiencia del usuario., En el caso de la web desarrollada se ha decidido implementar la recomendación de OWASP priorizando la seguridad.

1.3 Denegación de servicio

Otro posible vector de ataque relacionado con la identificación y autenticación es la denegación de servicio, esto se puede dar, por ejemplo, si un sistema permite un alto número de intentos de inicio de sesión fallidos. Es un ataque diseñado para bloquear el acceso a un sistema, abrumándolo con una gran cantidad de solicitudes o tráfico malicioso. Para intentar evitarlo se implementan diferentes recomendaciones como el uso del limiter antes mencionado o la realización de una validación básica en el lado del cliente (Figura 22) para disminuir el número de peticiones al servidor o para reducir el tamaño de los datos guardados (como en el tamaño de la contraseña) [23].

```
if (!this.username || !this.password) {
  this.error = 'Please complete all fields.'
  return
}
```

Figura 22 *Comprobación de campos en el cliente*

2- Control de acceso roto

Esta vulnerabilidad está muy relacionada con la autenticación y la autorización y se refiere a una debilidad en el diseño o la implementación de un sistema de control de acceso [24], que permite a los usuarios no autorizados obtener acceso a recursos o

funcionalidades restringidas. Las recomendaciones de OWASP [25] implementadas para evitar o reducir esta vulnerabilidad son las siguientes.

2.1 Recomendaciones OWASP

• Salvo recursos públicos, denegar por defecto.

Implica que cualquier solicitud de acceso a un recurso o funcionalidad debe ser denegada a menos que se verifiquen explícitamente los permisos y credenciales adecuados. Es decir, el acceso no se concede automáticamente.

Para implementarlo, en toda petición se comprueba si el recurso es público y en caso de que no lo sea, se comprueba si el usuario tiene acceso a este, de modo que los permisos de un usuario se definen en la base de datos. Con esto, en caso de crear un recurso nuevo, ningún usuario tendría permisos para acceder al recurso hasta que se le otorgue específicamente permiso, es decir, se deniega el acceso por defecto.

• Implemente mecanismos de control de acceso una vez y reutilícelos

Mediante un decorador *@require_access* se aplican las comprobaciones antes mencionadas, usando un único mecanismo que se reutiliza en todas las solicitudes, ya que se llama antes de cada una gracias a usar el decorador *@app_before_request* (Figura 23) de Flask.

```
@app.before_request
@require_access
def acces_control():
    pass
```

Figura 23 *Llamada al decorador @app_before_request y @require_access*

• Los controles de acceso del modelo deben imponer la propiedad de los registros

Es decir, el control de acceso debe ser implementado de manera que los usuarios solo puedan interactuar con los registros o datos sobre los cuales tienen propiedad o permisos específicos en lugar de permitir que cualquier usuario pueda realizar cualquier operación en cualquier registro.

Un ejemplo donde no se imponga la propiedad podría ser uno donde cada usuario tiene su dinero y este en un principio solo debe ser accesible para el mismo, pero en la implementación no se valida que el usuario que hace la petición sea el mismo que el del recurso que se quiere obtener, de manera, que simplemente devuelve el dinero del

usuario que le llega (Figura 24). Con esto, cualquier usuario autenticado podría cambiar el valor del parámetro *user* y obtener un registro que no debería ser accesible para él.

```
def get(self,user):
    if(user):
        money =AccountsModel.get_money(user)
        return jsonify({'money': money})
```

Figura 24 *Método sin comprobación de propiedad*

Para solucionar esto lo que se hace es usar el ID de usuario guardado en el token, para saber quien hace la petición (Figura 25) y es a partir de este usuario que se controla el acceso, permitiendo solo acceder a funciones y registros a los que este tiene permiso. Con el token, un atacante no pueda modificar el usuario que llega al servidor, a no ser que se vulnere el token firmado.

```
account = verify_auth_token(token,ctx,endpoint=='email')
if account is not None:
    g.user = account
```

Figura 25 *Obtener usuario del token*

• Prefiera el control de acceso basado en atributos y relaciones sobre RBAC

RBAC se basa en la asignación de roles predefinidos a los usuarios y la autorización de esos roles para acceder a ciertos recursos. Inicialmente, se usaba la definición de dos roles, *admin* y *user* que era una implementación de RBAC.

Finalmente, se ha determinado que cada usuario tenga una lista en la base de datos de los recursos o funciones a los que tiene acceso (Figura 26), lo que permite definir permisos más específicos e implementar de una mejor manera la denegación por defecto.

```
class Function(db.Model):
    __tablename__ = 'functions'
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), unique=True)
```

Figura 26 *Tabla de funciones*

• Verifique que las verificaciones de autorización se realicen en la ubicación correcta

Realizar la verificación únicamente en el cliente, puede ser evitado por un atacante, por eso se debe hacer desde el servidor. Como se ha venido explicando, la comprobación se realiza al recibir el token en el servidor, desde el cliente.

• Salga de forma segura cuando fallan las comprobaciones de autorización¶

Cuando el servidor rechaza la verificación del token, redirige al cliente a la página inicial (login) y además se registra en un *log*. Para implementarlo, al detectar un error en la verificación se usa el método *redirect* de Flask (Figura 27) en el servidor o se redirige desde el cliente mediante un interceptor (Figura 28) según el caso.

```
return redirect('/')
```

Figura 27 *Redireccionar a página inicial desde el servidor*

```
//Captura todas las respuestas
axios.interceptors.response.use(
  response => {
    return response;
  },
  error => {
    //Si la respuesta indica que ha fallado el token, lo elimina y redirige a la página inicial
    if (error.response.status === 401 && error.response.data.error === 'Token ha fallado') {
      sessionStorage.removeItem('token')
      console.log('No token')
        app.config.globalProperties.$router.push('/');    }
    return Promise.reject(error);
  }
```

Figura 28 *Redireccionar a página inicial desde el cliente*

2.2 Cross-Site Request Forgery Prevention (CSRF)

Es un tipo de vulnerabilidad [26] que permite a un atacante engañar a un usuario autenticado para que realice acciones no deseadas en una aplicación en la que el usuario tiene permisos. Para esto, el atacante aprovecha que la autenticación se hace a través de mecanismos que el usuario no ve, como el token y que en caso de estar en una cookie se envía automáticamente, engañando al usuario para que su navegador realicé una petición maliciosa que será autorizada gracias a la cookie guardada en el este.

Para engañar al usuario, un atacante puede realizar una web (Figura 29) que parezca de confianza, pero que al ser visitada internamente haga una petición HTTP

(incluyendo la cookie) a la otra aplicación , normalmente para acciones comprometidas como cambiar la contraseña.

```
<body>
    <h1>website</h1>

<script>
const url = 'http://127.0.0.1:8200/money';

fetch(url, {
  method: 'GET',
    credentials: 'include',
  headers: {
    'Content-Type': 'application/json',
  },
})
```

Figura 29 *HTML con petición HTTP oculta*

Para evitar este tipo de ataques se han implementado las siguientes defensas [27].

Contexto en el token

Así, en la implementación de la autorización ya se incluye la protección contra CSRF (sigue un esquema similar a una cookie de envío doble), esto es debido a que aunque la cadena aleatoria está guardada en las cookies, el token está en el *Session Storage*, de modo que este no se envía de forma automática.

CSRF Token

Es un tipo de token muy usado y su funcionamiento se basa en generar (para cada sesión o cada petición) una cadena aleatoria e impredecible que debe ser incluida en cada petición del cliente al servidor y en caso de no estar disponible o no ser válida la petición se rechaza

Con el contexto, no es del todo necesario implementarlo, pero finalmente se ha incluido para demostrar como se implementa y tener protección CSRF antes de realizar el login, para hacerlo, se usa la biblioteca *flask_wtf* como se muestra en la figura 30, que se encarga de la generación y la validación del token.

```
from flask_wtf.csrf import  CSRFProtect
csrf = CSRFProtect(app)
```

Figura 30 *Implementación del token CSRF con flask_wtf*

Una vez generado el token, debe ser almacenado en el cliente, esto se suele hacer como un valor oculto en un formulario, aunque también se puede incluir en la etiqueta <meta> que es lo que finalmente se ha implementado (Figura 31).

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

Figura 31 *Token CSRF en etiqueta meta de un HTML*

Uso del atributo *SameSite*

Este atributo para las cookies ayuda a prevenir CSRF, tal como se ha mencionado en la definición de una cookie segura. Se puede definir como *Lax*, *Strict* o *None* y determina cuándo las cookies se incluyen en las solicitudes. En el proyecto se ha elegido *Strict*, ya que es el más seguro, al indicar que la cookie solo se enviará si la solicitud proviene del mismo sitio web que la generó.

La barrera que aporta *SameSite* no es infranqueable y existen modos de evitarla como el uso de dominios hermanos vulnerables [28]. Por eso se debe usar únicamente como una capa adicional.

3- Fallos criptográficos

Se refiere a los problemas de seguridad relacionados con el uso inadecuado de algoritmos y mecanismos criptográficos en las aplicaciones web que pueden conducir a fallos en la confidencialidad, integridad y autenticidad de los datos [29].

3.1 Protección de datos en reposo: encriptación

En cuanto a los datos en reposo o almacenados, se debe intentar evitar el almacenamiento y únicamente guardar los datos que sean estrictamente necesarios, cuanto menos tiempo y menos datos se almacenen, menor será el riesgo de robo. Con los datos confidenciales que se deban guardar, se deben aplicar mecanismos de ocultación.

Encriptación de datos:

Se utiliza la encriptación para proteger los datos almacenados en reposo, de modo que convierte los datos en un formato ilegible para aquellos que no tengan la clave de desencriptación.

Para asegurar la eficacia de la encriptación hay 2 puntos muy importantes, el algoritmo escogido y la gestión de las claves:

Algoritmos

Existen muchos algoritmos que se pueden utilizar para cifrar los datos sensibles que se quieren ocultar, clasificados en 2 tipos que son asimétrico y simétrico. La recomendación es utilizar AES-GCM o AES-CCM en el caso del simétrico y ECC en el caso del asimétrico.

Las claves asimétricas son especialmente útiles cuando el cifrado y el descifrado son realizados por sistemas o entidades diferentes, debido a que se utiliza una clave pública para cifrar y una privada para descifrar. Si un mismo sistema hace las 2 tareas, el cifrado simétrico puede ser más eficiente y adecuado, ya que simplifica el proceso y no requiere la complejidad adicional que implica el uso de claves asimétricas.

Por estos motivos, en el proyecto, tal y como se ha mostrado en la Figura 7, se ha usado el algoritmo simétrico AES-GCM que es una combinación de dos algoritmos criptográficos, AES para la encriptación simétrica y GCM para la autenticación de mensajes y la integridad de los datos.

En cuanto a la aplicación en producción, un sistema como Key Vault de Azure, solo proporciona claves asimétricas, de manera que la clave privada solo estará disponible dentro del propio entorno Azure, mientras que la publica se puede descargar y compartir (Figura 32)

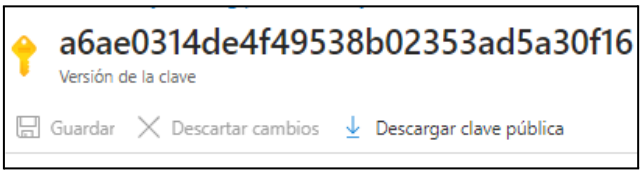


Figura 32 *Descarga de la clave pública en Azure Key Vault*

Gestión de claves

Las claves son un punto crítico de la seguridad criptográfica, esto es debido a que si un atacante consigue acceder a ellas o adivinarlas podrá convertir los datos en texto legible.

La gestión segura de claves incluye la generación segura, el almacenamiento seguro, el control de acceso, y la rotación periódica de las claves.

Generación

Para crear una clave de cifrado, token o ID de sesión se busca generar una cadena aleatoria de modo que sea imposible de adivinar, pero en general los ordenadores no pueden formar números totalmente aleatorios porque son deterministas, es decir, están diseñados para seguir instrucciones y realizar cálculos de manera predecible.

En Python habitualmente se usa la función *random()* pero no es segura, ya que su aleatoriedad es baja y no deben usarse para generar claves en términos de seguridad, por tanto, en el proyecto se ha decidido usar la biblioteca *secrets* [15] (Figura 33) tal como OWASP recomienda debido a que incluye características que lo hacen adecuado para su uso en criptografía.

```
nonce = secrets.token_bytes(12)
```

Figura 33 *Creación de cadena aleatoria segura con secrets*

Almacenamiento

Se recomienda guardar las claves en mecanismos de almacenamiento seguro. Además, las claves que se usan para cifrar datos si es posible deben guardarse también cifradas. Para poder llevar a cabo esto, se necesitan dos claves, la de cifrado de datos (DEK), que se utiliza para cifrar los datos, y la clave de cifrado de clave (KEK), que se utiliza para cifrar el DEK. Las 2 claves deben almacenarse separadas para aportar esta seguridad.

En el proyecto en local todas las claves se guardan en las variables de entorno encriptadas con la KEK que luego se usa para desencriptarlas (Figura 34)

```
KEK = os.environ.get('KEK')
secret_key = descifrar_clave(os.environ.get('MI_SECRET_KEY'),KEK)
secret_key2 = descifrar_clave(os.environ.get('MI_SECRET_KEY2'),KEK)
```

Figura 34 *Desencriptación de variables de entorno con la KEK*

En cuanto a las claves en producción, la idea era guardar la KEK en el Key Vault que proporciona Azure, que es un mecanismo de almacenamiento seguro pensado específicamente para esto, pero finalmente se ha acabado guardando la KEK junto a las DEK en “Configuración de aplicaciones” (Figura 35) que se pasan como variables de entorno al código de la aplicación, se cifran en reposo y se transmiten a través de

un canal cifrado. Esto ha sido así, ya que no se ha conseguido acceder a los valores guardados en Key Vault (Figura 36).

Figura 35 *Claves guardadas en configuración de aplicaciones*

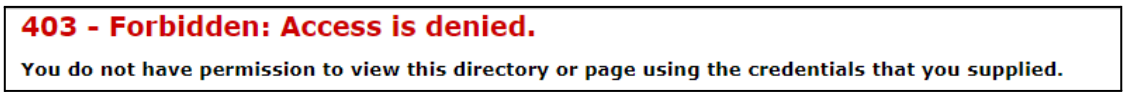


Figura 36 *Error al acceder al Key Vault*

Rotación

La rotación de claves es un proceso que consiste en generar nuevas claves y reemplazar las claves antiguas para mejorar la seguridad. Para implementarlo, Azure Key Vault incluye un sistema de rotación de claves (Figura 37).

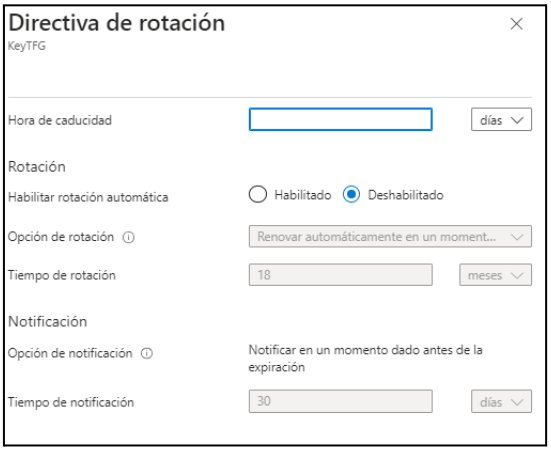


Figura 37 *Rotación de claves en Azure*

3.2 Protección de datos en tránsito: HTTPS

En el desarrollo web, típicamente se usa HTTP, que es un protocolo de comunicación utilizado para transferir información en la web, el cual no proporciona una capa de seguridad por sí mismo, lo que significa que los datos transmitidos a través de este no están cifrados, con lo que pueden ser interceptados y leídos por terceros. Por este motivo aparece HTTPS, que es una variante segura de HTTP que utiliza SSL o TLS para agregar una capa de cifrado a las comunicaciones, protegiendo los datos de posibles ataques o interceptaciones, siendo TLS y SSL protocolos criptográficos donde SSL fue el protocolo original y sus versiones posteriores pasaron a llamarse TLS [30].

En el proyecto, para poder probar la conexión HTTPS en local, se ha usado un certificado autofirmado, generado a través de OpenSSL (Figura 38).

```
openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365 -out certificate.pem
```

Figura 38 *Generar certificado con openssl*

Con la instrucción anterior, se generan dos archivos, el *cert.pem* y el *key.pem*, que son añadidos al proyecto para ejecutar en HTTPS (Figura 39). Este tipo de certificados son identificados como no seguros (Figura 40) debido a que no se ha aprobado por una entidad de confianza, pero sirven para realizar pruebas.

```
app.run(host='127.0.0.1', port=5000, ssl_context=('cert.pem', 'key.pem'))
```

Figura 39 *Ejecutar Flask con HTTPS*

En producción, Azure proporciona un certificado firmado como se muestra en la Figura 40.

Figura 40 *Certificado de Azure y certificado autofirmado en el navegador*

Además, existen herramientas que permiten generar certificados firmados con una duración limitada, como es el caso de ZeroSSL (Figura 41) que obliga a certificar la posesión del dominio para obtenerlo.

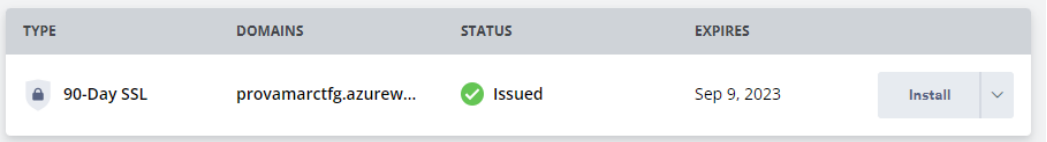


Figura 41 *Certificado generado con ZeroSSL*

En cuanto al protocolo usado, la recomendación es usar TLS 1.3 de manera predeterminada, añadir compatibilidad con 1.2 y deshabilitar los demás. En Azure se puede definir la versión mínima de TLS usada, pero no tiene la versión 1.3 (Figura 42)

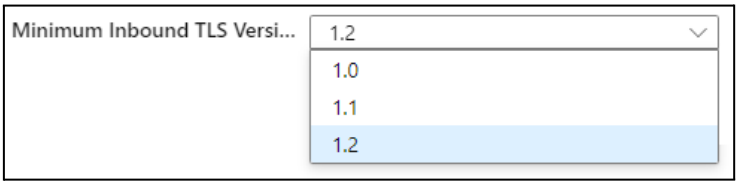


Figura 42 *Opción para definir versión mínima de TLS en Azure*

Eligiendo 1.2 como versión mínima, y analizando la página de Azure con una herramienta como *https://www.ssllabs.com/ssltest/* que examina la configuración del servidor SSL, devuelve el resultado de la Figura 43.

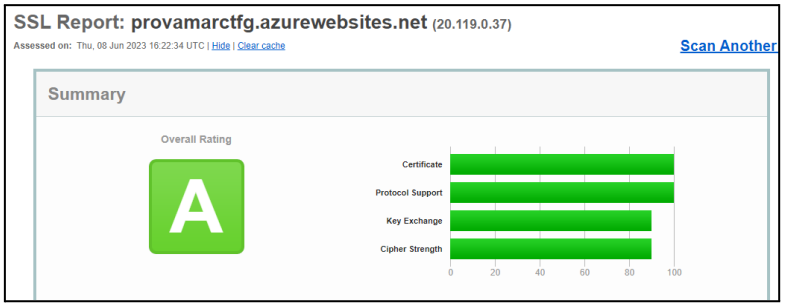


Figura 43 *Análisis SSL*

Fijándonos en el informe (Figura 44) vemos que no usamos TLS 1.3, que sería la mejor opción.

TLS 1.3	No
TLS 1.2	Yes

Figura 44 *Protocolos TLS de la web*

Comparación entre HTTP y HTTPS

Utilizando un programa llamado *WireShark* [31] para examinar el tráfico de red en tiempo real, se ha podido comprobar que al usar una conexión HTTP e interceptar la comunicación se pueden ver los paquetes HTTP (Figura 45) y su contenido (Figura 46)

```
HTTP/JSON          1202 POST /login HTTP/1.1 , JavaScript Object Notation (application/json)
```

Figura 45 *Paquete con protocolo HTTP/JSON interceptado con WireShark*

```
JavaScript Object Notation: application/json
  Object
    Member: username
    Member: password
```

Figura 46 *Contenido de un paquete HTTP*

En cambio, con HTTPS solo se ven paquetes con protocolo TCP que no muestran el contenido, aportando así la seguridad contra intercepción de datos.

HSTS

OWASP recomienda aplicar HSTS [32], ya que este proporciona protección contra ataques que podrían intentar forzar al usuario a utilizar una conexión HTTP, indicando a los navegadores que solo se deben comunicar con HTTPS. Para aplicarlo en Flask se puede añadir la cabecera mostrada en la figura 47.

```
response.headers['Strict-Transport-Security'] = 'max-age=31536000; includeSubDomains'
```

Figura 47 *Cabecera HSTS en Python*

También existe una biblioteca, que permite redirigir todas las peticiones HTTP a HTTPS, llamada SSlify (Figura 48)

```
sslify = SSLify(app)
```

Figura 48 *Llamada a SSlify en Python*

En cuanto a producción, Azure permite activar una opción para que todas las conexiones sean HTTPS (Figura 49).

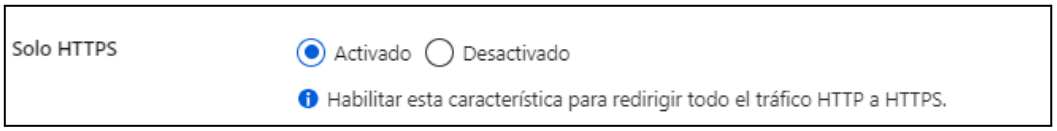


Figura 49 *Opción de solo HTTPS en Azure*

4- Inyección

La inyección se refiere a una vulnerabilidad de seguridad en la que un atacante puede insertar código malicioso en una aplicación o sistema, engañándola para que ejecute ese código no deseado. Esto ocurre cuando una aplicación no valida o filtra correctamente los datos de entrada del usuario antes de enviarlos a un intérprete [33].

4.1 SQL

La inyección más común es la de SQL. Con este tipo de inyección, un atacante busca insertar instrucciones SQL maliciosas dentro de una consulta legítima, lo que puede permitirle ver, modificar o eliminar datos en una base de datos. Existen 3 tipos de inyección SQL, *inband* (un único canal), *out-of-band* (un canal para inyectar y otro para extraer) e *inferential* (no puede ver los resultados, solo información sensible) [34].

Causas de la inyección

Este tipo de inyección se puede producir básicamente por 2 motivos, el primero es usar consultas dinámicas con concatenación de cadenas y el segundo es no validar la entrada del usuario que se usará en las consultas, permitiendo que valores maliciosos afecten a esta.

Las consultas dinámicas son consultas SQL que se generan y modifican en tiempo de ejecución, por otro lado, la concatenación de cadenas implica la combinación de cadenas de texto utilizando un operador para juntarlas.

Un ejemplo en el que se produce inyección SQL es el que vemos en la Figura 50, donde usa la concatenación al insertar el parámetro *username* mediante el operador *+.*

```
query = db.text("SELECT available_money FROM accounts WHERE username = '" + username  + "'")
```

Figura 50 *Consulta dinámica con concatenación de cadenas en Python*

Si además de esto no hay validaciones al crear un usuario, un atacante podría crear un usuario con el siguiente nombre *'or username='marc' –,* que al ser incluido en la consulta (concatenando) haría que el sistema lo interpretara como la instrucción de la Figura 51 devolviendo el registro del usuario *marc* al que el atacante no debería tener acceso.

```
"SELECT available_money FROM accounts WHERE username = '' or 'marc "
```

Figura 51 *Consulta modificada con el nombre de usuario 'or username='marc' –*

Soluciones

Para evitar esto hay diferentes estrategias, como escapar el parámetro contra inyección SQL (menos seguro) o usar consultas parametrizadas (Figura 52), que aseguran que el sistema distinga entre código y datos, independientemente de la entrada del usuario.

```
query = db.text("SELECT available_money FROM accounts WHERE username = :username")
result = connection.execute(query, {'username': username}).fetchone()
```

Figura 52 *Consulta parametrizada en Python*

Finalmente, la opción aplicada ha sido usar las funciones que incluye el ORM SQLAlchemy. Este se encarga de manera interna de generar y ejecutar consultas SQL de forma segura. En este caso se realiza mediante *filter_by()* que devuelve el usuario de la base de datos con el nombre indicado (Figura 53) y luego se obtiene el dinero.

```
account = cls.query.filter_by(username=username).first()
```

Figura 53 *Uso de filter_by de SQLAlchemy*

4.2 XSS

Otra inyección muy común en aplicaciones web es Cross-site *scripting* (XSS) [35], la cual permite a los atacantes insertar código malicioso, generalmente en forma de scripts de JavaScript que se ejecutan en el navegador web del usuario. Estos scripts permiten al atacante robar información confidencial, como credenciales de inicio de sesión, realizar acciones en nombre del usuario o incluso redirigir al usuario a sitios web maliciosos.

Existen 3 tipos de ataque XSS que son el reflejado, el persistente y el de DOM. El primero ocurre cuando una web devuelve inmediatamente la entrada del usuario o parte de esta, el segundo cuando se almacena en el servidor y el último cuando el script malicioso manipula directamente el DOM sin pasar por el servidor.

Solución

El vector de este tipo de ataques suele ser la entrada de datos por parte del usuario, por lo tanto, debemos tomar medidas estrictas en los casos donde permitimos a un usuario introducir datos que posteriormente se muestren en la web.

La primera acción que debemos tomar para reducir este tipo de ataques es el uso de un framework moderno, ya que proporciona herramientas para facilitar la seguridad mediante el uso de plantillas, escape automático y otros. En este caso se usa Vue, así que debemos ver cómo actúa y tomar medidas.

Vue tiene tres formas principales de mostrar contenido dinámico en las plantillas, v-text, interpolación doble {{}} y v-html, donde las 2 primeras incluyen escapado automático para evitar la interpretación de caracteres especiales mostrando los datos como texto plano, pero la tercera permite renderizar contenido HTML, por lo tanto, es la más vulnerable contra XSS [36].

Un ejemplo donde se produciría XSS es al añadir un comentario sin filtrar en la página, donde un atacante podría decidir escribir el texto mostrado en la Figura 54 [37].

```
<img style="display:none" src="https://hacker-website.com"
onerror="console.log(sessionStorage.getItem('token'))">
```

Figura 54 *Comentario malicioso para inyección XSS*

Con esto, al cargarse mediante v-html, mostraría el token por consola (Figura 55) y además intentaría hacer un GET a la web indicada en el src (Figura 56).

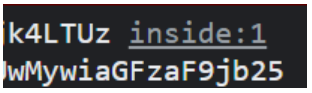


Figura 55 *Mensaje por consola mostrando un token (Censurado)*

Figura 56 *Mensaje de error al hacer un GET a https://hacker-website.com*

Con el GET, un atacante podría hacer que cualquier usuario que cargara el comentario con su enlace, enviara una solicitud a su sitio web, permitiéndole por ejemplo robar credenciales, cookies o enviar un script malicioso (aunque hay técnicas implementadas en el proyecto que deberían evitarlo como las *flags* de la cookie).

En cambio, con v-text y {{}} el valor interno en el navegador, no es el texto introducido, sino que se escapan los símbolos < y > como *< y >* evitando que se ejecute.

El problema es que la documentación de Vue [38] indica que *Este escape se realiza mediante API nativas del navegador, como setAttribute, por lo que una vulnerabilidad solo puede existir si el navegador en sí es vulnerable.* Teniendo esto en cuenta, si nos quedamos con la solución de Vue, nuestra seguridad dependería del navegador. Por este motivo, la solución es implementar la seguridad dentro del propio sistema.

La primera idea era mantener el uso de {{}} para escapar en el cliente y añadir un escapado en el servidor, pero con esto se produciría un doble escapado, generando el resultado de la Figura 57.

Figura 57 *Comentario con doble escapado*

Para evitar esto finalmente solo se implementa en el servidor, porque además el cliente es más propenso a ataques, mientras que el servidor asegura que se aplique de manera consistente en diferentes clientes.

El escapado usado es de la biblioteca *markupsafe* [39] (Figura 58) y sustituye los caracteres peligrosos por sus entidades HTML.

```python
from markupsafe import escape
txt = escape(data["text"])
```

Figura 58 *Uso de escape de markupsafe en Python*

La parte negativa de escapar el HTML, es que convierte totalmente a texto plano. En caso de querer permitir algún tipo de etiqueta HTML, se puede utilizar la sanitización, que implica validar y filtrar el contenido para permitir solo ciertos elementos, en lugar del escapado.

En el proyecto se ha añadido sanitización en el lado del cliente, ya que así no hay doble escapado y es una capa de seguridad adicional. Para hacerlo se ha usado *DomPurify* [40] (Figura 59).

```python
return DOMPurify.sanitize(input, {ALLOWED_ATTR: []})
```

Figura 59 *Uso de sanitize de DomPurify*

Otra defensa implementada ha sido CSP [41], que permite a los administradores de un sitio web especificar una serie de directivas que controlan qué tipos de contenido y de qué orígenes se pueden cargar y ejecutar.

Para implementarlo hay diferentes formas y en el proyecte se hace uso del header *Content-Security-Policy*, donde se define que por defecto (default-src) solo se permite cargar recursos desde el mismo origen del sitio web (self) y en cuanto a las imágenes no se permite desde ningún sitio (Figura 60).

```python
response.headers['Content-Security-Policy'] = "default-src 'self'; img-src 'none'"
```

Figura 60 *Cabecera para aplicar CSP en Python*

Al aplicar CSP y añadir el comentario malicioso de la Figura 54, se puede ver que bloquea la ejecución (Figura 61), aun sin aplicar las otras defensas.

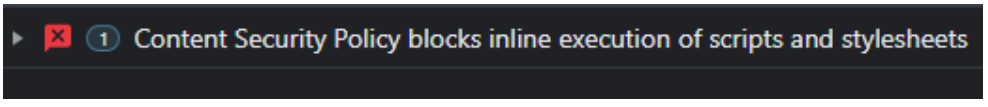


Figura 61 *Mensaje del navegador informando el bloqueo de un script mediante CSP*

Gracias a este defensa, se cambió la versión de Vue usada previamente y que era vulnerable, porque internamente ejecutaba la función insegura *Eval* detectada por CSP.

4.3 Validación de entrada

La validación de entrada se encarga de garantizar que los datos que se reciben sean correctos y estén correctamente formados, evitando que datos incorrectos o maliciosos se propaguen a través del sistema y causen errores [42].

Cuando se realiza la validación de entrada, se verifican varios aspectos de los datos, como su formato, tipo, tamaño y cualquier restricción específica que deba cumplirse. Aunque no debe usarse como el método principal para prevenir XSS o inyección SQL, puede ayudar a reducir o limitar a estos.

Implementación de validaciones de entrada

Se han realizado diversas validaciones de entrada durante el flujo de datos.

En los POST se añade explícitamente el tipo de cada uno de los valores recibidos (Figura 62), de modo que en caso de no ser del tipo indicado lanza una excepción.

```
parser.add_argument('username', type=str, required=True, help="This field cannot be left blanck")
```

Figura 62 *Inserción del argumento username al parser de Flask-RESTful*

En los modelos de la base de datos se hace una comprobación de tipo, y en algunos casos también de longitud (Figura 63). En caso de no coincidir con lo requerido no añadirá la entrada.

```
id = db.Column('id', db.String(length=36), default=lambda: str(uuid.uuid4()), primary_key=True)
```

Figura 63 *Definición de columna id con SQLAlchemy*

La validación de longitud también se aplica en otras partes del servidor y en el cliente en casos como el texto de los comentarios o la contraseña (Figura 64).

```
if (this.password.length < 8)
```

Figura 64 *Comparación la longitud de password en Python*

También se usan expresiones regulares para validar el formato de parámetros como el nombre de usuario o correo (Figura 65).

```python
valid_username = re.match(r'^\w+$', user)
```

Figura 65 *Validación de formato de user* en Python

En cuanto a la validación de los archivos que se permite introducir, se verifica la extensión (Figura 66) y el tamaño máximo (Figura 67).

```
<input type="file" accept=".xml,.json,.pickle" @change="cargarArchivo" />
```

Figura 66 *Campo de input en Vue*

```python
if(size > 1024 * 1024)
```

Figura 67 *Comparación de tamaño en Python*

5- Diseño inseguro

Esta vulnerabilidad se centra en los riesgos relacionados con fallos en el diseño y para evitarlo se debe seguir una metodología de diseño seguro que realiza evaluaciones periódicas de las amenazas y se asegura de que el código esté diseñado y probado de forma sólida para evitar vulnerabilidades conocidas [43].

En el proyecto, teniendo en cuenta el objetivo y contenido, se ha seguido en todo el desarrollo un enfoque basado en la seguridad, de modo que la identificación de amenazas y las posibles soluciones se encuentran reflejadas en todo el proceso.

6- Mala configuración de seguridad

Como su nombre indica, esta vulnerabilidad se produce cuando la configuración de seguridad de una aplicación web o de su infraestructura no se implementa correctamente. Para evitarla, OWASP propone usar la infraestructura como código [44]. Es un enfoque que permite administrar la infraestructura, mediante el uso de archivos de configuración y scripts de programación en lugar de configuraciones manuales tradicionales para la gestión de la infraestructura en la nube.

En el caso de Azure tiene una función que permite generar plantillas (Figura 68) que son una forma de implementar la infraestructura como código.

```
"resources": [
    {
        "type": "Microsoft.Web/sites",
        "apiVersion": "2022-09-01",
        "name": "provamarctfg",
        "location": "East US",
        "kind": "app,linux",
        "properties": {
```

Figura 68 *Parte de plantilla Azure*

Otra recomendación para evitar esta vulnerabilidad es el uso de complementos IDE que ayudan a detectar problemas de seguridad de forma rápida y automática. En el proyecto se ha usado PyCharm como IDE [45] y los complementos de seguridad aplicados han sido los siguientes:

Python Security [46]**,** se enfoca en la detección de vulnerabilidades en Python (Figura 69).

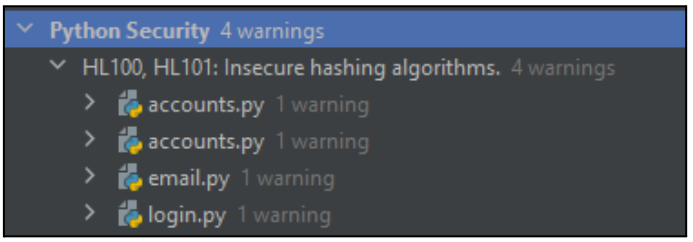


Figura 69 *Resultado del análisis con Python Security*

Al usar esta herramienta durante el desarrollo, se vio que el algoritmo de hash usado era vulnerable y, por tanto, se cambió.

Bandit [47], se enfoca en encontrar problemas de seguridad comunes en el código y se puede ejecutar con el comando *bandit -r nombre_carpeta -f html -o out.html*

Con esta herramienta se descubrió una vulnerabilidad en la generación de números pseudoaleatorios no seguro (Figura 70) y se cambió por uno criptográficamente fuerte.

blacklist: Standard pseudo-random generators are not suitable for security/cryptographic purposes.
Test ID: B311
Severity: LOW
Confidence: HIGH
CWE: CWE-330
File: .\tfg\resources\login.py
Line number: 86
More info: https://bandit.readthedocs.io/en/1.7.5/blacklists/blacklist_calls.html#b311-random

```
85                              caracteres = string.ascii_letters + string.digits
86                              contexto_usuario = ''.join(random.choices(caracteres, k=20))
87
```

Figura 70 *Resultado del análisis con Bandit*

6.1 XXE

En aplicaciones web que procesan XML aparece la vulnerabilidad XML eXternal Entity [48]. Esta vulnerabilidad está relacionada con el uso de entidades XML que son referencias a un objeto, siendo las externas, referencias a entidades de otro archivo que pueden ser usadas para distintos tipos de ataque como Billion Laughs o External Entity Expansion que son ataques de DoS o robo de información entre otros.

Por esto, la mejor opción si se quiere permitir introducir archivos XML es deshabilitar las entidades, aunque tiene como punto negativo que, ya que no podremos usarlas.

Implementación XXE

En el proyecto se ha añadido un apartado donde poder pasar archivos de tipo XML.

A continuación, se muestra un ejemplo de una vulnerabilidad y como se ha mitigado.

Lo primero que se hace es crear un archivo que simule datos privados (Figura 71).

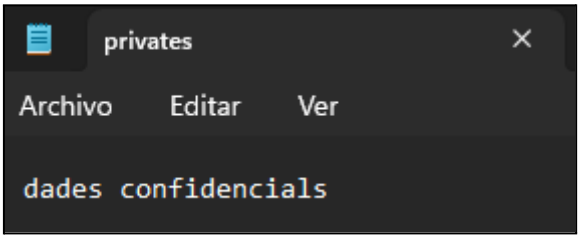


Figura 71 *Archivo privates.txt*

Seguidamente, se crea un XML (Figura 72) que explota la vulnerabilidad XXE utilizando una entidad externa para acceder al archivo de la Figura 71.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ENTITY xxe SYSTEM "file:///C:/privates.txt">
]>
<user>
  <username>&xxe;</username>
</user>
```

Figura 72 *Archivo XML con una entidad externa para ataque XXE*

Cuando se procese el XML de forma no segura, el contenido del archivo se pondrá en el lugar de la entidad *&xxe;* lo que expone el contenido del archivo en el resultado.

Usando la librería *lxml* [49] y su método *etree,* al imprimir por pantalla el resultado, vemos que se obtiene el texto *dades confidencials* incluido en el txt (Figura 73).

```
tree2 = etree.parse(archivo)
for element in tree2.iter():
    print(f"Elemento: {element.tag}")
    print(f"Contenido: {element.text}")
```

```
Elemento: user
Contenido:

---
Elemento: username
Contenido: dades confidencials
```

Figura 73 *Análisis de archivo XML con etree y contenido del XML.*

En la documentación de Python, instan a usar el paquete *defusedxml* [50] que está especializado en evitar vulnerabilidades XML. Para usar esta solución simplemente se debe importar la biblioteca y usarla en lugar de *lxml* (Figura 74).

```
tree3 = defusedxml.ElementTree.parse(archivo)
```

Figura 74 *Análisis de archivo XML con deusedxml.*

En este caso, con el archivo de la Figura 72, *deusedxml* lanza una excepción que evita la ejecución y nos permite, por ejemplo, registrar un intento de vulneración.

7- Componentes vulnerable y desactualizados

Los componentes vulnerables y obsoletos son aquellos software o librerías utilizados en una aplicación que contienen fallas de seguridad conocidas o que no reciben actualizaciones. Son un punto débil en la seguridad de la aplicación, porque los atacantes pueden aprovechar esas vulnerabilidades públicas para comprometer la aplicación y obtener acceso no autorizado, robar datos o realizar otros ataques.

7.1 Gestión de dependencias vulnerables

Para facilitar el trabajo de detección de vulnerabilidades en las dependencias de un proyecto, existen bases de datos donde se publican los fallos de las librerías y herramientas y servicios especializados que permitan monitorearlas [51] como OWASP Dependency Check, NPM Audit, Safety y Pip-audit que se exponen a continuación.

OWASP Dependency Check

Es una herramienta de código abierto que escanea las dependencias de un proyecto en busca de vulnerabilidades conocidas y genera informes detallados. Puede integrarse en el flujo de trabajo de desarrollo para realizar análisis automáticos y usa las fuentes de datos de NVD (National Vulnerability Database) alojadas por el NIST (National Institute of Standards and Technology) [52].

Con la herramienta descargada, se realizó un primer análisis del proyecto haciendo uso de la instrucción de la Figura 75.

```
dependency-check --project 'MiProyecto' --out directoriosalida --scan directorioproyecto
```

Figura 75 *Ejecución de dependency-check*

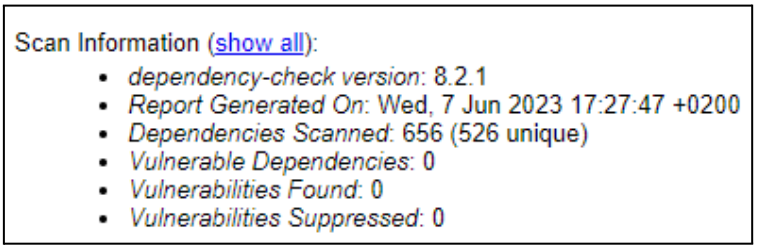
El resultado de la ejecución es un informe (Figura 76) detallado de las dependencias usando estándares como CPE o CVE.

Display: Showing Vulnerable Dependencies (click to show all)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
ajv:5.5.2	cpe:2.3:a:ajv.js:ajv:5.5.2:*:*:*:*:*:*:*	pkg:npm/ajv@5.5.2	MEDIUM	1	Highest	8
ansi-html:0.0.7	cpe:2.3:a:ansi-html_project:ansi-html:0.0.7:*:*:*:*:*:*:*	pkg:npm/ansi-html@0.0.7	HIGH	1	Highest	8
bootstrap.bundle.js		pkg:javascript/bootstrap@4.0.0	MEDIUM	5		3
bootstrap.bundle.min.js		pkg:javascript/bootstrap@4.0.0	MEDIUM	5		3
bootstrap.js		pkg:javascript/bootstrap@4.0.0	MEDIUM	5		3
bootstrap.min.js		pkg:javascript/bootstrap@4.0.0	MEDIUM	5		3
browserslist:1.7.7	cpe:2.3:a:browserslist_project:browserslist:1.7.7:*:*:*:*:*:*:*	pkg:npm/browserslist@1.7.7	MEDIUM	1	Highest	6
browserslist:2.11.3	cpe:2.3:a:browserslist_project:browserslist:2.11.3:*:*:*:*:*:*:*	pkg:npm/browserslist@2.11.3	MEDIUM	1	Highest	6
browserslist:3.2.8	cpe:2.3:a:browserslist_project:browserslist:3.2.8:*:*:*:*:*:*:*	pkg:npm/browserslist@3.2.8	MEDIUM	1	Highest	6

Figura 76 *Parte del informe inicial de vulnerabilidades generado con dependency-check*

El primer análisis realizado informó de una gran cantidad de vulnerabilidades debido al uso de una versión de Vue antigua. Al actualizar se eliminaron la mayoría de las dependencias vulnerables, quedando unas pocas que se fueron tratando individualmente, actualizando en caso de tener versiones sin errores o eliminándolas en caso de no tenerlas. Finalmente, con los cambios realizados el resultado obtenido fue de 0 vulnerabilidades encontradas (Figura 77).



Scan Information (show all):

- *dependency-check version*: 8.2.1
- *Report Generated On*: Wed, 7 Jun 2023 17:27:47 +0200
- *Dependencies Scanned*: 656 (526 unique)
- *Vulnerable Dependencies*: 0
- *Vulnerabilities Found*: 0
- *Vulnerabilities Suppressed*: 0

Figura 77 *Informe final de vulnerabilidades generado con dependency-check*

NPM Audit

NPM es el administrador de paquetes para la plataforma Node y con audit se pide un informe de vulnerabilidades conocidas usando NVD como base de datos. Inicialmente, el resultado de *npm audit* [53] fue el mostrado en la Figura 78 y después de realizar los cambios antes mencionados se obtuvieron 0 vulnerabilidades como se muestra en la Figura 79.

Figura 78 *Resultado inicial al ejecutar npm audit*

Figura 79 *Resultado final al ejecutar npm audit*

Las dos herramientas anteriores están centradas en las dependencias de la parte de Vue, así que también se han usado unas específicas para Python.

Safety

Es una herramienta de seguridad específica para proyectos de Python que verifica las dependencias usando su propia base de datos y en caso de encontrar vulnerabilidades, proporciona información de versiones no vulnerables [54].

Al aplicar esta herramienta inicialmente se obtuvieron vulnerabilidades como la de la librería *wheel* (Figura 80), pero al realizar las actualizaciones a la última versión se obtuvo el resultado de 0 vulnerabilidades como muestra a Figura 81.

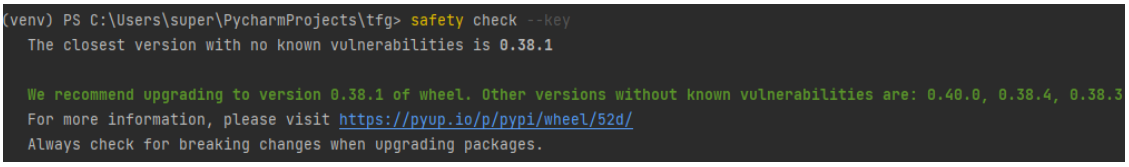


Figura 80 *Resultado de ejecutar safety*

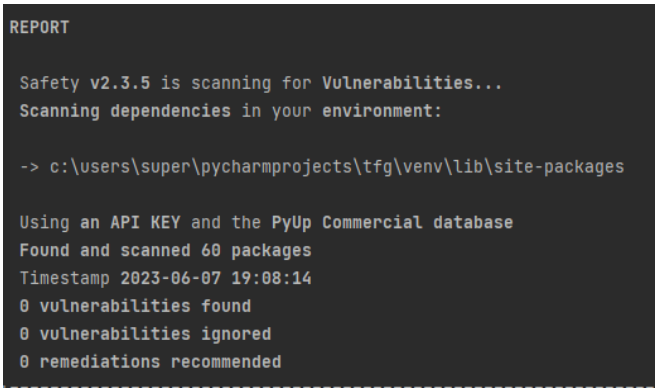


Figura 81 *Resultado final ejecutando safety*

Pip-audit

También es una herramienta específica de Python que escanea entornos de este lenguaje para detectar dependencias vulnerables, utilizando la base de datos Packaging Advisory Database y GHSA.

Como en el caso anterior, inicialmente encontraba alguna vulnerabilidad, pero, actualmente, no detecta ninguna (Figura 82) [55].

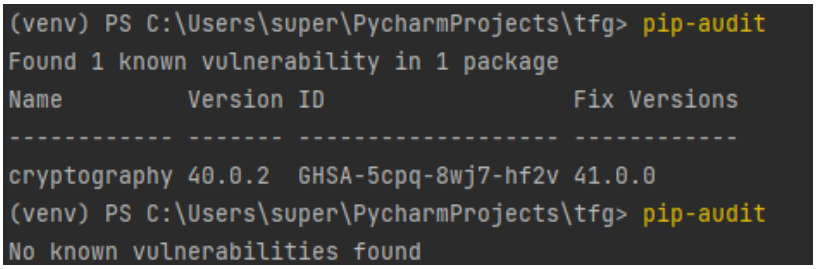


Figura 82 *Ejecución de pip-audit*

8- Fallas de integridad de software y datos

Esta vulnerabilidad está relacionada con realizar actualizaciones en el software o los datos, sin verificar si estos, han sido modificados, dañados o manipulados de manera malintencionada o no autorizada. Para evitar esto se debe confirmar que los datos y el software (incluyendo bibliotecas y dependencias) provengan de un origen de confianza, usar herramientas como OWASP Dependency Check ya mencionada anteriormente o revisar los cambios en el código [56]. Además, en esta vulnerabilidad se incluye la deserialización.

8.1 Serialización

La serialización es el proceso de convertir un objeto en una secuencia de bytes, que más adelante permite su restauración, mientras que la deserialización , como indica su

nombre, es el proceso contrario , permitiendo volver al objeto original. El formato en el que se guarda el archivo puede ser JSON, XML, YAML o incluso mecanismos de deserialización nativos en diferentes lenguajes, sin embargo, es importante tener precaución al trabajar con la serialización debido a los posibles riesgos de seguridad que implica.

La desrealización´puede llevar a diferentes tipos de ataques, como la ejecución remota de código (RCE) donde si un atacante puede controlar los datos deserializados, puede incluir código malicioso que se ejecutará en el servidor o en el sistema objetivo, provocando ataques DoS o ataques para evadir el control de acceso [57].

Implementación serialización

La mejor solución es bloquear la inserción de objetos serializados por los usuarios, aunque en el proyecto que se desarrolla se ha permitido esta inserción para analizar su problemática.

En el caso de Python, tiene su propio formato llamado *pickle* [58], pero se ha demostrado que tiene ciertas vulnerabilidades que pueden dar lugar a ataques RCE. Esto se produce porque al deserializar, pickle llama a la función reduce, de modo que si un atacante puede definir el objeto serializado, podría modificar este método para ejecutar la función que quiera. Si definimos una clase cualquiera, como Persona, incluimos un comando echo en el reduce (Figura 83) y serializamos un objeto de este tipo, vemos que al deserializar el objeto usando también pickle (Figura 84), el comando echo se ejecuta (Figura 85) y con esto, un atacante podría por ejemplo crear una Shell inversa que permite ejecutar diferentes comandos en el sistema atacado.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __reduce__(self):
        print('hola')
        cmd = ('echo "Comando ejecutado"')
        return os.system, (cmd,)
```

Figura 83 *Clase Persona con comando en reduce*

```
persona_recuperada = pickle.load(archivo)
```

Figura 84 *Deserializar con pickle*

```
127.0.0.1 - - [01/Jun/2023 17:50:21] "POST /sendxml HTTP/1.1" 200 -
"Comando ejecutado"
```

Figura 85 *Ejecución del comando echo al deserializar con pickle*

Para prevenir ataques de deserialización, en el proyecto, en lugar de usar *pickle* se ha usado JSON, que es un formato muy habitual y seguro que solo representa datos y no ejecuta código, lo que lo hace menos vulnerable a ataques, aunque tiene algunas desventajas como un soporte limitado para tipos de datos complejos en Python.

Para añadir más seguridad, se ha aplicado una validación, comprobando que contenga el esquema esperado. En este caso se trabaja con el de Persona, por lo tanto, debe ser un diccionario con los atributos y tipos de esta clase (Figura 86).

```
if isinstance(persona_recuperada, dict) and
    'nombre' in persona_recuperada and
    'edad' in persona_recuperada and
    len(persona_recuperada) == 2:
```

Figura 86 *Comprobación del esquema JSON*

9- Registro de seguridad y fallas de monitoreo

Esta vulnerabilidad incluye la falta o los errores en el registro y monitoreo de eventos en el sistema. Tratarlo es vital para la detección temprana de amenazas y la respuesta adecuada ante incidentes de seguridad.

Siguiendo los consejos de OWASP [59], se ha implementado un sistema de registros, donde se guardan las siguientes acciones:

- Problemas de validación:

Se registran los casos donde en el servidor llegan datos con un formato incorrecto que debería haber sido validado y bloqueado en el cliente, repitiendo la comprobación en el servidor, ya que un atacante podría saltarse la del cliente.

- Éxitos y errores de autenticación:

Se registra cuando un usuario inicia sesión correctamente, cuando falla o supera el límite de intentos, y cuando se actualiza (o se intenta actualizar) la contraseña que ha sido olvidada por el usuario.

- Problemas de autenticación

Por último, se registra cuando se intenta realizar una operación usando un token inválido, expirado o con algún problema en el contexto.

En cuanto el formato del registro, se sigue en todos los casos un esquema parecido al de la figura 87, basado en el vocabulario de registro de OWASP [60].

```
registro_exitoso = {
    "datetime": datetime.now().strftime('%Y-%m-%d %H:%M:%S,%f')[:-3],
    "event": "authn_login_success:" + username,
    "level": "INFO",
    "description": f"User {username} login successfully",
    "source_ip": ip,
    "request_url": url,
    "request_method": method,
    "host": host,
    "port": port,
    "user_agent": user_agent,
    "content_type": content_type
}
```

Figura 87 *Datos registrados al realizar un registro exitoso*

10- Falsificación de Solicitudes del Lado del Servidor (SSRF)

Esta vulnerabilidad permite a un atacante provocar que una aplicación realice solicitudes no autorizadas en su nombre desde el servidor en el que se encuentra alojada, permitiendo así enviar solicitudes desde el servidor hacia otros recursos de la red y dando acceso al atacante a recursos internos o externos a los que normalmente no tendría acceso, pero al hacer esas peticiones desde el propio servidor, se consigue la autorización [61].

Generalmente, ocurre cuando se permite a un atacante realizar solicitudes HTTP a un dominio arbitrario.

Realizar simulaciones de ataques SSRF en un entorno local puede ser complicado. Hay 2 tipos principales de ataque y en ambos aparecen limitaciones. El primer tipo es contra el **propio servidor**, usando una URL con un nombre de host como *127.0.0.1* o *localhost* que se da por ejemplo en casos donde el control de acceso se implementa de forma separada al servidor de aplicaciones, pero al ejecutar el proyecto en un entorno local con un único servidor, la seguridad ya está implementada en las llamadas a recursos 'internos'. El otro tipo es contra **otros sistemas *backend***, ubicados en la misma red interna y que solo sean accesibles desde la propia red, en

local también puede ser complicado de simular, ya que no se tienen sistemas internos a los que el *backend* tenga acceso, por lo que se ha usado el entorno en producción.

10.1 Implementación de SSRF

El primer paso para la vulnerabilidad es crear un punto de entrada, en el caso de la web implementada se ha creado un componente en Vue que es un botón que al ser pulsado envía una petición HTTP en la que se incluye como uno de los parámetros una URL, que será la fuente de la vulnerabilidad. También se ha añadido un input de texto en el que escribir la dirección, al recibir la URL, el servidor es quien envía un *GET* a esa URL (Figura 88), que crea la vulnerabilidad SSRF y podría ser explotada por parte del atacante.

```
url = data['stockApi']
inventory_response = requests.get(url)
inventory_data = inventory_response.json()
```

Figura 88 *GET a URL desde el servidor Python*

Con la aplicación en producción se han realizado diferentes pruebas, teniendo un entorno más completo con servicios internos.

La primera prueba fue realizar un *GET* a la propia página, lo que hizo que se quedara colgada, demostrando lo peligroso que puede ser dejar enviar URLs arbitrarias. Además, se probaron diferentes direcciones, entre ellas la propia IP virtual del App Service en Azure (Figura 89) y otras como por ejemplo, con 10.119.16.39.

Con la segunda dirección el error mostrado es *ErrorHTTPSConnectionPool* y con la IP virtual es *ErrorExpecting* (Figura 90). En el primer caso, puede deberse a que el servidor en esa dirección no está activo o no responde en ese momento, en cambio, en el segundo caso, el resultado puede deberse a que se está intentando acceder a una dirección IP que no devuelve una respuesta válida en formato JSON, es por ejemplo el mismo error que al intentar hacer un *GET* a https://www.google.com (que evidentemente existe).

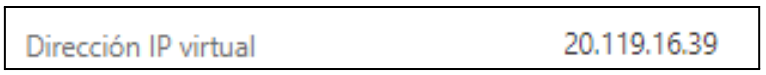


Figura 89 *Dirección IP virtual de App Service en Azure*

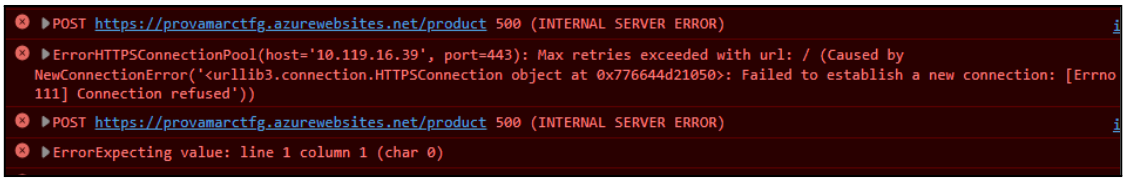


Figura 90 *Resultados al hacer GET interno a IP Virtual y 10.119.16.39*

Finalmente, sé probo con un GET a una página que devuelve un número tal y como espera el método stock, como es el caso de:

https://www.random.org/integers/?num=1&min=1&max=100&col=1&base=10&format=plain&rnd=new

Con esta la respuesta sí se recibe y se muestra correctamente, al no fallar al intentar acceder al JSON (mostrado en la figura 88). A la vista de estos resultados, en los que se muestran 2 errores distintos, la conclusión es que la petición a la IP privada sí se realiza y, por tanto, la vulnerabilidad estaría ocurriendo.

Para defenderse contra SSRF hay diferentes opciones, siendo la mejor y más segura modificar la lógica para evitar el paso de URLs siempre que sea posible, por ejemplo haciendo la llamada a la URL desde el cliente de forma directa.

Para los casos en los que la lógica implementada no se puede modificar y es necesario el uso de URL, aparecen también una serie de soluciones.

Una posible implementación sería una **lista negra**, que bloqueara rutas comprometidas, por ejemplo incluyendo *localhost* o *127.0.0.1* para evitar ataques SSRF contra el propio servidor u otras rutas que puedan generar vulnerabilidad. Esta solución tiene múltiples inconvenientes, como que la lista es limitada y, por tanto, solo puede incluir un subconjunto de direcciones o nombres de host maliciosos y en consecuencia otros podrían ser accesibles. Además, necesita revisión constante, ya que los atacantes pueden cambiar las direcciones IP o nombres de host utilizados para llevar a cabo ataques, por último existen técnicas para evitar la lista negra, usando diferentes definiciones de una misma IP, por ejemplo *127.0.0.1*, representada como *2130706433* (valor decimal equivalente de la dirección IP en notación de puntos.), debido a que en ciertos contextos aceptan diferentes formatos de direcciones IP por último también existen modos de codificación de URL y otras técnicas para engañar al sistema [62].

Una mejor solución es una **lista blanca**, que consiste simplemente en generar una lista que contenga las URLs permitidas (Figura 91), de modo que al recibir una dirección se comprueba si es una de esas y en caso de no serlo se rechaza.

```
def es_url_permitida(url):

    lista_blanca = [
        url1,url2
    ]
    return url in lista_blanca
```

Figura 91 *Comprobación de URL en lista blanca*

La forma más sólida de usarla es incluir en la lista blanca el nombre de host o la dirección IP de las aplicaciones a las que se necesita acceder. Lo malo de la lista blanca es que limita las URL usadas y hay casos en que la lógica pide que la aplicación pueda enviar solicitudes a cualquier dirección IP externa o nombre de dominio, en ese caso se suelen implementar listas negras, pero como se ha comentado estas son menos restrictivas que las blancas y tienen muchas posibilidades de ser vulnerables.

Por último, para aumentar la seguridad es recomendable habilitar la autenticación siempre que sea posible, incluso para los servicios en la red local.

En la implementación en producción se ha optado por una lista blanca que solo permite la llamada a la página antes mencionada para mostrar un número entero.

Conclusiones y trabajo futuro

Durante la realización del proyecto de desarrollo de una web segura y tal como consta en los objetivos que tenía planteados al inicio, se ha procedido al estudio de la guía de OWASP y siguiendo las medidas en ella recomendadas, se ha desarrollado una aplicación para conocer y mitigar los principales riesgos y vulnerabilidades presentes en webs y corrigiendo en la medida de lo que me ha sido posible esas vulnerabilidades.

Documentar e implementar las diferentes funcionalidades ha servido para obtener un conocimiento que antes no se tenía sobre la seguridad informática, viendo que la ciberseguridad es un ámbito imprescindible y a la vez muy complejo, debido a que existen muchas formas para vulnerar las webs y sus defensas, que pueden ser aprovechadas por ciberdelincuentes para provocar ataques, robar datos, suplantar identidades… llegando a la conclusión de que aun incorporando todas las defensas posibles, estas no son perfectas ni pueden serlo.

Así, el proyecto final, puede servir para comprender mejor como funcionan las diferentes defensas en un entorno común, entendiendo, por ejemplo, que la seguridad está basada en capas.

Respecto a los problemas surgidos, el principal impedimento ha sido la planificación y gestión del tiempo debido a que algunas de las implementaciones realizadas en el proyecto, han costado más de lo que tenía previsto. La puesta en producción, por ejemplo, ha costado mucho de implementar debido a la falta de experiencia previa con el uso de Azure.

En cuanto al trabajo futuro, como se ha dicho, no existe la defensa perfecta, por tanto, siempre es posible mejorar el resultado, en el caso de las claves, por ejemplo, sería recomendable usar Key Vault (se ha intentado) u otro servicio parecido, con lo cual una ampliación del proyecto podría ser la implementación de este servicio que en el presente no se ha conseguido. Además, existen muchos tipos de ataque, los cuales no han sido tratados durante el proyecto y podrían servir para ampliarlo, sobre todo si se tiene en cuenta que con el tiempo aparecen nuevas vulnerabilidades y técnicas de ataque, por lo que sería necesario mantenerse actualizado.

Referencias

[1] Riera, M. F. (2022). *Puesta en producción segura* [Digital]. Ra-Ma Editorial. https://ebooks.grupoeditorialrama.com/reader/puesta-en-produccion-segura

[2] Instalación — Vue.js. (s. f.). https://es.vuejs.org/v2/guide/installation.html

[3] *Python Release Python 3.11.3*. (s. f.). Python.org. https://www.python.org/downloads/release/python-3113/

[4] *Installation — Flask Documentation (2.3.x)*. (s. f.). https://flask.palletsprojects.com/en/2.3.x/installation/

[5] *¿Qué es SQL? - Explicación de lenguaje de consulta estructurado (SQL) - AWS*. (s. f.). Amazon Web Services, Inc. https://aws.amazon.com/es/what-is/sql/

[6] *Flask-SQLAlchemy — Flask-SQLAlchemy Documentation (3.0.x)*. (s. f.). https://flask-sqlalchemy.palletsprojects.com/en/3.0.x/

[7] *Servicios de informática en la nube | Microsoft Azure*. (s. f.). https://azure.microsoft.com/es-es

[8] *Index Top 10 - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/IndexTopTen.html

[9] *IBM Documentation*. (s. f.). https://www.ibm.com/docs/es/ibm-mq/7.5?topic=ssfksj-7-5-0-com-ibm-mq-sec-doc-q009740--htm

[10] Ramos, J. (s. f.). *¿Qué es JWT y cómo se diferencia de Cookies y Sesiones?* Programación y más. https://programacionymas.com/blog/jwt-vs-cookies-y-sesiones

[11] Ghate, S. (2020). Uso de cookies de sesión vs. JWT para autenticación. *HackerNoon*. https://hackernoon.com/es/usando-sesion-cookies-vs-jwt-para-autenticacion-sd2v3vci

[12] *JSON Web Token for Java - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_for_Java_Cheat_Sheet.html

[13] *Critical vulnerabilities in JSON Web Token libraries*. (s. f.). Auth0 - Blog. https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

[14] *PyJWT*. (2023, 9 mayo). PyPI. https://pypi.org/project/PyJWT/

[15] *secrets — Generate secure random numbers for managing secrets*. (s. f.). Python documentation. https://docs.python.org/3/library/secrets.html

[16] *Using HTTP cookies - HTTP | MDN*. (2023, 10 abril). https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

[17] *Authentication - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

[18] *Multifactor Authentication - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html

[19] *pyotp*. (s. f.). PyPI. https://pypi.org/project/pyotp/

[20] *Password Storage - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[21] *passlib*. (2020, 8 octubre). PyPI. https://pypi.org/project/passlib/

[22] *Forgot Password - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Forgot_Password_Cheat_Sheet.html

[23] *Denial of Service - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Denial_of_Service_Cheat_Sheet.html

[24] *A01 Broken Access Control - OWASP Top 10:2021*. (s. f.). https://owasp.org/Top10/A01_2021-Broken_Access_Control/

[25] *Authorization - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

[26] Cantelli, F., & Cantelli, F. (2022, 7 abril). ¿Qué es una Cross Site Request Forgery (CSRF) y cómo se soluciona? *Hackmetrix Blog*. https://blog.hackmetrix.com/csrf-cross-site-request-forgery/

[27] *Cross-Site Request Forgery Prevention - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[28] *Bypassing SameSite cookie restrictions | Web Security Academy*. (s. f.). https://portswigger.net/web-security/csrf/bypassing-samesite-restrictions

[29] *Cryptographic Storage - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Cheat_Sheet.html

[30] *Transport Layer Protection - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html

[31] *Wireshark · Download*. (s. f.). Wireshark. https://www.wireshark.org/download.html

[32] *HTTP Strict Transport Security - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Strict_Transport_Security_Cheat_Sheet.html

[33] *Injection Prevention - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html

[34] *SQL Injection Prevention - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

[35] *Cross Site Scripting Prevention - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

[36] Salwiczek, B. (2022, 12 febrero). Danger of using v-html in Vue applications - Bartosz Salwiczek - Medium. *Medium*. https://medium.com/@bsalwiczek/danger-of-using-v-html-in-vue-applications-see-better-approach-3def415ba32b

[37] Lethani. (2020). Cross-Site Scripting: Inyección XSS. *Hacking Lethani*. https://hackinglethani.com/es/cross-site-scripting-inyeccion-xss/

[38] *Security | Vue.js*. (s. f.). https://vuejs.org/guide/best-practices/security.html

[39] *MarkupSafe. (2023, 2 junio). PyPI.* https://pypi.org/project/MarkupSafe/

[40] *npm: dompurify*. (s. f.). npm. https://www.npmjs.com/package/dompurify

[41] *Content Security Policy - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html

[42] *Input Validation - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

[43] *A04 Insecure Design - OWASP Top 10:2021*. (s. f.). https://owasp.org/Top10/A04_2021-Insecure_Design/

[44] *Infrastructure as Code Security - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Infrastructure_as_Code_Security_Cheat_Sheet.html

[45] *Download PyCharm: Python IDE for Professional Developers by JetBrains*. (2021, 2 junio). JetBrains. https://www.jetbrains.com/pycharm/download/?section=windows

[46] *PyCharm Python Security plugin — PyCharm Python Security plugin documentation*. (s. f.) . https://pycharm-security.readthedocs.io/en/latest/

[47] *bandit*. (2023, 10 marzo). PyPI. https://pypi.org/project/bandit/

[48] *XML External Entity Prevention - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html

[49] *lxml*. (s. f.). PyPI. https://pypi.org/project/lxml/

[50] *defusedxml*. (2021, 8 marzo). PyPI. https://pypi.org/project/defusedxml/

[51] *Vulnerable Dependency Management - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Vulnerable_Dependency_Management_Cheat_Sheet.html

[52] *OWASP Dependency-Check | OWASP Foundation*. (s. f.). https://owasp.org/www-project-dependency-check/

[53] *npm-audit | npm Docs*. (s. f.). https://docs.npmjs.com/cli/v9/commands/npm-audit

[54] *safety*. (s. f.). PyPI. https://pypi.org/project/safety/

[55] *pip-audit*. (2023, 2 julio). PyPI. https://pypi.org/project/pip-audit/

[56] *A08 Software and Data Integrity Failures - OWASP Top 10:2021*. (s. f.). https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/

[57] *Deserialization - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html

[58] *pickle — Serialización de objetos Python*. (s. f.). Python documentation. https://docs.python.org/es/3/library/pickle.html

[59] *Logging - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

[60] *Logging Vocabulary - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Logging_Vocabulary_Cheat_Sheet.html

[61] *Server Side Request Forgery Prevention - OWASP Cheat Sheet Series*. (s. f.). https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html

[62] Bytemind. (2022). Cómo funciona la vulnerabilidad SSRF. *Byte Mind*. https://byte-mind.net/como-funciona-la-vulnerabilidad-ssrf/#%C2%BFQue-es-Server-Side-Request-Forgery

Beautiful CSS buttons examples - CSS Scan. (s. f.). https://getcssscan.com/css-buttons-examples

Bootstrap html snippet. bs4 beta comment list. (s. f.). https://www.bootdey.com/snippets/view/bs4-beta-comment-list#html