Hacking Modern Web Apps

Fernando Gutiérrez Calderón

Máster Universitario en Ciberseguridad y Privacidad

Seguridad empresarial

Nombre Tutor/a de TF
Pau del Canto Rodrigo
Profesor/a responsable de la asignatura
Víctor García Font

Fecha Entrega
Junio de 2024

Universitat Oberta de Catalunya

Esta obra está sujeta a una licencia de Atribución-NoComercial-SinDerivadas 4.0 Internacional de Creative Commons

Universitat Oberta de Catalunya uoc.edu

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Hacking Modern Web Apps*
Nombre del autor:	*Fernando Gutiérrez Calderón*
Nombre del consultor/a:	*Pau del Canto Rodrigo*
Nombre del PRA:	*Víctor García Font*
Fecha de entrega (mm/aaaa):	*06/2024*
Titulación o programa:	Máster Universitario en Ciberseguridad y Privacidad
Área del Trabajo Final:	*Seguridad empresarial*
Idioma del trabajo:	*Castellano*
Palabras clave	*Seguridad informática, aplicaciones web*

Resumen del Trabajo

Muchas de las aplicaciones web modernas utilizan una pila tecnológica que emplea JavaScript tanto para el cliente como el servidor. Este trabajo analiza las debilidades de este tipo de aplicaciones, con el objetivo de desarrollar una metodología de análisis de vulnerabilidades de fácil aplicación a cualquiera de esas aplicaciones web.

Incorporando el uso de herramientas gratuitas y *open source* y siguiendo las guías de seguridad y mejores prácticas de organizaciones como la fundación OWASP, cualquier equipo de desarrollo sin presupuesto de ciberseguridad puede mejorar la seguridad de sus aplicaciones web de manera sencilla.

Abstract

Many of the modern web applications are developed using a JavaScript-based technology stack, using that language for both the frontend and the backend. This master thesis analyzes the weaknesses of these kinds of applications, with the purpose of developing a vulnerability analysis methodology that is easy to apply to any of those web applications.

By incorporating the use of free and open source tools and by following the security and best practices guides from organizations such as the OWASP foundation, any development team without a cybersecurity budget can improve the security of their web applications easily.

Índice

1. Introducción 1
1.1. Contexto y justificación del Trabajo 1
1.2. Motivación del Trabajo 2
1.3. Objetivos del Trabajo 2
1.4. Impacto en sostenibilidad, ético-social y de diversidad 3
1.5. Enfoque y método seguido 4
1.6. Planificación del Trabajo 5
1.7. Breve sumario de productos obtenidos 5
1.8. Breve descripción de los otros capítulos de la memoria 6
2. El desarrollo web moderno 7
2.1. Arquitecturas web 10
2.2. *Frameworks* web 12
2.3. Pilas tecnológicas …. 17
3. Herramientas de escaneo de vulnerabilidades 18
3.1. OWASP ZAP (Zed Attack Proxy) …. 19
3.2. Burp Suite 20
3.3. Limitaciones de las herramientas de análisis de vulnerabilidades 22
4. OWASP Juice Shop 23
4.1. Arquitectura de la aplicación …. 23
4.2. Escaneo automático con ZAP 24
4.3. Escaneo manual con ZAP …. 27
4.4. Escaneo manual con Burp Suite …. 29
5. OWASP Top Ten 31
5.1. A01:2021 – Pérdida de Control de Acceso 31
5.2. A02:2021 – Fallos Criptográficos 34
5.3. A03:2021 – Inyección 37
5.4. A04:2021 – Diseño inseguro 41
5.5. A05:2021 – Configuración de seguridad incorrecta 43
5.6. A06:2021 – Componentes Vulnerables y Desactualizados 45
5.7. A07:2021 – Fallos de Identificación y Autenticación 47
5.8. A08:2021 – Fallos en el Software y en la Integridad de los Datos 51
5.9. A09:2021 – Fallos en el Registro y Monitorización 54
5.10. A10:2021 – Falsificación de Solicitudes del Lado del Servidor (SSRF) 55
6. Consideraciones específicas 56
6.1. Node.js …. 56
6.2. Angular …. 59
6.3. MongoDB 62
7. Metodología de análisis de aplicaciones web 64
7.1. Jenkins 65
7.2. Configuración de la automatización con ZAP 65
7.3. Integración del escaneo de ZAP en Jenkins 69
7.4. Integración del escaneo de paquetes `npm` 71
8. Conclusiones 73
9. Trabajo futuro 75
10. Glosario 76
11. Referencias 79

Lista de figuras

Ilustración 1: Planificación de las tareas del trabajo 5
Ilustración 2: Cronograma de las tareas del trabajo 5
Ilustración 3: Evolución de los *frameworks* de desarrollo web (1999-2019)..... 7
Ilustración 4: *Frameworks* web según el número de nuevos repositorios en GitHub 8
Ilustración 5: *Frameworks* web según el número de preguntas publicadas en Stack Overflow 9
Ilustración 6: Pila tecnológica MExN..... 17
Ilustración 7: Cliente ZAP 2.14.0 20
Ilustración 8: Cliente Burp Suite Community Edition v2024.3.1.4..... 21
Ilustración 9: Arquitectura de la aplicación web OWASP Juice Shop..... 24
Ilustración 10: Página principal de OWASP Juice Shop 24
Ilustración 11: Escaneo automático de ZAP contra OWASP Juice Shop 25
Ilustración 12: Resultado del análisis automático de Juice Shop con ZAP 26
Ilustración 13: Exploración manual en ZAP 27
Ilustración 14: Juice Shop en exploración manual desde Chrome 28
Ilustración 15: Escaneo manual de Juice Shop con OWASP ZAP 29
Ilustración 16: Modo proxy en Burp Suite 30
Ilustración 17: Resultado del escaneo manual en Burp Suite 30
Ilustración 18: Reseña de un producto en la Juice Shop..... 33
Ilustración 19: Captura de la petición de reseña en la utilidad *Requester* de ZAP 33
Ilustración 20: Publicación de reseña en nombre de otro usuario. 34
Ilustración 21: URL que sirve un listada de carpetas y ficheros 36
Ilustración 22: Fichero confidencial almacenado y transmitido sin cifrar 37
Ilustración 23: Petición de inicio de sesión capturada en la herramienta Fuzzer de ZAP 39
Ilustración 24: Configuración de la utilidad Fuzzer para inyección SQL..... 39
Ilustración 25: Resultado del Fuzzer en ZAP..... 40
Ilustración 26: Petición realizada con Fuzzer con una inyección SQL exitosa . 40
Ilustración 27: Proceso de restablecimiento de cuenta mediante preguntas de seguridad 42
Ilustración 28: Manejador de errores mostrando la traza de la pila 44
Ilustración 29: Escaneo automatizado con ZAP mostrando bibliotecas vulnerables..... 46
Ilustración 30: Captura de la petición de inicio de sesión en Burp Suite 48
Ilustración 31: Petición capturada en la herramienta Intruder de Burp Suite.... 48
Ilustración 32: Configuración de la herramienta Intruder con *payloads* de contraseñas 49
Ilustración 33: Ejemplo de petición del ataque con la herramienta Intruder de Burp Suite..... 50
Ilustración 34: Página de ayuda de una API en Juice Shop 52
Ilustración 35: Resultado de la ejecución de código arbitrario tras el uso de la API 53
Ilustración 36: Ataque XSS contra la búsqueda de productos en Juice Shop.. 60
Ilustración 37: Ejecución del código JavaScript del ataque XSS..... 60
Ilustración 38: DOM modificado tras el ataque XSS..... 60

Ilustración 39: Captura en Burp Suite de la petición de modificación de una reseña 62
Ilustración 40: Ejemplo de reseña modificada tras la inyección NoSQL..... 63
Ilustración 41: Configuración de URLs en el contexto 66
Ilustración 42: Configuración de tecnologías utilizadas en nuestra aplicación . 67
Ilustración 43: Configuración de las opciones de autenticación del contexto... 68
Ilustración 44: Nuevo plan de automatización 68
Ilustración 45: Plan de automatización recién creado 69
Ilustración 46: Informe HTML resultado de la ejecución automatizada 70
Ilustración 47: *Build* fallido tras el fallo en el paso de evaluación del informe ZAP 71
Ilustración 48: Evaluación de la seguridad de los paquetes con `npm audit` ... 72

1. Introducción

1.1. Contexto y justificación del Trabajo

Asistimos en los últimos años a un proceso de aceleración de la transformación digital en las empresas. La pasada pandemia de COVID-19 supuso un punto de inflexión en esta evolución, especialmente para las pymes. Muchas de las empresas que iban rezagadas en esa transformación digital, se vieron rápidamente en la necesidad de acelerarla.

La web estuvo en el centro de este cambio durante el confinamiento. Gracias a las aplicaciones web y los servicios en la nube, compañías de todos los sectores y tamaños pudieron implementar nuevas formas de conectar y comunicarse con sus trabajadores, clientes y proveedores.

Los cambios positivos de esta transformación permitieron que hoy disfrutemos de modelos de trabajo y educación flexibles. Sin embargo, no todos fueron aspectos positivos los que se derivaron de este proceso.

La Agencia de la Unión Europea para la Ciberseguridad (ENISA), en su informe *ENISA Threat Landscape* de 2020 [1], se hace eco de estos otros aspectos. Los profesionales de las TIC debieron afrontar grandes retos derivados de una nueva realidad de teletrabajo que aumentaba considerablemente la superficie de ataque de las empresas.

De la lista de las 15 principales amenazas recogidas en el informe, dos están relacionadas con tecnologías web: "ataques basados en la web", en segunda posición por detrás del *malware* y "ataques que afectan a aplicaciones web", en cuarto lugar, tras el *phishing*.

Esto refleja la importancia que ha cobrado recientemente la seguridad de las aplicaciones web modernas, tema sobre el que gira este trabajo.

La necesidad de aumentar la seguridad de estas aplicaciones, o la seguridad informática en general, no ha disminuido en 2024. El complejo panorama geopolítico actual presenta grandes retos para los equipos de ciberseguridad. Se observan ciberataques cada vez más sofisticados, que ahora se ven potenciados gracias al uso de diversas técnicas de inteligencia artificial (IA).

En este escenario, la seguridad de las aplicaciones web es una pieza más para mejorar este panorama y una pieza importante.

1.2. Motivación del Trabajo

Como desarrollador web *full-stack* en mi experiencia laboral reciente, también he presenciado un punto de inflexión en el interés de nuestros clientes por los aspectos relacionados con la seguridad de nuestros productos.

Eran principalmente los clientes grandes y la administración pública quienes nos empezaban a enviar listas de comprobación para ver si nuestras aplicaciones cumplían ciertas medidas de seguridad o seguían buenas prácticas.

Como microempresa dedicada al desarrollo de software en el ámbito de la Prevención de Riesgos Laborales, no teníamos personal dedicado a la ciberseguridad. Tampoco seguíamos ninguna metodología de seguridad ni teníamos conocimientos específicos en desarrollo de *software* seguro.

Esta información requerida por nuestros clientes se basaba, habitualmente, en las buenas prácticas y recomendaciones de la fundación OWASP (Open Worldwide Application Security Project). Algunos conceptos (como la inyección SQL), sí nos eran familiares al equipo de desarrollo; mientras que otros nos resultaban más exóticos.

Pero la realidad de una pyme es que el día a día deja poco tiempo para indagar en estos conceptos tan importantes. La motivación de este trabajo es precisamente esa, poder investigar en detalle sobre la seguridad de las aplicaciones web modernas.

1.3. Objetivos del Trabajo

Los objetivos principales de este trabajo son:

- Diseñar una metodología de análisis de seguridad de aplicaciones web desarrolladas con *frameworks* y pilas tecnológicas modernas.
- Poner en práctica la metodología desarrollada para detectar las vulnerabilidades de aplicaciones web actuales concretas.

Como objetivos secundarios se plantean los siguientes:

- Investigar cuáles son los *frameworks* de desarrollo web más importantes en la actualidad y qué lenguajes de programación utilizan.
- Analizar las características de seguridad que incorporan dichos *frameworks* por defecto y cuáles deben implementarse aparte.
- Identificar las principales herramientas de análisis de vulnerabilidades y el papel que cumplen en el desarrollo de software seguro.
- Seleccionar las técnicas de prueba de software que se utilizarán en la metodología de análisis de seguridad desarrollada.
- Identificar los principales riesgos de seguridad a los que están sometidas las aplicaciones web actuales.
- Buscar aplicaciones que presenten las vulnerabilidades analizadas para aplicarles la metodología desarrollada.

1.4. Impacto en sostenibilidad, ético-social y de diversidad

La competencia de compromiso ético y global (CCEG) de la UOC se define de la siguiente manera a nivel de TFM:

> *"Actuar de manera honesta, ética, sostenible, socialmente responsable y respetuosa con los derechos humanos y la diversidad, tanto en la práctica académica como en la profesional, y diseñar soluciones para mejorar estas prácticas".*

Respecto a la dimensión de sostenibilidad medioambiental, este trabajo no tiene un gran impacto (más allá de la energía consumida para crearlo). Esta es mínima en comparación con otro trabajo cuyo objetivo fuera la fabricación de un producto o el desarrollo de un nuevo proceso industrial, por ejemplo.

Sí se puede conectar este trabajo con aspectos de derechos humanos si acudimos a la Carta de los Derechos Fundamentales de la UE. Esta recoge los derechos y libertades de la ciudadanía de la UE, destacando estos dos artículos:

- 8.1: *"Toda persona tiene derecho a la protección de los datos de carácter personal que le conciernan"*.
- 17.2: *"Se protege la propiedad intelectual".*

Cada vez mayor cantidad de nuestros datos personales, así como la propiedad intelectual de personas y empresas, se comparte y procesa mediante aplicaciones web y servicios en la nube. El uso de herramientas seguras es fundamental para evitar ataques contra la seguridad de la información gestionada por estas.

Por último, el presente trabajo tiene un gran impacto positivo en cuanto a comportamiento ético y responsabilidad social. Para ello debemos repasar el significado de las palabras *hacker* y *hackear*.

Ambas están recogidas en la edición de 2023 del diccionario de la RAE, en su primera acepción, como sinónimos de pirata informático: *"persona que accede ilegalmente a sistemas informáticos ajenos para apropiárselos u obtener información secreta"*.

Puede verse que recoge una visión un tanto estereotipada de estas personas (tal vez por la influencia del entretenimiento audiovisual). El delito informático es, obviamente, una de las formas que toma esta figura. Pero es más habitual en el ámbito de la ciberseguridad que el término *hacker* se refiera al profesional de la seguridad informática encargado de proteger sistemas, aplicaciones y datos.

Es en la segunda acepción de la voz castellana, jáquer, donde sí encontramos esta segunda definición más positiva: *"persona con grandes habilidades en el manejo de computadoras que investiga un sistema informático para avisar de los fallos y desarrollar técnicas de mejora"*.

Con el término *hacking* del título de este TFM nos estamos refiriendo, por tanto, a este segundo tipo de profesional. Es precisamente el comportamiento ético de estos y estas profesionales lo que hace que desempeñen su labor con el objetivo de mejorar aquellos sistemas y aplicaciones vulnerables.

1.5. Enfoque y método seguido

La primera parte de este TFM es de aspecto teórico. En una primera sección se analizará la arquitectura de una aplicación web moderna y la pila tecnológica que emplea. También se indagará sobre las características de seguridad que ofrecen este tipo de aplicaciones y las vulnerabilidades que pueden presentar.

Para desarrollar esta primera sección, el trabajo se basará en diversos estudios y trabajos de investigación sobre tecnologías web actuales.

En una segunda sección se estudiarán las herramientas especializadas en detección de vulnerabilidades de aplicaciones web. Se hará referencia principalmente a herramientas gratuitas y *open-source*, de forma que una pequeña empresa pueda usarlas fácilmente en su análisis. No obstante, se mencionarán también herramientas comerciales que son referente en la industria, si estas disponen de una versión gratuita de tipo *community*.

Para desarrollar esta segunda sección, el trabajo se basará en las páginas web y documentación técnica de los fabricantes de estas herramientas.

La segunda parte del trabajo tiene un carácter más práctico. Se probarán las herramientas investigadas sobre una o varias aplicaciones web modernas reales. Esto permitirá comprobar la eficacia de las herramientas detectando las vulnerabilidades web más importantes en la actualidad.

De esta parte práctica se derivará la metodología de análisis de aplicaciones web que es objetivo de este trabajo. Esta debe poder servir a una pequeña empresa que desarrolle aplicaciones web y no tenga presupuesto de ciberseguridad o personal con formación específica para analizar sus aplicaciones web fácilmente.

Para desarrollar esta segunda parte, acudiremos a los recursos de la fundación OWASP, referente en cuanto a seguridad web, que publica:

- Una lista actualizada de las vulnerabilidades web más importantes.
- Una guía de desarrollo de aplicaciones web seguras.
- Un catálogo de aplicaciones inseguras listas para ser sometidas al análisis de vulnerabilidades.

El enfoque de la parte práctica será el de la ciberseguridad ofensiva: pensando como un atacante buscaremos las vulnerabilidades de las aplicaciones y las formas de explotarlas.

1.6. Planificación del Trabajo

La siguiente tabla muestra la planificación de tareas. La parte teórica descrita en la metodología corresponde a las secciones "Arquitectura aplicaciones web" y "Herramientas análisis vulnerabilidades". La sección "Análisis de seguridad" corresponde a la parte práctica.

Tareas	Fechas
Plan de trabajo	28 de febrero de 2024 → 12 de marzo de 2024
Arquitectura aplicaciones web	13 de marzo de 2024 → 9 de abril de 2024
Revisión bibliográfica	13 de marzo de 2024 → 20 de marzo de 2024
Redacción memoria	21 de marzo de 2024 → 9 de abril de 2024
Herramientas análisis vulnerabilidades	10 de abril de 2024 → 7 de mayo de 2024
Investigación herramientas	10 de abril de 2024 → 17 de abril de 2024
Redacción memoria	18 de abril de 2024 → 7 de mayo de 2024
Análisis de seguridad	8 de mayo de 2024 → 11 de junio de 2024
Revisión recursos OWASP	8 de mayo de 2024 → 22 de mayo de 2024
Utilización herramientas	23 de mayo de 2024 → 11 de junio de 2024
Redacción memoria	23 de mayo de 2024 → 11 de junio de 2024
Presentación y defensa	12 de junio de 2024 → 28 de junio de 2024
Edición vídeo	12 de junio de 2024 → 18 de junio de 2024
Defensa TFM	24 de junio de 2024 → 28 de junio de 2024

Ilustración 1: Planificación de las tareas del trabajo

La siguiente figura muestra la misma planificación en forma de cronograma:

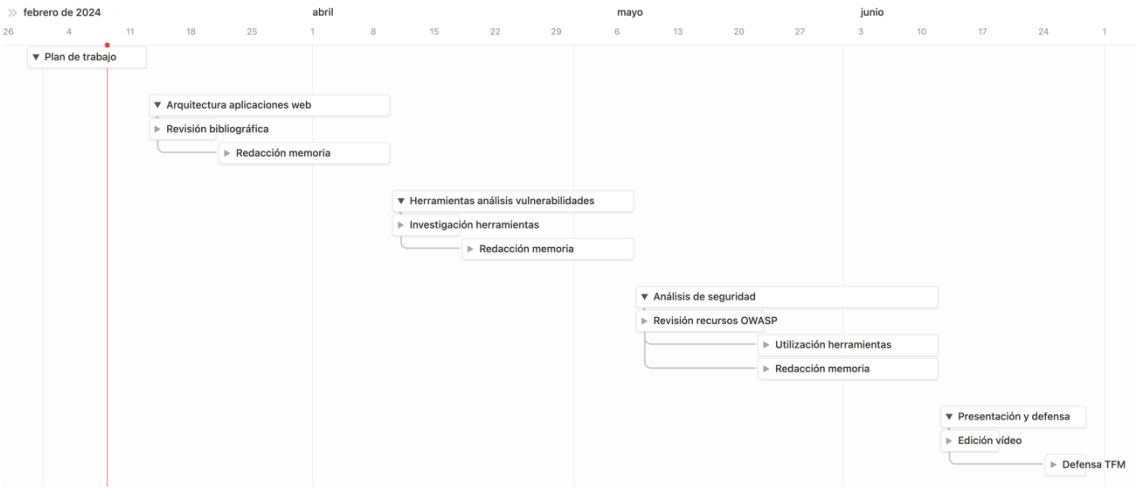


Ilustración 2: Cronograma de las tareas del trabajo

1.7. Breve sumario de productos obtenidos

Una metodología de análisis de vulnerabilidades de aplicaciones web modernas que, utilizando herramientas y recursos *open-source*, permita a una pequeña compañía que desarrolla aplicaciones web mejorar de manera sencilla la seguridad de las aplicaciones que desarrolla.

1.8. Breve descripción de los otros capítulos de la memoria

El segundo capítulo trata sobre el desarrollo web moderno, presentando una visión global sobre arquitecturas, *frameworks* y pilas tecnológicas web modernas.

En el tercer capítulo se analizan las herramientas de análisis de vulnerabilidades de aplicaciones web que se utilizarán como de la metodología de análisis. Se analiza una herramienta *open-source* y la versión gratuita de una herramienta comercial.

En el cuarto capítulo se introduce una aplicación web deliberadamente insegura, que sirve de base para explorar las vulnerabilidades de las aplicaciones web modernas, utilizando para ello las herramientas descritas en el capítulo anterior.

El quinto capítulo describe en sí las diez vulnerabilidades más importantes en las aplicaciones web según la fundación OWASP. Para cada vulnerabilidad, se presenta una descripción, las formas de mitigar dicha vulnerabilidad y un ejemplo de explotación con las herramientas descritas en el capítulo tercero. También se incluyen, si aplica, las recomendaciones específicas de seguridad de los distintos *frameworks* que se han comentado.

En el capítulo sexto, se presentan algunas consideraciones de seguridad específicas de los distintos módulos de la pila tecnológica que se ha introducido en el capítulo segundo.

El séptimo capítulo presenta una metodología de análisis de aplicaciones web sencilla que permita a una pequeña empresa sin personal o presupuesto de ciberseguridad mejorar la seguridad de sus aplicaciones web utilizando las herramientas y los recursos *open-source* que se han presentado en el trabajo.

Los capítulos octavo y noveno cierran el trabajo, con las conclusiones y trabajo futuro, respectivamente.

El capítulo décimo es el glosario, donde se incluyen ciertos términos que aparecen a lo largo del documento.

El último capítulo recoge el material que se ha referenciado para la elaboración del trabajo.

2. El desarrollo web moderno

A principios de este siglo la *World Wide Web* era joven todavía y bastante desconocida fuera del ámbito académico o de la empresa. Las primeras páginas y sitios web eran sencillos, pero fueron rápidamente ganando en complejidad a medida que más y más público general se conectaba a internet y las tecnologías web iban evolucionando.

El desarrollo web entonces era también rudimentario. No existían *frameworks* web como los conocemos hoy, aunque sí herramientas de creación de sitios web, como Dreamweaver de la desaparecida Macromedia (ahora Adobe).

Esta herramienta la utilizaba principalmente el equipo de diseño gráfico. Pero también quienes nos dedicábamos por aquel entonces a programar estos primeros sitios web. JavaScript se encontraba en sus primeras versiones y se utilizaba, sobre todo, para añadir interactividad a las páginas web.

Utilizar herramientas específicas de diseño web, como Dreamweaver, tenía como ventaja la interfaz WYSIWYG de este editor, que mostraba el aspecto final del sitio web. Pero carecían de cualquier otra ayuda para la programación en JavaScript (al menos en las versiones iniciales). En ese sentido, editar código no era mucho más sencillo que hacerlo con el bloc de notas.

Requería, por tanto, un profundo conocimiento de las tecnologías subyacentes a cualquier sitio web: HTML, CSS y JavaScript y cómo cada una de ellas se interrelaciona con las demás.

Pronto comenzaron a surgir los primeros *frameworks* web de servidor, como ASP.NET, Symfony o Rails. La ventaja de estos era que permitían generar sitios web de manera dinámica, a menudo a partir de bases de datos. El contenido que se enviaba al navegador web del cliente se generaba en el servidor, evitando crear de forma manual cada página.

Es en torno a 2008 cuando se empieza a producir una explosión en el número de *frameworks* de desarrollo web, como se puede observar en la siguiente figura [2]:

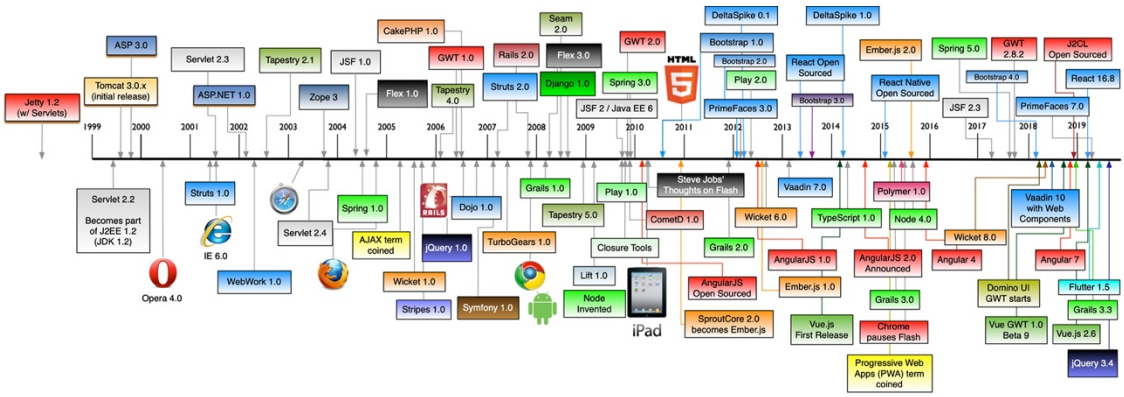


Ilustración 3: Evolución de los *frameworks* de desarrollo web (1999-2019)

Un *framework* web es una herramienta de desarrollo que permite construir aplicaciones web de manera más rápida y sencilla al ofrecer un modelo básico, un conjunto de APIs relevantes, así como bibliotecas y extensiones [3] (P. 1).

A pesar del elevado número de *frameworks* de desarrollo web observados en la figura anterior, no todos gozan de la misma popularidad en la actualidad. Este es un factor importante que considerar, especialmente al iniciar un nuevo proyecto.

Utilizar un *framework* popular tiene ventajas como la fiabilidad y la seguridad, las actualizaciones y extensiones de funcionalidad más frecuentes o un mejor soporte por parte de la comunidad. Además, es importante en los aspectos de adquisición y retención de talento y las políticas de formación de los equipos de desarrollo [3] (P. 3).

En el estudio comparativo sobre popularidad de *frameworks* de desarrollo web, Swacha y Kulpa establecen dos indicadores para el análisis de la popularidad relativa de estos entornos [3] (P. 5): 1) el número de nuevos repositorios creados en GitHub asociados a cada *framework* y 2) el número de consultas realizadas en Stack Overflow relativas a cada *framework*.

En la siguiente gráfica se muestra la popularidad relativa de los *frameworks* de desarrollo web en torno al primer indicador (nuevos repositorios en GitHub):

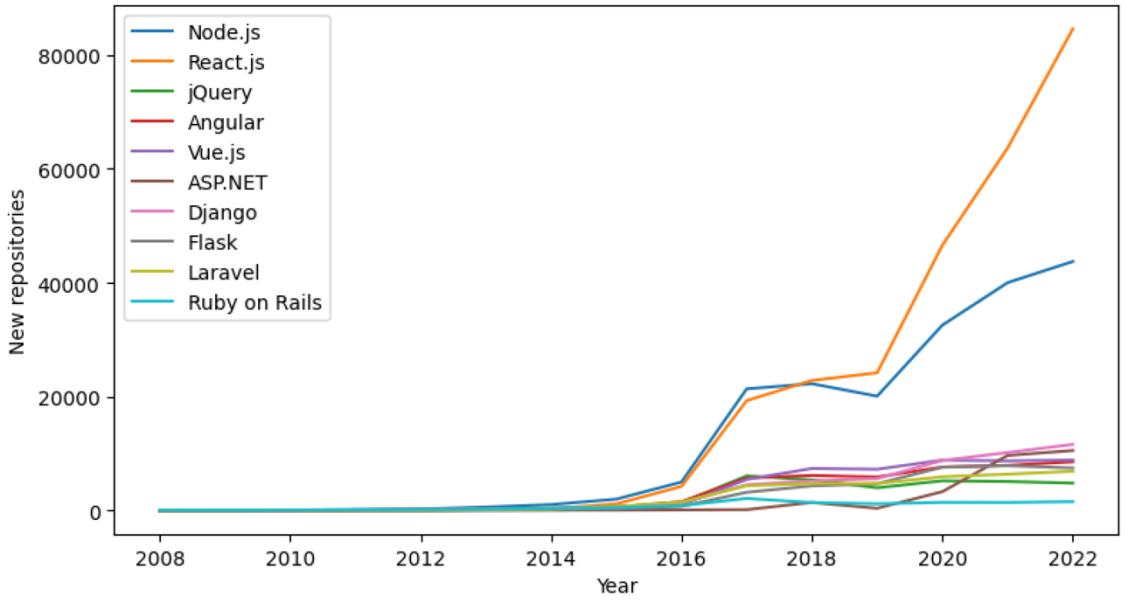


Ilustración 4: *Frameworks* web según el número de nuevos repositorios en GitHub
Fuente: [3] (P. 11)

La figura siguiente muestra la popularidad relativa de los *frameworks* de desarrollo web en torno al segundo indicador (preguntas publicadas en Stack Overflow):

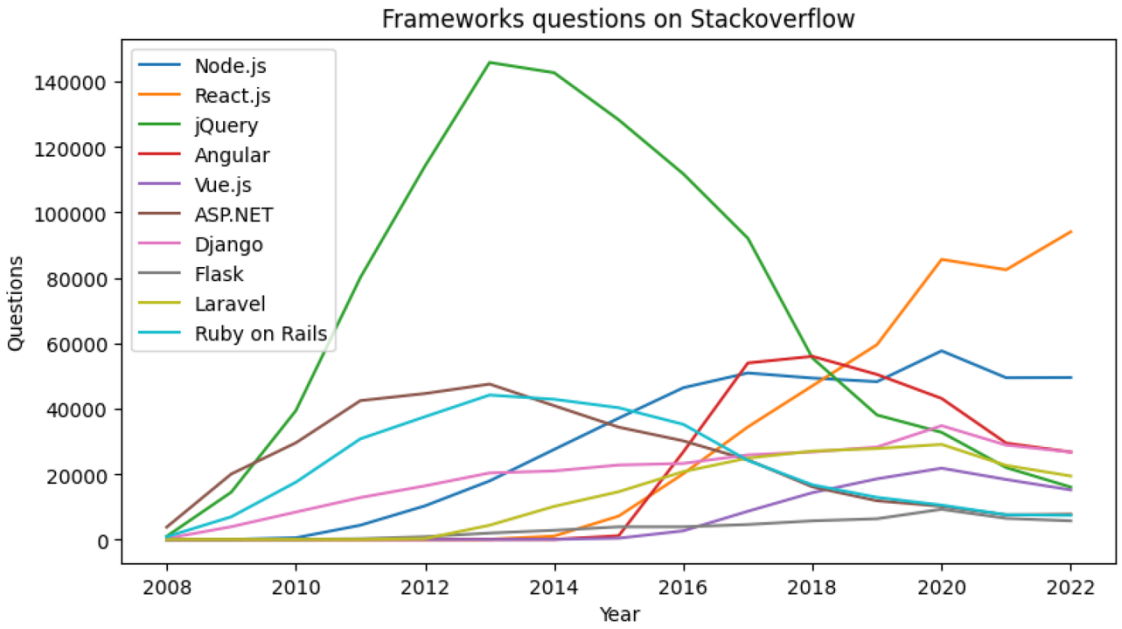


Ilustración 5: *Frameworks* web según el número de preguntas publicadas en Stack Overflow
Fuente: [3] (P. 12)

El estudio continúa con el análisis estadístico de ambos juegos de datos para determinar si la clasificación de popularidad en función de un indicador está correlacionada con la clasificación en función del otro.

Pero, sin entrar en este análisis, podemos concluir de la observación de ambas gráficas que Node.js y React.js están siendo muy populares en los últimos años[1]. Node.js es un entorno de ejecución de JavaScript fuera del navegador y React.js una biblioteca JavaScript para desarrollo del *frontend*.

La popularidad de JavaScript también ha aumentado en los últimos años: la mitad de los *frameworks* analizados en el estudio utilizan ese lenguaje[2]. Al uso tradicional que tenía en el *frontend* de la aplicación web (el código que se ejecuta en el navegador), se ha añadido la posibilidad de utilizarlo también en el desarrollo del *backend* (con Node.js).

Sirva el estudio citado aquí para delimitar lo que se entiende en este trabajo por "aplicación web moderna": aquella que está desarrollada con los *frameworks* web más relevantes en la actualidad y donde se utiliza JavaScript como lenguaje principal de desarrollo.

[1] Se debe tener en cuenta que el estudio consolida algunos *frameworks* que son añadidos o versiones alternativas de otros (por ejemplo, incluye Express dentro de Node.js y Next.js dentro de React.js) [3] (P. 6).
[2] Node.js, React.js, jQuery, Angular y Vue.js.

2.1. Arquitecturas web

En la actualidad existen dos paradigmas comunes de desarrollo de aplicaciones modernas: la arquitectura monolítica y la basada en microservicios [4]. A continuación, se describen las características de ambas arquitecturas.

Arquitectura monolítica [4]

Una aplicación web tradicional utiliza una arquitectura monolítica, donde la aplicación se construye como una única unidad. Estas aplicaciones se dividen internamente en las tres capas clásicas siguientes:

- La aplicación en el lado del servidor (*backend*) gestiona las peticiones HTTP del cliente, ejecuta la lógica de negocio, accede a la BD y genera una vista HTML para enviar al cliente (navegador).
- La interfaz de usuario en la parte cliente (*frontend*) está formada por páginas HTML y código JavaScript y se ejecuta en el navegador del equipo del usuario.
- La BD, habitualmente un SGBD de tipo relacional.

La aplicación en el *backend* es un monolito, un único ejecutable que corre como único proceso en un servidor de aplicaciones. Cualquier cambio en la aplicación supone construir y desplegar una nueva versión de la aplicación que reemplaza a la anterior. Cuando se requiere *escalar* la aplicación, se hace horizontalmente, es decir, replicando la aplicación monolítica en varios servidores bajo balanceo de carga [5].

La ventaja de esta arquitectura es su sencillez si se compara con otro tipo de aplicaciones distribuidas y es el enfoque natural al construir una aplicación. Sin embargo, a medida que el tamaño y la complejidad crecen, empiezan a surgir algunos problemas:

- Modificar el código de la aplicación se complica y este comienza a comportarse de manera impredecible.
- Los cambios en un módulo pueden provocar cambios inesperados en otros módulos.
- El creciente tamaño del monolito implica un mayor tiempo de inicialización, que frena el desarrollo y el despliegue.
- Se vuelve cada vez más difícil mantener una estructura modular en la aplicación y conseguir que los cambios en un módulo sólo afecten a este.

Estos inconvenientes condujeron a la arquitectura de microservicios.

Arquitectura de microservicios [4]

Lewis y Fowler [5] definieron este tipo de arquitectura como *"un enfoque de desarrollo de una aplicación como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos ligeros, a menudo un recurso API HTTP".*

Los principios de la arquitectura son:

- Cada servicio tiene una única responsabilidad, de acuerdo con los principios SOLID y dos servicios no comparten la misma responsabilidad.
- Los microservicios son autónomos: son servicios autocontenidos y desplegables independientemente unos de otros. Cada uno contiene todas sus dependencias (bibliotecas, entornos de ejecución, etc.).
- Los servicios exponen sus *endpoints* como APIs y abstraen los detalles de implementación. Su estructura interna (arquitectura, tecnologías, lenguaje de programación, BD, etc.) queda totalmente oculta tras la API.

El modelo de comunicación de los microservicios difiere de modelos basados en arquitectura orientada a servicios (SOA) tales como bus de servicio empresarial (ESB). No existe un estándar de comunicación o mecanismos de trasporte, comunicándose los microservicios con protocolos ligeros como HTTP y REST.

La principal ventaja de la arquitectura de microservicios es la descomposición de aplicaciones complejas en componentes más pequeños que son más fáciles de desarrollar, gestionar y mantener que en una aplicación monolítica. Mientras la API pública no cambie, las modificaciones a un servicio son más directas y menos costosas que el modelo tradicional.

Otra ventaja de esta arquitectura es la tolerancia a fallos: un fallo en un componente no afecta a todo el sistema, ya que los microservicios que sí funcionan pueden seguir respondiendo a las peticiones de los usuarios. Se puede identificar la funcionalidad de negocio crítica y desplegar los microservicios implicados de manera más redundante.

A pesar de las ventajas, la arquitectura de microservicios presenta algún inconveniente relativo a su naturaleza distribuida. El despliegue, *escalado* y monitorización de un sistema multi-servicio es más complejo que en el caso de una aplicación monolítica. Por eso, al desarrollar este tipo de aplicaciones, es frecuente emplear automatización de pruebas y procesos de integración y entrega continua (CI/CD).

Escalado **de aplicaciones** [4]

La *escalabilidad* es la propiedad de un sistema de gestionar una carga de trabajo creciente añadiendo recursos al sistema. El *escalado* vertical implica añadir más recursos a una máquina (CPU, memoria, almacenamiento). Y el *escalado* horizontal consiste en añadir más máquinas y distribuir la carga entre ellas.

El escalado horizontal es más frecuente con aplicaciones basadas en microservicios, aunque una aplicación monolítica también se puede escalar horizontalmente ejecutando varias instancias con balanceo de carga. Pero el escalado horizontal en una aplicación monolítica puede no ser efectivo ya que, al replicarse toda la aplicación, los servicios menos demandados también consumen recursos, aunque no se utilicen. Mientras que, en una aplicación de microservicios, sólo se replicarán los servicios más demandados, llevando a un mejor aprovechamiento de los recursos.

2.2. *Frameworks* web

Se ha definido antes un *framework* web como una herramienta de desarrollo dirigida a construir aplicaciones web de manera más rápida y sencilla, por las facilidades que incluye (APIs, bibliotecas, extensiones, etc.). Además de la capa de BD, las otras dos capas clásicas en que se divide una aplicación web (*frontend* y *backend*) nos indica que existen *frameworks* web orientados al desarrollo del cliente y del servidor.

A continuación, se analizarán las características de los siguientes *frameworks* web: 1) *Backend*: Node.js y Express. 2) *Frontend*: React.js, Angular y Vue. Por último, se hablará del papel de MongoDB como SGBD no relacional.

2.2.1 Node.js [6]

Node.js es una plataforma de software que permite crear un servidor web y construir aplicaciones web sobre este. En sí mismo no es un servidor web ni tampoco un lenguaje. Pero contiene una biblioteca de servidor HTTP integrada, de forma que no es necesario un servidor web independiente como Apache o Internet Information Services (IIS).

Node.js ha ganado en popularidad al utilizar JavaScript, el lenguaje con el que los desarrolladores web ya estaban familiarizados y que ha permitido el desarrollo *full-stack* sin tener que conocer dos lenguajes, JavaScript en el *frontend* y otro lenguaje como PHP o Ruby en el *backend*.

Node.js corre sobre un solo hilo, mientras que los servidores web tradicionales son multi-hilo. Por eso es rápido y hace un uso eficiente de los recursos, pudiendo servir a más usuarios con menos recursos que la mayoría de las otras tecnologías de servidor.

Con el modelo multi-hilo, cada cliente recibe un hilo separado y una cantidad de memoria RAM. En Node.js, todos los clientes usan un mismo proceso central. Para que este enfoque funcione, el código no debe ser bloqueante. Operaciones como el acceso a ficheros o BD podrían evitar que los clientes reciban respuesta. Por eso, este tipo de operaciones se deben hacer de manera asíncrona para evitar el bloqueo del proceso principal.

Respecto a la seguridad, existe el grupo de trabajo de seguridad Node.js (*Node.js Security Working Group*[3]), que se encarga de mejorar el estado de la seguridad del ecosistema Node.js. Este grupo produce el documento de mejores prácticas de Node.js [7], que incluye: 1) una guía de mejores prácticas, 2) una explicación de los ataques mencionados en el modelo de amenazas [8] y 3) un conjunto de mejores prácticas en cuanto a la gestión de bibliotecas de terceros y dependencias entre módulos.

Se analizarán estos documentos en la parte práctica de este trabajo, cuando se desarrolle la metodología de análisis de seguridad.

[3] https://github.com/nodejs/security-wg

2.2.2 Express [6]

Node.js, como plataforma o entorno de ejecución de JavaScript en el servidor, no indica cómo debe utilizarse o configurarse. A través de su gestor de paquetes, npm (*Node Package Manager*), se tiene acceso a cientos de miles de módulos para extender la funcionalidad de la aplicación. Pero eso implica que, cada vez que se crea una nueva aplicación web, hay unas cuantas tareas a realizar. Express es un *framework* web para Node.js diseñado para facilitar esas tareas.

Express levanta un servidor web que escucha las peticiones entrantes y devuelve las respuestas adecuadas. También define una estructura de directorios, en una de cuyas carpetas se puede ubicar el contenido estático, que se sirve de manera no bloqueante.

Express ofrece una interfaz sencilla para dirigir una URL entrante a un cierto trozo de código, que puede servir un recurso estático, leer o escribir información a una BD, etc.

Soporta muchos sistemas de plantillas web, que permiten construir páginas HTML de forma inteligente, utilizando componentes reutilizables y datos de la aplicación. Express los compila juntos y los sirve al navegador como HTML.

HTTP es un protocolo sin estado, por tanto, Node.js sólo recibe una serie de peticiones HTTP sin recordar a un usuario entre peticiones. Express permite utilizar sesiones, de forma que se pueda identificar a los usuarios a lo largo de sus diferentes peticiones y las páginas que visita.

Respecto a la seguridad, la página oficial de Express publica el documento de mejores prácticas de seguridad para aplicaciones en producción [9], al que también se hará referencia en la parte práctica del trabajo.

2.2.3 React.js

React.js es una biblioteca JavaScript para construir interfaces web de usuario y nativas. Desarrollada por Facebook (ahora Meta), utiliza un enfoque basado en componentes. Cada uno de ellos puede contener su propio estado y son reutilizables en toda la interfaz de usuario [10].

La característica principal de React es que utiliza un DOM virtual para llevar un registro de los cambios que se hacen a los componentes. En lugar de *renderizar* el DOM completo cuando un componente cambia, los cambios se aplican al DOM virtual. Luego se comparan los DOM del navegador y el DOM virtual y se aplican al DOM los cambios de una vez. Gracias a ello solo se vuelven a *renderizar* los componentes que han cambiado. Esto se conoce como una interfaz *reactive* [10].

Para crear los distintos elementos de los componentes se utiliza la función `createElement()` del módulo React. Con jerarquías de elementos anidadas, esto se puede volver complejo. Para solucionarlo, React dispone de un lenguaje de *markup* llamado JSX (JavaScript XML). Es similar a HTML y se puede utilizar para construir un elemento o jerarquía de elementos, con un aspecto muy similar

a HTML, en lugar de las llamadas a `createElement()`. Pero los motores de JavaScript no entienden JSX y este debe ser transformado en JavaScript normal. Para ello se utiliza un compilador, Babel [11].

Una variante de React, React Native, permite la creación de aplicaciones móviles independientes de la plataforma. El código de la aplicación se escribe también en JavaScript y JSX, pero se *renderiza* como código nativo, específico a la plataforma (Android, iOS, etc.). Poder utilizar la misma base de código, sin que el equipo de desarrollo deba aprender nuevos lenguajes, lo hace muy potente [10].

2.2.4 Angular

Angular es un *framework* web para desarrollar el *frontend* de la aplicación desarrollado por Google. Las versiones 1.x se llamaban AngularJS y estaban escritas en JavaScript. De la versión 2 en adelante, se llaman Angular y se basan en TypeScript [10].

Angular se ha diseñado específicamente para crear aplicaciones de página única (SPA: *Single-Page Application*). Este tipo de aplicaciones se ejecutan completamente en el navegador y nunca recargan la página por completo. Gmail es un ejemplo de aplicación web SPA [6].

Este tipo de aplicaciones pueden reducir la cantidad de recursos necesarios en el servidor ya que el navegador del usuario hace gran parte del trabajo. El servidor se limita a servir ficheros estáticos y datos a petición del usuario. Toda la lógica de la aplicación, proceso de datos, etc. se gestiona en el cliente [6].

Como React, Angular sigue un enfoque basado en componentes, siendo estos los bloques con que se construye la aplicación. Cada componente incluye una clase TypeScript, una plantilla HTML y estilos CSS [10].

A diferencia de React, los componentes de Angular se construyen a través de una plantilla. No están limitados a devolver un solo elemento DOM (como un `<div>`). En lugar de JSX, Angular extiende el HTML con sintaxis adicional entre llaves dobles. Estas indican a Angular que interpole su contenido, el cual cambia cuando lo hace el estado. Se pueden pasar datos a los componentes como propiedades, que se indican con corchetes [10].

Para visualizar la interfaz de usuario, Angular utiliza un DOM incremental, diferente del DOM virtual de React. Cada componente se compila en una serie de instrucciones, contenidas en las plantillas, que crean el DOM y lo actualizan cuando los datos cambian [10].

Angular usa JavaScript para generar el contenido HTML a partir de las plantillas y los datos. Si el navegador no soporta JavaScript o hay un error en el código, la aplicación web no se mostrará. Esto también causa problemas con los motores de búsqueda, que no ejecutarán el código JavaScript [6].

La documentación oficial de Angular incluye una guía de seguridad [12], donde se describen las protecciones que incorpora Angular frente a vulnerabilidades y ataques comunes, tales como *cross-site scripting*.

2.2.5 Vue.js [10]

Es un *framework* de JavaScript declarativo y basado en componentes, similar a React y Angular. Está escrito en TypeScript y soporta este lenguaje. El *framework* es progresivo, es decir, se puede comenzar usando como biblioteca base e ir añadiendo utilidades poco a poco.

Vue utiliza el concepto de *rendering* declarativo. Como Angular, utiliza una sintaxis de plantillas que permite extender el HTML básico. La salida HTML se describe declarativamente en función del estado JavaScript. Con un enfoque basado en componentes, un fichero Single-File Component (SFC) encapsula la lógica JavaScript, la plantilla HTML y los estilos CSS en un único fichero.

Una característica principal es la reactividad. Vue gestiona automáticamente los cambios de estado para actualizar de manera eficiente el DOM cuando ocurren. También utiliza un DOM virtual actualizando, como React, los cambios de manera rápida. Las plantillas se compilan en código altamente optimizado. Vue decide el mínimo número de componentes que actualizar y aplica el mínimo número de manipulaciones del DOM cuando cambia el estado.

En la sección de seguridad de su documentación [13], se describen las protecciones que Vue incluye por defecto, así como las mejores prácticas de seguridad y potenciales peligros.

2.2.6. MongoDB

Para mediados de los años 80, los sistemas de gestión de bases de datos relacionales (RDBMS, por sus siglas en inglés) y SQL se habían convertido en el estándar para almacenar y consultar información estructurada. Otros enfoques para almacenar datos, como el modelo de red o el jerárquico, incluso las BD XML o las orientadas a objetos, generaron mucho revuelo, pero no llegaron a desplazar a los sistemas relacionales [14] (CAP. 2).

Tras más de 30 años de hegemonía, las BD NoSQL (*Not Only SQL*) son el último intento de acabar con el dominio de los RDBMS. Surgen de estos factores [14]:

- La necesidad de una mayor escalabilidad que la ofrecida por los SGBD relacionales.
- La preferencia extendida por productos de BD *open-source* antes que el *software* comercial.
- Las operaciones de consulta especializadas no soportadas bien por el modelo relacional.
- La frustración con las restricciones impuestas por los esquemas relacionales y el deseo de un modelo de datos más dinámico y expresivo.

La mayor parte del desarrollo moderno se hace con lenguajes de programación orientados a objetos y esto conduce a una crítica frecuente del modelo de datos relacional y del lenguaje SQL: se requiere una capa de traducción entre los objetos del código de la aplicación y el modelo de tablas, filas y columnas [14].

Esta discrepancia entre las diferentes formas de representar la información se conoce como adaptación de impedancias objeto-relacional. Aunque existen *frameworks* de mapeo relacional de objetos (ORM: *Object-relational mapping*) como Hibernate para solucionar este problema, no ocultan completamente las diferencias entre ambos modelos [14].

MongoDB es un sistema gestor de base de datos (SGBD) no relacional utilizado para muchas y diversas aplicaciones en el desarrollo web y móvil, IoT (*Internet of Things*), analítica, gestión de contenidos, etc. [10].

A diferencia de una BD relacional, que guarda la información en los registros de las tablas, MongoDB maneja los conceptos de colecciones y documentos. La colección es equivalente a la tabla y el documento al registro. Sin embargo, los documentos se encuentran en formato JSON (*JavaScript Object Notation*) [10].

JSON es un formato de intercambio de información que, aunque basado en JavaScript, es completamente independiente del lenguaje. Los objetos JSON son contenedores asociativos, donde una clave de tipo cadena se vincula a un valor (número, cadena, valor lógico, array, valor vacío u otro objeto). Casi todos los lenguajes de programación implementan esta estructura de datos: objetos en JavaScript, diccionarios en Python, etc. [15].

MongoDB internamente utiliza BSON (Binary JSON), una representación binaria de JSON optimizada para almacenar y recorrer los datos. Sin embargo, al utilizar los drivers de MongoDB para cada lenguaje, se trabaja directamente con las estructuras de dicho lenguaje [15].

Por tanto, al utilizar un *framework* de desarrollo web basado en JavaScript, el uso de MongoDB tendría la ventaja de reducir el problema ya comentado de la adaptación entre el código y la capa de datos: la información se recibiría de la BD y se manipularía como objetos JavaScript.

Como se ha comentado, Node.js permitió el desarrollo *full-stack* utilizando un único lenguaje tanto para el *frontend* como el *backend*. Con el controlador oficial de MongoDB para Node.js, se pueden manipular los datos directamente como objetos en JavaScript. Esto convierte a MongoDB en un candidato perfecto para ser el SGBD de los *frameworks* de desarrollo web con JavaScript. De hecho, como se verá en el punto siguiente, MongoDB forma parte de las pilas tecnológicas JavaScript modernas.

Respecto a la seguridad, MongoDB ofrece a través de su sitio web varios documentos técnicos que se analizarán, como la guía de Arquitectura de seguridad MongoDB [16].

2.3. Pilas tecnológicas [10]

Una pila tecnológica es una combinación de lenguajes de programación, *frameworks*, bases de datos y otras herramientas o programas que se utilizan para construir aplicaciones web. Las pilas tecnológicas se han convertido en esenciales para construir aplicaciones web escalables y fáciles de mantener.

Un desarrollador o desarrolladora habitualmente se especializa en el *frontend* o el *backend*. Las pilas del desarrollo *frontend* pueden incluir *frameworks*, bibliotecas, gestores de paquetes, sistemas de *build*, herramientas de pruebas, sistemas de control de versiones, herramientas de caché o software de despliegue.

En el desarrollo *backend* se utilizan herramientas de creación de contenedores, APIs, bases de datos, motores de búsqueda, mecanismos de caché y herramientas DevOps. Con el desarrollo *full-stack*, todos los aspectos del desarrollo están cubiertos por una misma pila tecnológica.

Si se utiliza JavaScript como lenguaje tanto en el *frontend* como el *backend*, se habla entonces de pila tecnológica para desarrollo web *full-stack* con JavaScript. Se trata de una combinación de *framework* para *frontend*, *framework* para *backend*, un entorno de ejecución, como Node.js (que permite usar JavaScript en el *frontend* y el *backend*) y la base de datos.

MEAN, MERN y MEVN [10]

Las principales pilas tecnológicas que cubren el desarrollo *full-stack* en JavaScript son MEAN, MERN y MEVN, siendo las dos primeras las más populares. Todas tienen en común MongoDB (M) como sistema gestor de BD NoSQL, Express.js (E) como *framework* web de *backend* y Node.js (N) como entorno de ejecución de JavaScript en el servidor. Lo que las diferencia es el *framework* de *frontend*, que puede ser Angular (A), React (R) o Vue (V) y es lo que da lugar a las tres variantes.

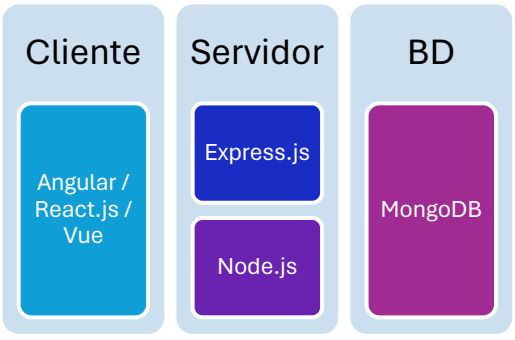


Ilustración 6: Pila tecnológica MExN
Fuente: adaptación de [10] (FIGURA 2.71)

La pila MEAN se utiliza en grandes compañías como Google y PayPal. MERN se emplea también en grandes empresas como Meta (Facebook, Instagram) y Netflix. MEVN tiene uso en compañías más pequeñas, como GitLab.

Existen otras pilas tecnológicas no basadas en JavaScript, tales como LAMP (Linux, Apache, MySQL, PHP) o LEMP (Linux, Nginx, MySQL, PHP). Tienen el inconveniente de usar diferentes lenguajes para el *frontend* y el *backend*. Como consecuencia de ello, no se puede reutilizar código y el equipo de desarrollo debe aprender un lenguaje adicional.

El trabajo no se centrará en estos y sí en la pila MExN, dada la prominencia de JavaScript en el desarrollo web moderno, comentada en puntos anteriores.

3. Herramientas de escaneo de vulnerabilidades

Los escáneres de vulnerabilidades de aplicaciones web son herramientas automatizadas que analizan las aplicaciones web, a menudo desde fuera, en busca de vulnerabilidades de seguridad tipo *cross-site scripting*, inyección SQL, inyección de comandos, salto de directorio (*path traversal*), configuraciones del servidor incorrectas, etc. [17].

A las herramientas de esta categoría se les conoce como DAST (*Dynamic Application Security Testing*) o herramientas de pruebas dinámicas de seguridad de aplicaciones. Existen un buen número de estas herramientas, tanto comerciales como *open source*, cada una de ellas con sus fortalezas y debilidades [17].

En el momento de redactar este trabajo, OWASP lista 103 aplicaciones de este tipo en su página sobre herramientas de escaneo de vulnerabilidades [17]. Dada la imposibilidad de examinar todas las herramientas por los límites de espacio y tiempo del trabajo, se ha acudido a un estudio comparativo para dirigir la selección de las herramientas que se emplearán.

En el artículo de 2022 "*An empirical comparison of commercial and open-source web vulnerability scanners*" [18], Amankwah *et al*. presentan una comparativa empírica de varias herramientas de análisis de vulnerabilidades web, tanto comerciales (Burp Suite Professional, Qualys WAS, Fortify WebInspect) como *open source* (OWASP ZAP, Arachni, Wapiti3). Para elegir estas herramientas, el estudio se basa en aquellas que más aparecían en trabajos comparativos anteriores, así como las respuestas a encuestas de los profesionales de ciberseguridad [18] (P. 6).

El estudio toma como base varios *frameworks* de evaluación de este tipo de aplicaciones, pero añade más parámetros que permitan validar también la usabilidad y el rendimiento de las herramientas. Así, propone 19 métricas en cinco categorías, cada una de ellas con un rango de puntuaciones [18] (PP. 6-11). A continuación, pasa a evaluar cada herramienta, atacando el OWASP Benchmark Project[4] con las herramientas y puntuándolas de acuerdo con el modelo de evaluación propuesto.

[4] Es una aplicación web Java *open source* que contiene miles de casos de prueba explotables, destinado a evaluar la precisión, cobertura y velocidad de las herramientas de escaneo de vulnerabilidades. Fuente: https://owasp.org/www-project-benchmark/.

Como resultado del estudio, una herramienta de cada categoría recibe la máxima puntuación: Burp Suite en el grupo de herramientas comerciales[5] y ZAP de OWASP en la categoría de herramientas *open source*[6] [18] (P. 14). En cualquier caso, se concluye del estudio que cada herramienta tiene sus fortalezas y debilidades y que el equipo de seguridad debe servirse de las fortalezas de cada herramienta por separado [18] (PP. 17-18).

Parece interesante fijarse en la métrica "cobertura de las vulnerabilidades OWASP Top 10", ya que habitualmente se emplea esta lista de las diez principales vulnerabilidades en aplicaciones web para los análisis de seguridad. En las herramientas comerciales, Burp Suite cubre el 70% de vulnerabilidades [18] (P.13). En el grupo *open source*, ZAP de OWASP sólo cubre el 1%, frente a Wapiti3 con el 40% [18] (P.16). Este resultado puede parecer sorprendente, pero el estudio indica que se necesita añadir ciertas extensiones a ZAP[7] para mejorar la detección de las vulnerabilidades OWASP Top 10 [18] (P.15).

Hecha esta puntualización, para los propósitos de este trabajo utilizaremos las dos herramientas que han recibido más puntuación, Burp Suite y OWASP ZAP. De Burp Suite se utilizará la versión Community, que es gratuita. Hay que tener en cuenta que el estudio mencionado ha analizado la versión comercial y los resultados no serán replicables en la versión Community, que tendrá funcionalidad reducida en comparación. Pero la elección encaja con los objetivos expuestos de ofrecer a una pyme sin presupuesto de seguridad herramientas gratuitas como parte de la metodología de análisis de seguridad.

3.1. OWASP ZAP (Zed Attack Proxy) [19]

ZAP de OWASP es una herramienta de *pentesting* gratuita y *open source*, diseñada específicamente para la prueba de aplicaciones web.

Técnicamente, ZAP es un *proxy man-in-the-middle*. Es decir, se coloca entre el navegador web y la aplicación, de forma que puede interceptar los mensajes que se intercambian entre ambos, modificar los contenidos si es necesario y dirigir dichos mensajes al destino.

ZAP se puede ejecutar como aplicación independiente o como proceso *daemon*. También tiene funcionalidad en línea de comandos y una API. Dispone de versiones para Windows, Linux y macOS, además de imágenes para Docker. Además de las características que incluye por defecto, la funcionalidad de la aplicación se puede ampliar con el uso de extensiones, disponibles en el ZAP Marketplace y accesibles desde el propio cliente ZAP.

[5] Burp Suite Professional: 38; Qualsys WAS: 36; Fortify WebInspect: 32.
[6] OWASP ZAP: 32; Wapitit3: 20; Arachni: 15.
[7] Las recomendaciones sobre componentes manuales y automatizados para dicho análisis se recogen en la página: *ZAPping the OWASP Top 10 (2021)* https://www.zaproxy.org/docs/guides/zapping-the-top-10-2021/.

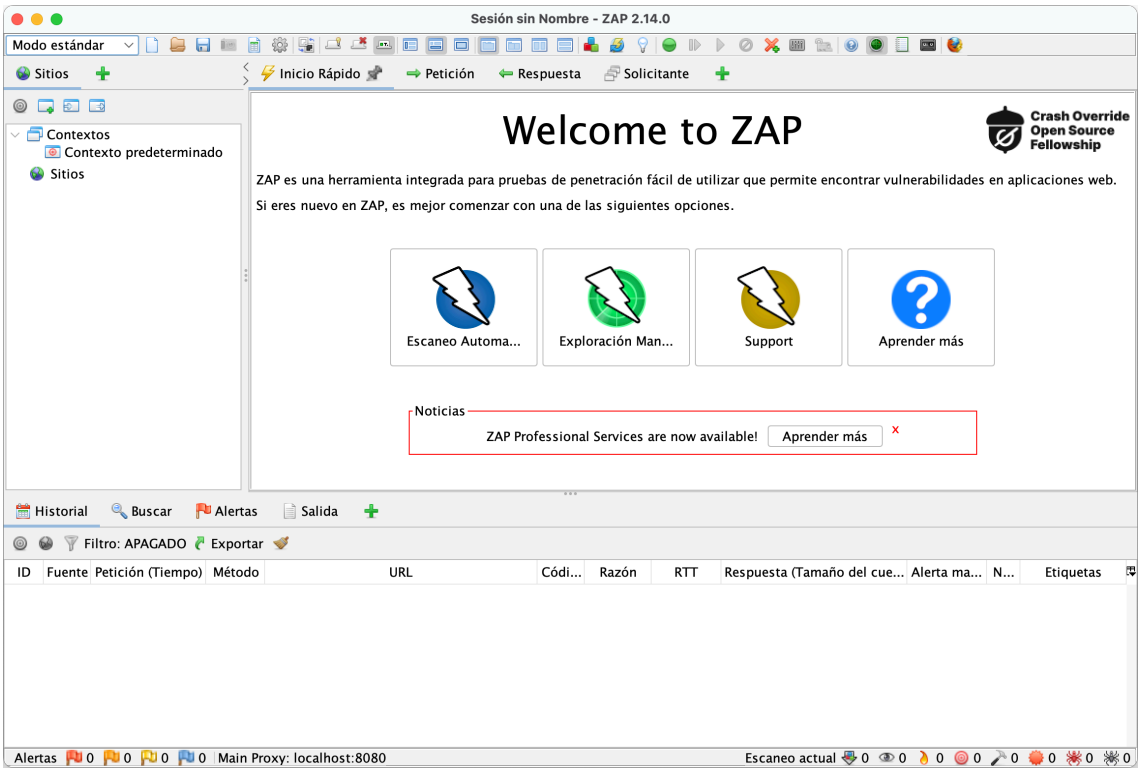


Ilustración 7: Cliente ZAP 2.14.0
Fuente: elaboración propia

ZAP dispone de una opción de escaneo automático, donde se introduce la URL de la aplicación a probar y ZAP la recorrerá con su *spider* para obtener la estructura de la aplicación. Este escaneo es pasivo, ya que no cambia las respuestas. Después atacará las páginas que ha descubierto mediante un escáner activo, que intenta descubrir otras vulnerabilidades utilizando ataques conocidos contra cada página.

Las limitaciones del escaneo pasivo y el ataque automatizado descritos son que 1) ZAP no descubrirá las páginas que requieran autenticación, mientras que no se configure esta y 2) no se tiene control sobre la secuencia de los ataques que se llevan a cabo de forma automatizada.

Por eso, tras el ataque automatizado es conveniente realizar una exploración manual. Con ella, se lanza un navegador web y se prueba cada página de la aplicación. Mientras, ZAP guarda cada petición hecha y respuesta recibida, mostrando alertas de vulnerabilidades potenciales que encuentre durante la exploración.

3.2. Burp Suite

Es una herramienta comercial de escaneo de vulnerabilidades web, desarrollada por la empresa PortSwigger, que dispone de varias versiones. Se indican a continuación las funcionalidades de Burp Suite Community Edition [20,21].

Proxy: al igual que ZAP, Burp funciona como servidor *proxy* web entre el navegador y la aplicación a analizar. Permite interceptar, inspeccionar y modificar el tráfico que pasa en ambas direcciones, incluyendo HTTPS. Registra un histórico del tráfico que ha pasado por el proxy, tanto para HTTP, como para mensajes WebSocket.

Repeater: permite enviar un mensaje HTTP o WebSocket varias veces. Por ejemplo, una petición variando los valores de un parámetro o una serie de peticiones HTTP en una secuencia específica.

Decoder: permite transformar datos utilizando formatos de codificación y decodificación comunes, como la codificación de URL, Base64, etc.

Sequencer: permite un análisis cualitativo de la aleatoriedad de una muestra de elementos. Está pensado para elementos tipo claves de sesión, tokens anti-CSRF o tokens de restablecimiento de contraseña.

Comparer: permite comparar dos elementos de datos cualquiera. Se utiliza para identificar las diferencias que puede haber entre varias peticiones o respuestas. Por ejemplo, peticiones fallidas de inicio de sesión, peticiones similares que originan comportamientos diferentes de la aplicación, etc.

Intruder: se utiliza para automatizar ataques personalizados contra las aplicaciones web. Permite configurar ataques que envían la misma petición HTTP varias veces, insertando cada vez distintos *payloads* en las posiciones predeterminadas.

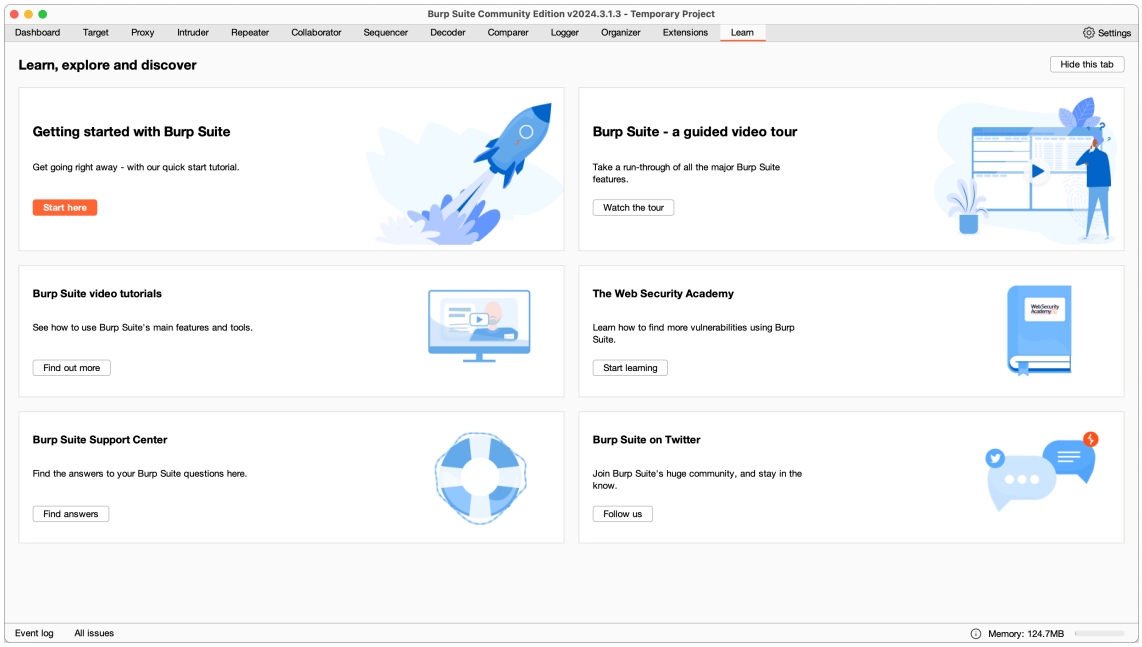


Ilustración 8: Cliente Burp Suite Community Edition v2024.3.1.4.
Fuente: elaboración propia.

Ciertas características de Burp Suite que están disponibles en la versión Professional pero no la Community, tienen equivalentes en ZAP con funcionalidades similares. Estas son las siguientes [22]:

Collaborator: es una característica hace que la aplicación que se prueba interactúe con un servidor externo para buscar ciertas vulnerabilidades que, de otra forma, no generan una respuesta diferente a un ataque de prueba. El equivalente de ZAP es la extensión "Out-of-band Application Security Testing Support".

Escaneo en vivo: son escaneos que se ejecutan como proceso de fondo mientras se explora manualmente la aplicación objetivo. Equivalen en ZAP al modo ATTACK, que escanea activamente los nodos tras ser descubiertos.

Ficheros de proyecto: la versión Community de Burp Suite sólo permite crear proyectos en memoria, sin poder persistirlos a disco. Esto sí es posible en ZAP.

Escáner: es el escáner dinámico de vulnerabilidades de aplicaciones web (DAST) de Burp y sólo está disponible en las versiones Enterprise y Professional. Corresponde en ZAP al escaneo activo, que busca vulnerabilidades potenciales utilizando ataques conocidos contra los objetivos.

3.3. Limitaciones de las herramientas de análisis de vulnerabilidades

La mayoría de los equipos de desarrollo no prueba el software hasta que este ha sido ya creado y está en la fase de despliegue de su ciclo de vida. Esta es una práctica muy ineficiente y costosa. Una de las mejores formas de evitar que aparezcan los fallos de seguridad es integrar la seguridad en cada fase del ciclo de vida del desarrollo del software (SDLC: *Software Development Life Cycle*) [23] (P. 12).

Esta integración supone un enfoque holístico de la seguridad de la aplicación que aprovecha los procedimientos ya existentes en las empresas. Cada fase del ciclo tiene consideraciones de seguridad que deberían formar parte de los procesos existentes, para garantizar un programa de seguridad completo y rentable en coste [23] (P. 15).

Hay que tener en cuenta que las herramientas de escaneo de vulnerabilidades comentadas hasta ahora pueden encontrar problemas genéricos. Pero no tienen suficiente conocimiento de la aplicación analizada como para detectar otros problemas más serios que se deriven de la lógica de negocio o el diseño específico de la aplicación [23] (P. 6).

Estas herramientas se pueden complementar con la revisión del código fuente. Esta consiste en buscar manualmente problemas de seguridad en el código fuente de la aplicación web. Muchas vulnerabilidades de seguridad graves no se pueden detectar con ninguna otra forma de prueba [23] (P. 19).

En cualquier caso, cuando el objetivo es encontrar los fallos más graves de la aplicación de manera rápida, las herramientas de análisis de vulnerabilidades pueden resultar muy útiles y formar parte de la estrategia de seguridad [23] (P. 6).

4. OWASP Juice Shop

OWASP mantiene un catálogo de aplicaciones web y móviles vulnerables que los equipos de desarrollo y los de pruebas pueden utilizar para poner a prueba sus habilidades y poder evaluar el rendimiento de herramientas de análisis de aplicaciones web [24] como las mencionadas en el apartado anterior.

Se ha elegido una de estas aplicaciones, OWASP Juice Shop [25] para poner a prueba las herramientas. Los criterios para esta elección han sido:

- La aplicación es un proyecto gestionado por la propia fundación OWASP, que lo cataloga como *flagship project* o proyecto de referencia.
- Utiliza Node.js, Express y Angular como tecnologías principales. Por tanto, se encuadra en lo que hemos definido como aplicación web moderna.
- Es un proyecto gratuito y *open source* con un elevado número de colaboradores y actualizado frecuentemente. En el momento de redactar este trabajo, tiene 103 colaboradores y su último *commit* en GitHub es de hace dos semanas.
- Dispone de abundante referencia, incluyendo la guía oficial "*Pwning OWASP Juice Shop*". Esto la convierte en una aplicación ideal como iniciación en la seguridad web para el equipo de desarrollo de una pyme sin presupuesto de ciberseguridad, como se ha propuesto al principio.

4.1. Arquitectura de la aplicación [26]

La Juice Shop de OWASP es una aplicación web implementada en JavaScript y TypeScript. Utiliza Angular como *framework* de desarrollo de *frontend* para crear una aplicación SPA (*Single-Page Application*). La interfaz de usuario se implementa mediante componentes Angular Material.

El *backend* es una aplicación Express alojada en un servidor Node.js que entrega el código cliente al navegador. También se sirve funcionalidad al cliente a través de una API REST.

Utiliza SQLite como SGBD y Sequelize y finale-rest como capa de abstracción de la BD. También utiliza MarsDB, SGDB NoSQL derivado de MongoDB y compatible con la mayoría de las operaciones de consulta y modificación de Mongo.

Utiliza el protocolo WebSocket para mostrar las notificaciones que muestra cuando se ha descubierto una vulnerabilidad. Y ofrece autenticación mediante OAuth 2.0 para permitir registrarse con una cuenta de Google.

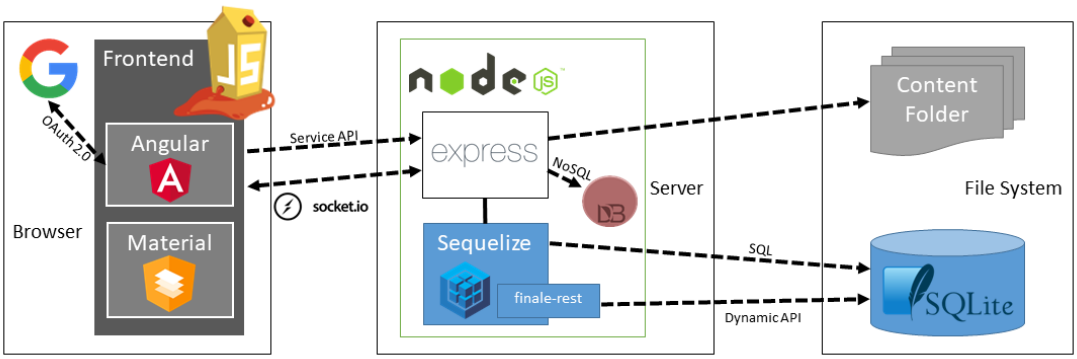


Ilustración 9: Arquitectura de la aplicación web OWASP Juice Shop
Fuente: [26] (ARCHITECTURE OVERVIEW)

Se instala la aplicación web vulnerable en nuestro equipo para hacer las pruebas, siguiendo las indicaciones para su instalación a partir del código fuente[8]. Una vez instalada, accedemos a ella desde la URL http://localhost:3000/:

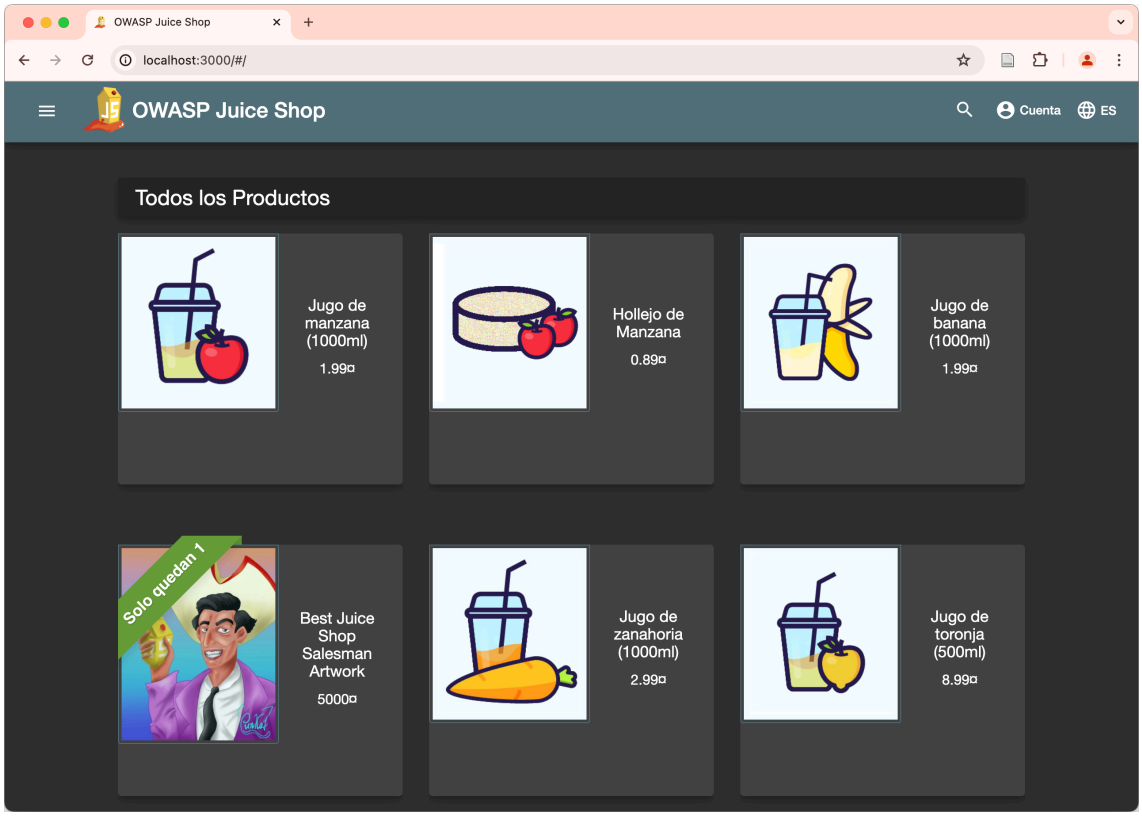


Ilustración 10: Página principal de OWASP Juice Shop
Fuente: elaboración propia

4.2. Escaneo automático con ZAP

La forma más fácil de empezar a utilizar las herramientas de análisis de vulnerabilidades web es con el escaneo automatizado. Este tipo de análisis no está disponible en la versión Community de Burp Suite, así que se utilizará ZAP.

[8] https://github.com/juice-shop/juice-shop#from-sources

Desde ZAP, elegimos la opción "Escaneo Automático" en la pantalla de bienvenida [27] e introducimos la URL atacar. En este caso, http://localhost:3000/, que es donde hemos instalado la aplicación Juice Shop:

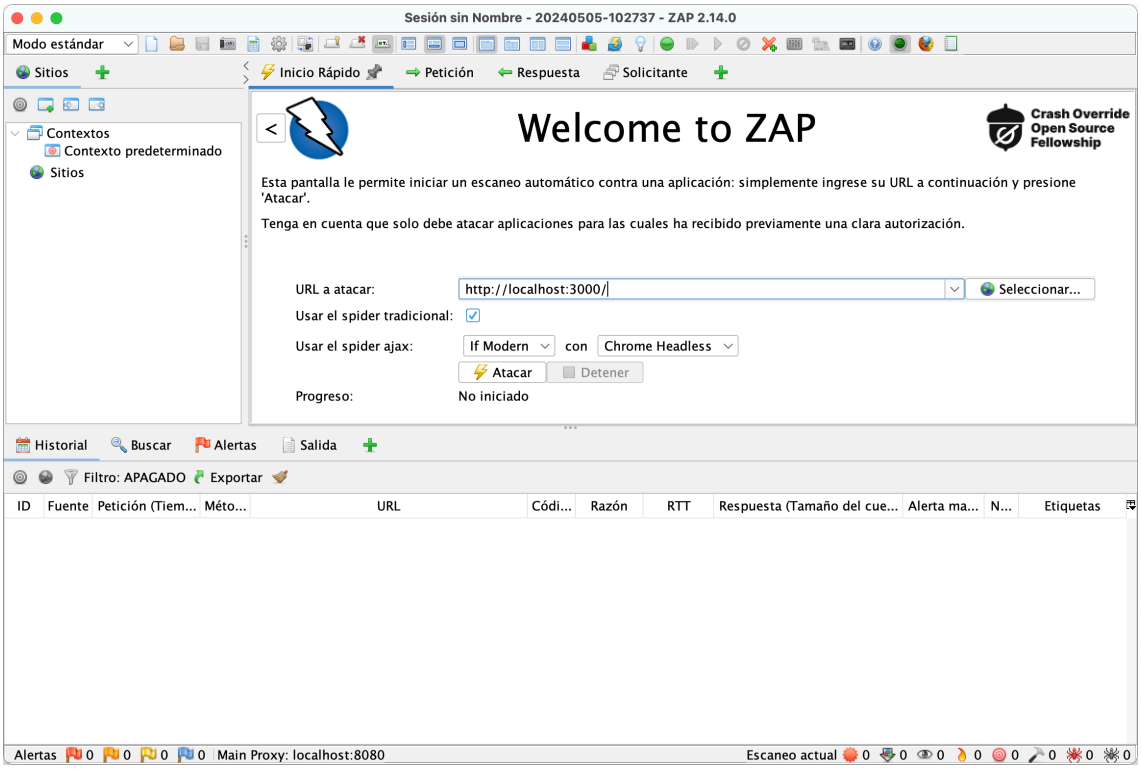


Ilustración 11: Escaneo automático de ZAP contra OWASP Juice Shop
Fuente: elaboración propia

El *spider* es el proceso que ZAP utilizará para recorrer las diferentes páginas que forman la aplicación web. El *spider* tradicional examina el HTML de las respuestas en busca de los enlaces. Esto no resulta efectivo para aplicaciones web como la Juice Shop, que es una aplicación tipo SPA cuyo contenido se genera dinámicamente con JavaScript.

Esta manipulación dinámica se puede observar mientras se interactúa con la aplicación desde las herramientas de desarrollo de los navegadores (accesibles, por ejemplo, desde Chrome, con la tecla F12). Para este tipo de aplicaciones es más efectivo utilizar el *spider* AJAX, que invoca navegadores que siguen los enlaces que se han generado.

En este ejemplo se ha utilizado el *spider* tradicional y se utilizará también el *spider* AJAX si se detecta una aplicación moderna. Pulsamos el botón "Atacar" para comenzar y ZAP recorrerá la aplicación web, construyendo un mapa de las páginas de esta y de los recursos utilizados para mostrar esas páginas. Registra las peticiones y respuestas de cada página y crea alertas si encuentra algo potencialmente peligroso en alguna de ellas.

La imagen siguiente muestra el resultado del escaneo automatizado:

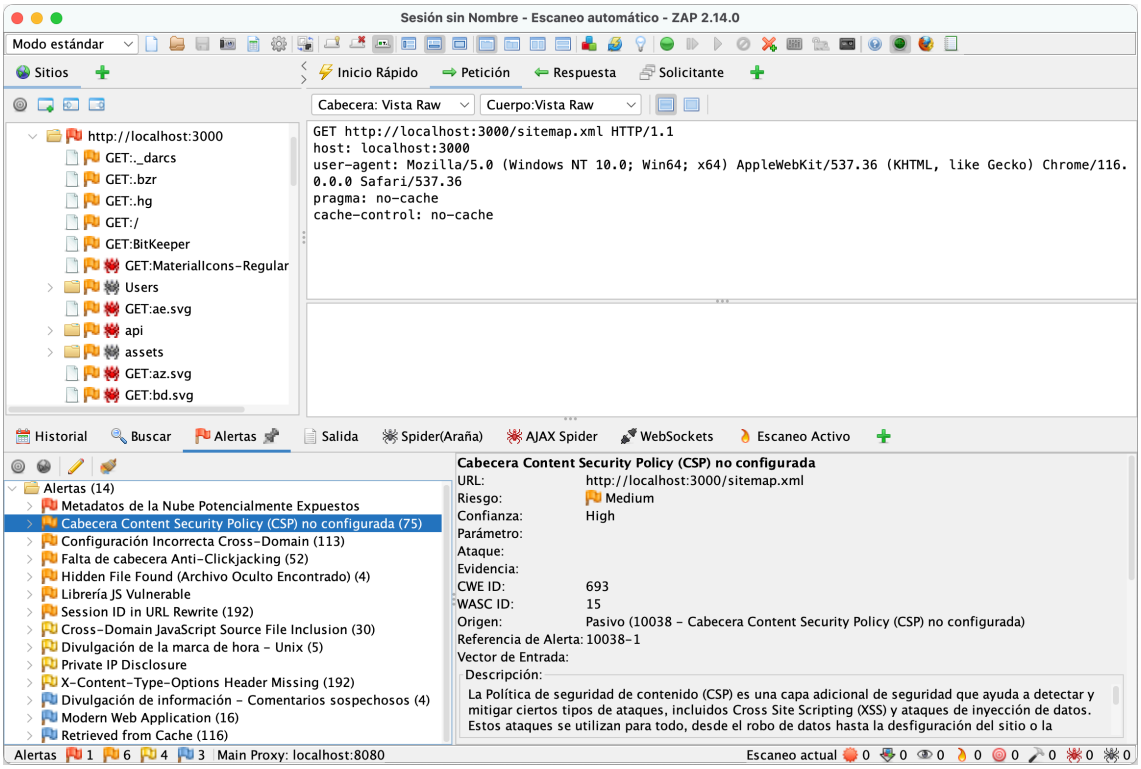


Ilustración 12: Resultado del análisis automático de Juice Shop con ZAP
Fuente: elaboración propia

En la sección "Sitios" se muestra el árbol de la aplicación web que los *spiders* han formado tras rastrear el sitio. En la mitad inferior de ZAP existen varias pestañas, destacando "Alertas", donde se muestra una lista de las incidencias encontradas clasificadas por varias categorías de riesgo.

En la captura se muestra la alerta "Cabecera Content Security Policy (CSP) no configurada" y se indica entre paréntesis el número de instancias detectadas de esta vulnerabilidad (75). Al seleccionar esta alerta, se nos presenta en la parte superior derecha la petición y la respuesta que han generado esta alerta.

A la hora de analizar las vulnerabilidades, además de tener en cuenta el nivel del riesgo de la incidencia, es importante la clasificación de confianza que ZAP le otorga. Por ejemplo, la alerta de nivel alto "Metadatos de la Nube Potencialmente Expuestos" tiene un nivel de confianza bajo. Mientras que la incidencia seleccionada tiene un nivel de confianza alto (efectivamente, podríamos comprobar en la pestaña "Respuesta" que no se incluye una cabecera CSP).

La alerta se clasifica según su código CWE (Common Weakness Enumeration) y se ofrece más información sobre ella. En el ejemplo, se muestra el objetivo de la cabecera CSP y qué tipos de ataques ayuda a mitigar.

Otra información adicional (que no aparece en la captura) es una propuesta de solución, enlaces a páginas de referencia con más información y etiquetas de la alerta. Estas últimas pueden contener, por ejemplo, qué vulnerabilidad del

OWASP Top 10 se ha detectado. En este ejemplo, corresponde a "A05:2021 - Configuración de Seguridad Incorrecta" de la edición de 2021 de la lista.

El escaneo automatizado es una buena forma de comenzar un análisis de vulnerabilidades web. Por ejemplo, para la alerta anterior, una vez que el equipo de desarrollo se informa sobre la cabecera CSP y cómo incluirla, se puede solucionar fácilmente esta incidencia.

Otro ejemplo es la alerta "Librería JS Vulnerable", donde se detecta la versión 2.2.4 de jQuery, que se considera vulnerable. En este caso, el equipo puede plantearse migrar a la versión más reciente que no sea vulnerable, tras hacer las pruebas que aseguren que no se rompe la funcionalidad actual.

4.3. Escaneo manual con ZAP [27]

Pero, como se ha comentado anteriormente, el análisis automático de vulnerabilidades tiene limitaciones. En este ejemplo concreto, la mayor parte de la funcionalidad de la Juice Shop requiere autenticación. Aunque se puede navegar por los productos, no es posible realizar ninguna compra si no nos hemos registrado.

La funcionalidad de los *spiders* es limitada y, por ejemplo, en un formulario de inicio de sesión sólo introducirán información aleatoria que no conseguirá un inicio de sesión. Mientras que el equipo que realiza las pruebas podrá iniciar sesión con una cuenta válida o, al menos, registrarse como usuario en la tienda.

Para realizar un análisis manual en ZAP, se elige la opción "Exploración Manual" en la pestaña de "Inicio Rápido" de la aplicación:

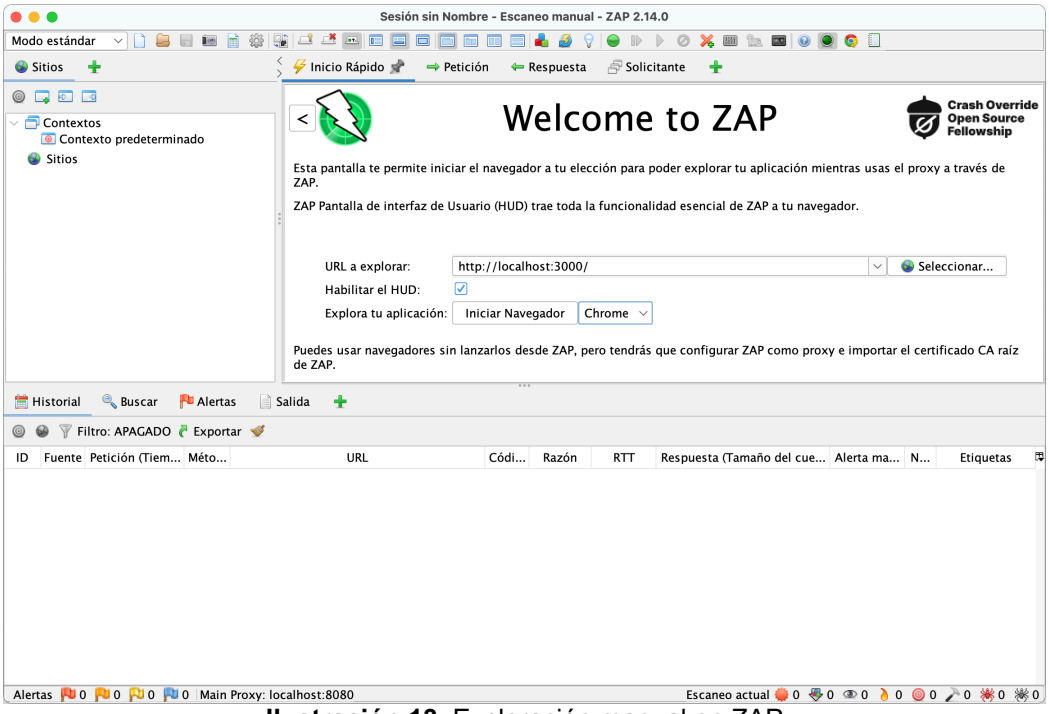


Ilustración 13: Exploración manual en ZAP
Fuente: elaboración propia

Nuevamente, se introduce la URL de la aplicación a analizar. La opción "Habilitar el HUD" sirve para mostrar información relevante de ZAP en la propia aplicación web mientras se navega. Se selecciona un navegador, en este ejemplo Chrome y, tras pulsar el botón "Iniciar Navegador", se lanza la aplicación web en dicho navegador.

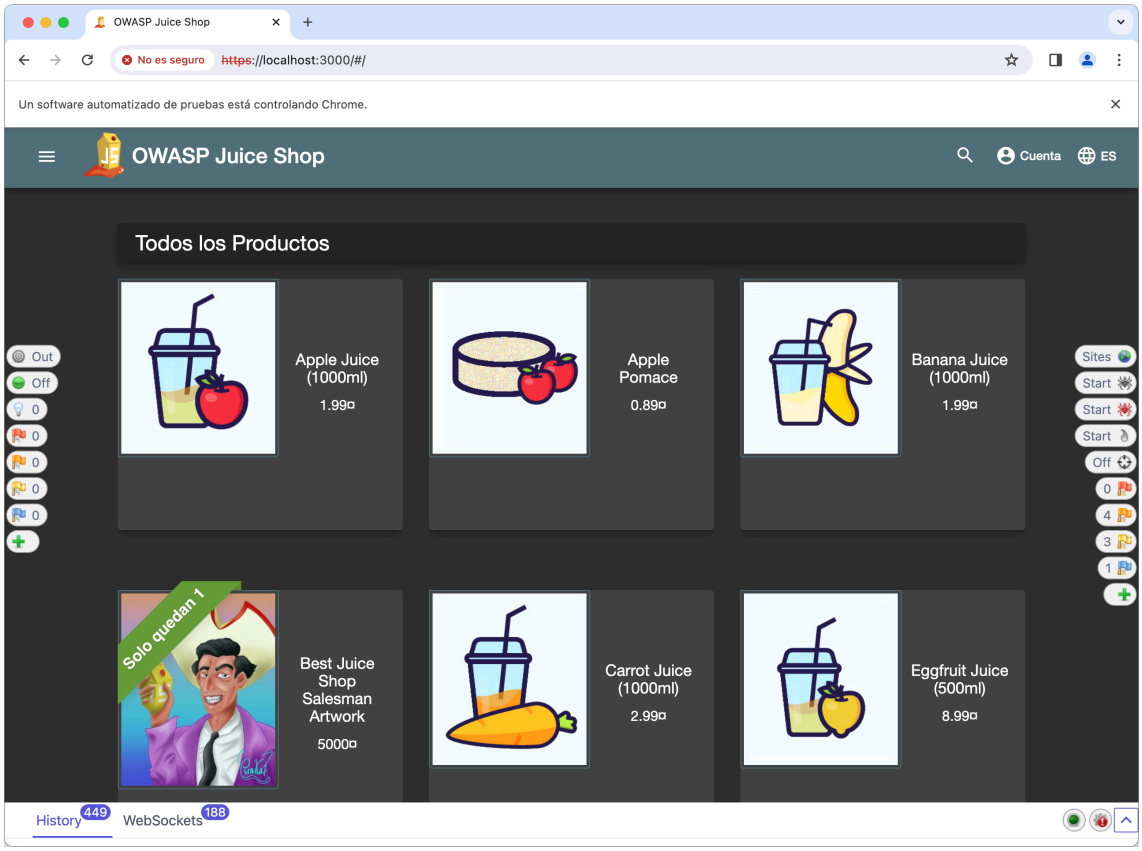


Ilustración 14: Juice Shop en exploración manual desde Chrome
Fuente: elaboración propia

Como se observa en la pantalla, Chrome indica que está siendo controlado por una aplicación de automatización de pruebas. Los iconos a ambos lados corresponden a la funcionalidad HUD y mostrarán en la propia aplicación información sobre las vulnerabilidades que se vayan encontrando y otras opciones.

A continuación, se maneja la aplicación web, registrando un nuevo usuario e iniciando sesión con este. Después se utiliza la compra de productos, el registro de direcciones de envío y métodos de pago, etc.

Mientras se va utilizando la aplicación, ZAP ha ido registrando las páginas visitadas y construyendo el árbol del sitio. Como en el análisis automático, ha identificado ciertas vulnerabilidades y las ha clasificado por severidad. Podemos ver el resultado en la siguiente pantalla:

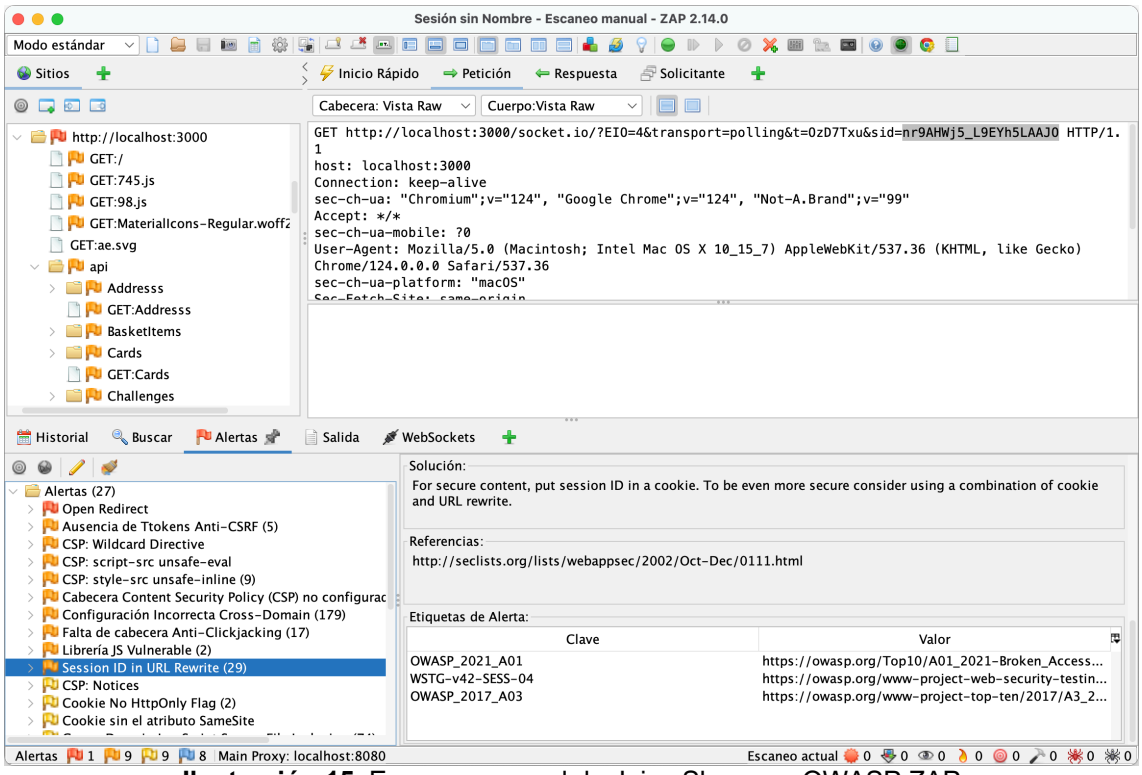


Ilustración 15: Escaneo manual de Juice Shop con OWASP ZAP
Fuente: elaboración propia

En la figura se muestra la alerta de la vulnerabilidad "Session ID in URL Rewrite", detectada con un nivel de confianza alto. Se observa, resaltado en la URL de la petición, lo que parece un código de sesión. Se sugiere como solución el uso de cookies y se etiqueta con las vulnerabilidades del OWASP Top 10 de 2021 y 2017, de las que se puede consultar más información. También se hace referencia al apartado relevante de la guía de pruebas de seguridad web de OWASP (WSTG: Web Security Testing Guide).

4.4. Escaneo manual con Burp Suite [28]

Al igual que ZAP, la edición Community de Burp Suite dispone de la función de interceptar el tráfico HTTP haciendo que la aplicación funcione como proxy. Para ello, nos situamos en la pestaña "Proxy → Intercept" de la aplicación:

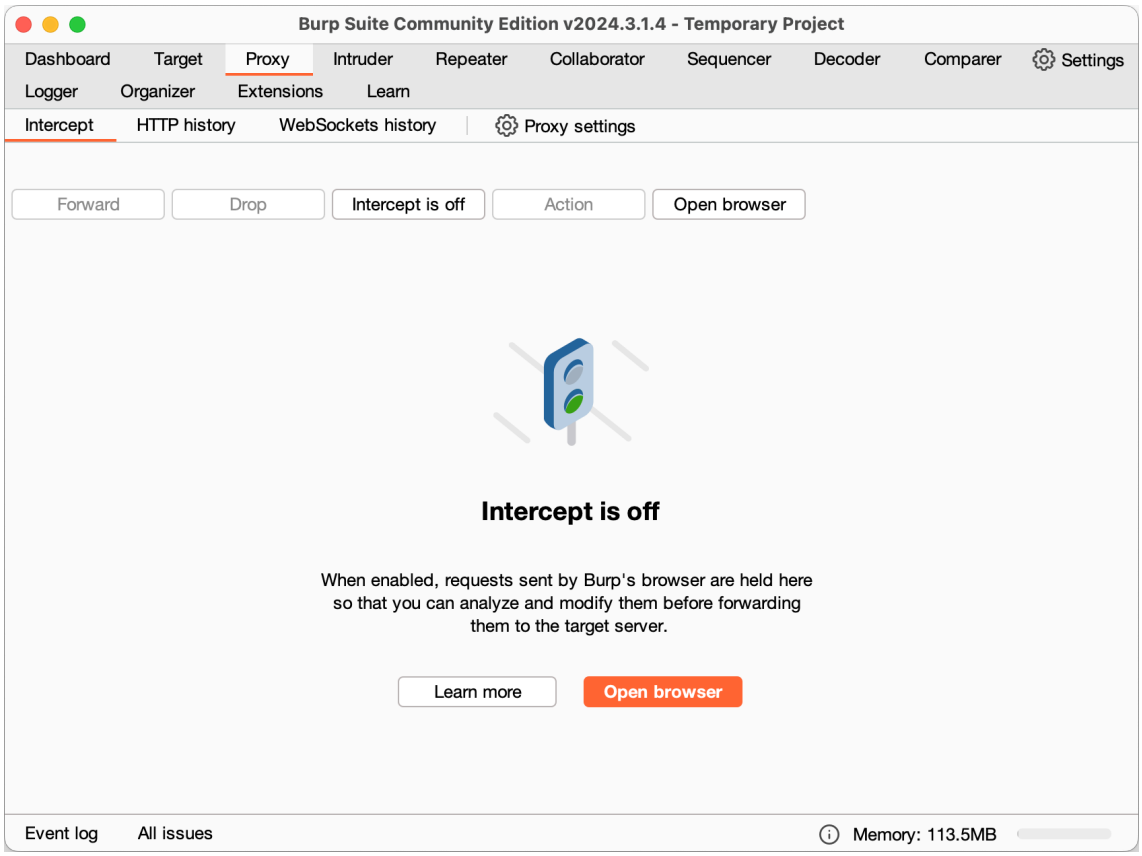


Ilustración 16: Modo proxy en Burp Suite
Fuente: elaboración propia

Pulsamos el botón "Open browser" y se abre un navegador, donde se introduce la URL de la aplicación a analizar. Se maneja la aplicación web y, una vez terminado, Burp Suite habrá capturado todas las URLs con las que hemos interactuado. Estas se pueden ver desde la opción "Proxy → HTTP history":

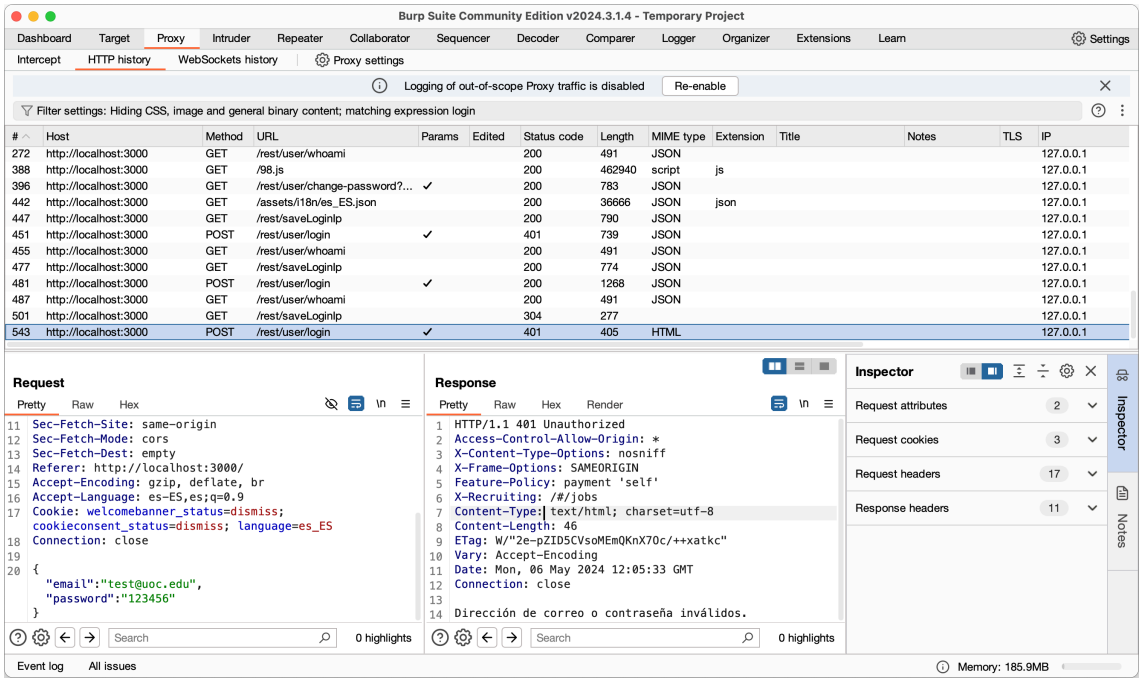


Ilustración 17: Resultado del escaneo manual en Burp Suite
Fuente: elaboración propia

En la pantalla se observa una de las peticiones capturadas, que corresponde al inicio de sesión del usuario. Sin embargo, no aparece ninguna información sobre vulnerabilidades que se hayan encontrado. Esto se debe a que la versión Community de Burp Suite no permite el escaneo automatizado de vulnerabilidades. En su lugar, se debe hacer un análisis manual.

Por ejemplo, se podría enviar la petición anterior a la herramienta Repeater. Con ella, se podría cambiar los valores de los parámetros de cada petición, por ejemplo, para intentar un ataque de fuerza bruta contra el inicio de sesión.

5. OWASP Top Ten

El OWASP Top 10 es un documento de concienciación sobre seguridad de aplicaciones web que representa el consenso sobre los riesgos de seguridad más críticos de dichas aplicaciones. Las empresas pueden adoptar este documento como punto de partida para asegurarse de que sus aplicaciones web minimizan estos riesgos y para cambiar la cultura de desarrollo de software hacia una que produzca código más seguro [29].

En el momento de redactar este trabajo, la guía de 2024 se encuentra en fase de desarrollo, siendo la última versión publicada la versión de 2021, sobre la que se trabajará. Para cada una de las diez vulnerabilidades, la guía presenta una introducción, una descripción, guías sobre cómo prevenir la vulnerabilidad, ejemplos de escenarios de ataque y una lista de CWEs relacionadas.

En este apartado se analizará cada vulnerabilidad del OWASP Top 10 [30] llevando a cabo, a modo de ejemplo, alguna de las formas de explotar cada una . Para ello se utilizarán las herramientas descritas anteriormente contra la aplicación web vulnerable OWASP Juice Shop.

En ciertos apartados se incluirá además información específica sobre seguridad de los *frameworks* web que se han tratado, que se deriva de las distintas guías de mejores prácticas de seguridad de los fabricantes.

5.1. A01:2021 – Pérdida de Control de Acceso

El control de acceso es parte de la política de seguridad y supone que ningún usuario pueda operar más allá de los permisos que se le han dado. Un fallo en el control de acceso puede conducir a la revelación no autorizada de información, la modificación o destrucción de datos o que el usuario pueda realizar una función que no le corresponde por sus permisos.

Algunas vulnerabilidades frecuentes relacionadas con el control de acceso son:

- La violación del principio del menor privilegio, donde el acceso, que sólo debería estar disponible para ciertos roles o usuarios, está disponible a cualquiera.
- Saltarse los controles de acceso modificando la URL (manipulando los parámetros, por ejemplo), el estado interno de la aplicación, el HTML de la página o modificando las peticiones a las APIs.
- Permitir ver o editar la cuenta de otro usuario, utilizando su identificador único.
- Acceder a una API que no tiene control de acceso en POST, PUT o DELETE.
- Elevación de privilegio: funcionar como un usuario sin haber iniciado sesión o como administrador siendo un usuario que no lo es.
- Manipular los metadatos, por ejemplo, reutilizando un token de control de acceso JWT (JSON Web Token) o una cookie o campo oculto para elevar los privilegios.
- Una mala configuración de CORS que permite el acceso a la API desde orígenes no autorizados o que no son de confianza.
- Forzar la navegación hacia páginas autenticadas como usuario no autenticado o como usuario estándar.

Algunas formas de prevenir estas vulnerabilidades:

- Salvo los recursos públicos, denegar el acceso por defecto.
- Implementar mecanismos de control de acceso una sola ver y reutilizarlos en toda la aplicación, minimizando el uso de CORS.
- El control de acceso debe hacerse cumplir a nivel de registro en lugar de permitir que el usuario pueda crear, leer, actualizar o borrar cualquier registro.
- Deshabilitar el listado de directorios del servidor web y asegurarse de que no hay ficheros de metadatos (ej.: .git) en las raíces web.
- Registrar los fallos de control de acceso y alertar a los administradores cuando sea necesario (por ejemplo, tras fallos repetidos).
- Limitar el número de peticiones que se pueden hacer a una API o un controlador de acceso para minimizar el daño del uso de herramientas automatizadas.
- Los identificadores del estado de la sesión deben ser invalidados en el servidor tras el cierre de sesión. Los tokens JWT sin estado deberían tener una duración breve para reducir la ventana de oportunidad de un atacante.

Como ejemplo de explotación, la Juice Shop permite a los usuarios escribir reseñas sobre los productos:

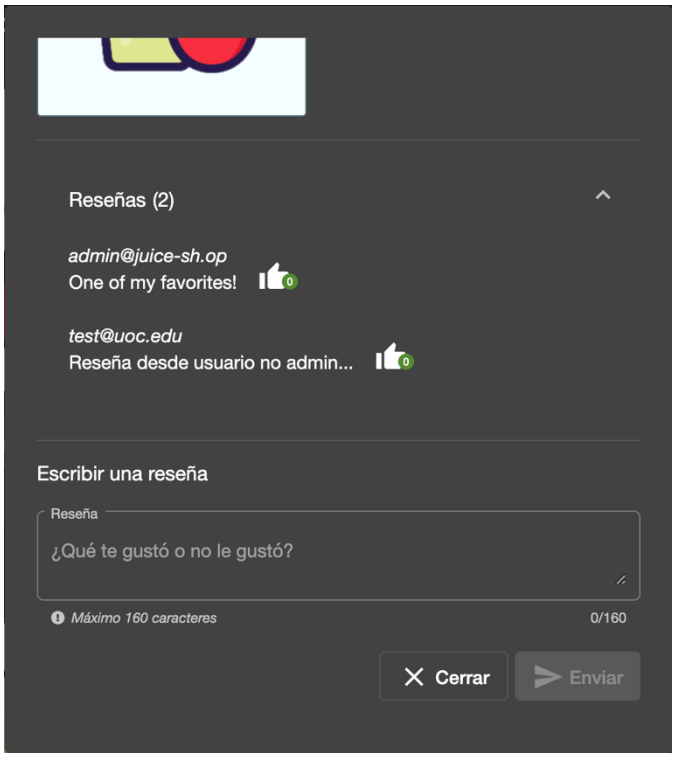


Ilustración 18: Reseña de un producto en la Juice Shop
Fuente: elaboración propia

En la imagen, el usuario no administrador (test@uoc.edu) ha publicado una reseña, que aparece junto a la del usuario administrador (admin@juice-sh.op). Se ha capturado esta petición desde ZAP y la podemos modificar desde la utilidad *Requester*:

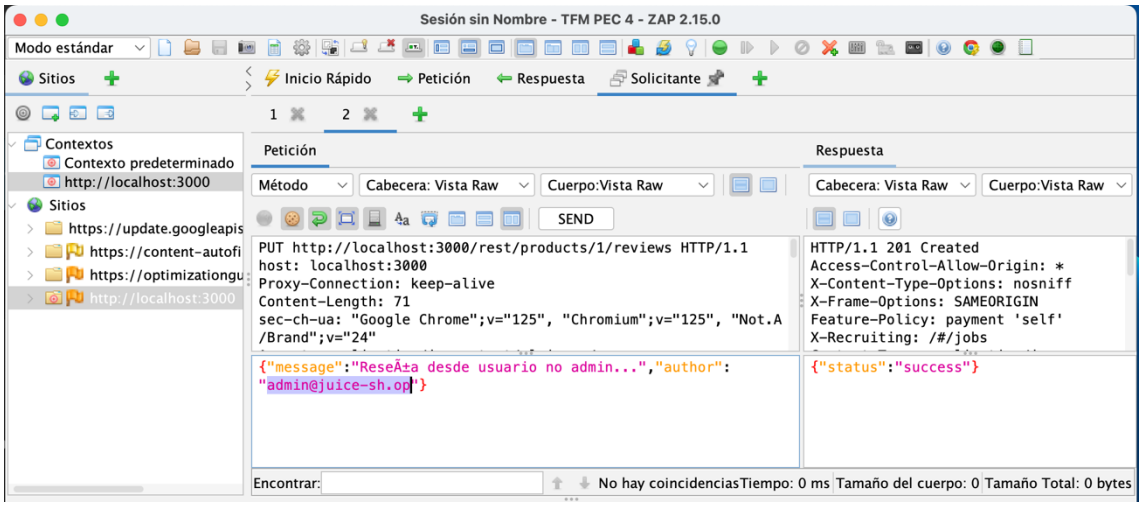


Ilustración 19: Captura de la petición de reseña en la utilidad *Requester* de ZAP
Fuente: elaboración propia

Se trata de una petición `PUT` a la API REST de la tienda y en ella hemos cambiado el valor de la propiedad `author` desde el email original del usuario no administrador al administrador. Podemos volver a enviar la petición con el botón "SEND":

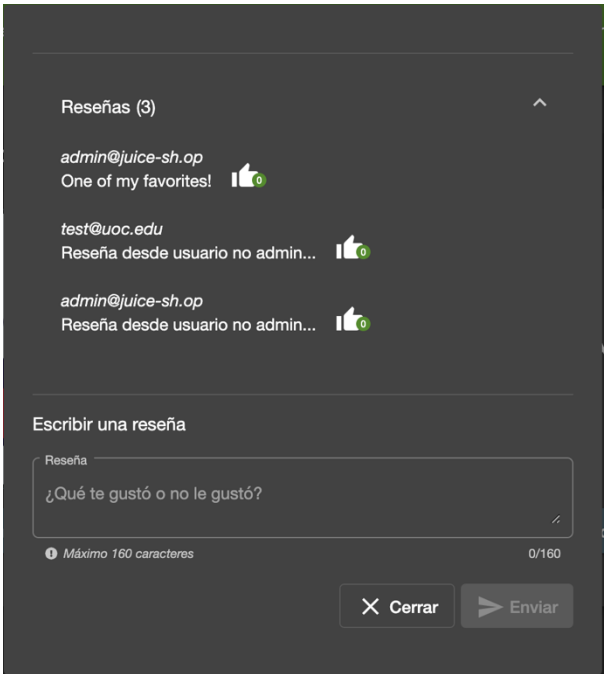


Ilustración 20: Publicación de reseña en nombre de otro usuario.
Fuente: elaboración propia.

Se puede comprobar que el usuario no administrador ha conseguido publicar una reseña como si la hubiera hecho el usuario administrador, al fallar el control de acceso en la llamada a la API.

Exposición de información sensible en Node.js

La CWE-552 (Files or Directories Accessible to External Parties) se menciona en la guía de seguridad de Node.js [7] y se incluye también en la lista de CWEs de este primer apartado del Top 10.

Durante el proceso de publicación de paquetes, los ficheros y carpetas de la aplicación se envían al registro de `npm`, lo que permite resolver paquetes por nombre y versión. Para evitar exponer potencialmente información confidencial, se puede establecer una lista de bloqueo en ficheros como `.npmignore` y `.gitignore` o una lista de ubicaciones permitidas en `package.json`.

5.2. A02:2021 – Fallos Criptográficos

Se trata de fallos relacionados con la criptografía que conducen a la exposición de datos sensibles.

Se deben determinar las necesidades de protección de los datos en tránsito y en reposo. Hay datos que requieren una protección adicional, especialmente si son

datos de carácter personal protegidos por el RGPD u otros, como el PCI Data Security Standard (PCI DSS). Para estos datos, se debe analizar:

- Si los datos se transmiten sin cifrar, con protocolos como HTTP, SMTP o FTP. Se debe verificar todo el tráfico interno, por ejemplo, entre balanceadores de carga, servidores web o sistemas *backend*.
- Si se usan algoritmos criptográficos antiguos o débiles.
- Si se utilizan claves criptográficas por defecto, o se generan o reutilizan claves criptográficas débiles o falta una adecuada gestión de claves o rotación de estas. ¿Se incluyen las claves en los repositorios de código?
- Si no se aplica el cifrado, es decir, si faltan directivas de seguridad de cabeceras HTTP.
- Si el certificado de servidor y la cadena de confianza están correctamente validados.
- Si los vectores de inicialización se ignoran, reutilizan o no se generan de manera suficientemente segura para el modo de operación criptográfica. ¿Se utiliza un modo de operación inseguro como ECB?
- Si se utilizan las contraseñas como claves criptográficas a falta de una función de derivación de claves a partir de las contraseñas.
- Si se emplean funciones aleatorias para propósitos criptográficos, pero sin haber sido diseñadas para cumplir con los requisitos de criptografía.
- Si se utilizan funciones hash en desuso como MD5 o SHA1 o se utilizan funciones de hash no criptográficas cuando estas son necesarias.
- Si se utilizan métodos de relleno *(padding)* obsoletos como PKCS número 1 v1.5.
- Si se pueden explotar los mensajes de error criptográfico como un canal lateral en forma de ataques de criptoanálisis por modificación de relleno *(oracle padding)*.

Cómo se previene:

- Clasificar los datos que la aplicación procesa, almacena y transmite. Identificar qué datos son sensibles de acuerdo con los reglamentos de privacidad, requisitos regulatorios y necesidades de negocio.
- No almacenar datos sensibles que no sean necesarios. Descartarlos tan pronto como sea posible o utilizar *tokenización* que cumpla con PCI DSS.
- Cifrar toda la información sensible en reposo.
- Asegurarse de que se utilizan algoritmos y protocolos actuales y fuertes. Usar una adecuada gestión de claves.
- Cifrar todos los datos en tránsito con protocolos como TLS con cifradores de confidencialidad adelantada (*forward secrecy*, o FS), priorización de cifradores por parte del servidor y parámetros seguros. Forzar el cifrado con directivas como HSTS (HTTP Strict Transport Security).
- Deshabilitar la caché de la respuesta que contenga datos sensibles.
- Aplicar los controles de seguridad requeridos según la clasificación de datos.
- No usar protocolos obsoletos como FTP y SMTP para enviar datos sensibles.

- Almacenar las contraseñas usando funciones de hash fuertes con un factor de retardo, como Argon2, script, bcrypt o PBKDF2.
- Los vectores de inicialización se deben elegir de manera adecuada al modo de operación. Para muchos modos, eso implica usar un CSPRNG (generador de números pseudoaleatorios criptográficamente seguro). Para los modos que requieren un *nonce*, el vector de inicialización (IV) no necesita un CSPRNG. En cualquier caso, el IV nunca se debería usar dos veces para una clave fija.
- Utilizar siempre cifrado autenticado en lugar de sólo cifrado.
- Las claves se deberían generar criptográficamente de manera aleatoria y almacenarse en memoria como *arrays* de bytes. Si se utiliza una contraseña, se debe convertir en clave mediante una función de derivación de claves basada en contraseñas.
- Hay que asegurarse de utilizar aleatoriedad criptográfica donde resulte apropiado y que no se haya inicializado de forma predecible o con baja entropía.
- Evitar funciones criptográficas obsoletas y esquemas de acolchado (*padding*) tales como MD5, SHA1 y PKCS número 1 v1.5.
- Verificar de manera independiente la efectividad de la configuración y los ajustes.

Ejemplo de explotación:

En la Juice Shop existe la página "Sobre nosotros", en la URL http://localhost:3000/#/about. En ella, el vínculo "Eche un vistazo a nuestras aburridas condiciones de uso" conduce a la URL http://localhost:3000/ftp/legal.md, que presenta un documento en formato Markdown con información legal.

Si accedemos a la URL http://localhost:3000/ftp/ observamos que se sirve un listado de directorio con sus ficheros:

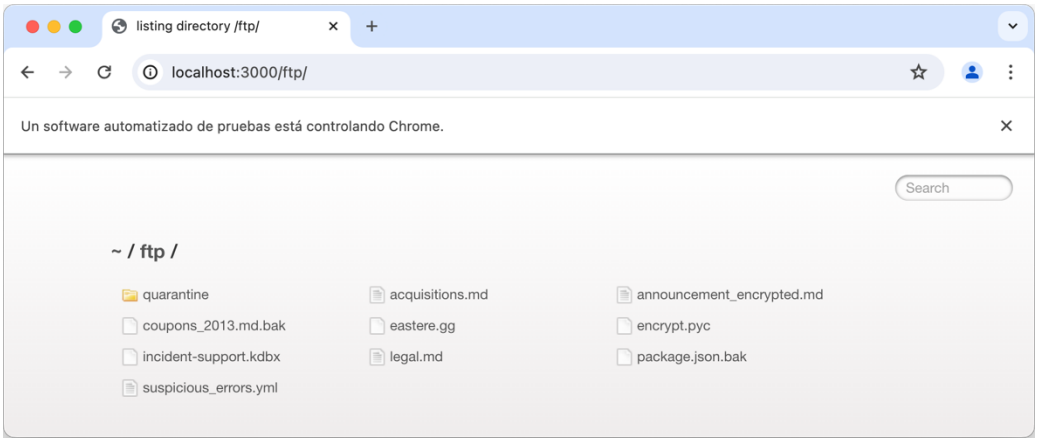


Ilustración 21: URL que sirve un listada de carpetas y ficheros
Fuente: elaboración propia

Si accedemos al documento `acquisitions.md`, vemos que contiene información confidencial sobre las adquisiciones planeadas sobre competidores. Se accede con la URL http://localhost:3000/ftp/acquisitions.md, por tanto, no se cifran los

datos en tránsito, pudiendo ser interceptados con facilidad, ni el documento está cifrado en reposo, pues se puede ver su contenido:

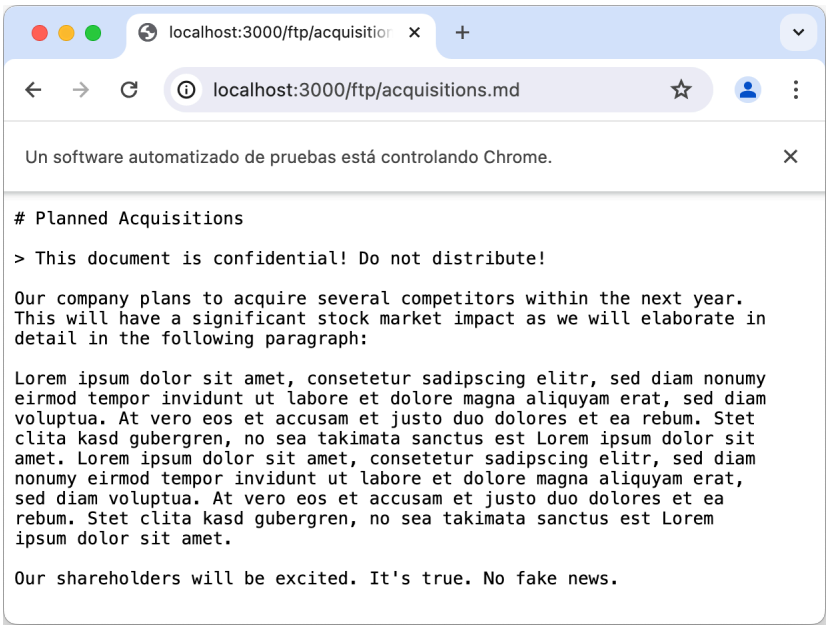


Ilustración 22: Fichero confidencial almacenado y transmitido sin cifrar
Fuente: elaboración propia

Uso de Helmet en Express

La guía de mejores prácticas de seguridad de Express [9] recomienda el uso de la biblioteca Helmet para incorporar ciertas cabeceras del servidor relacionadas con la seguridad. En el caso de `Strict-Transport-Security`, comentado en esta sección, se introduciría dicha cabecera con `helmet.hsts` para forzar la conexión HTTPS con el servidor.

También se pueden generar otras cabeceras como `Content-Security-Policy`, que puede mitigar ataques de XSS (*cross-site scripting*). Se ha visto que ZAP ha detectado en el escaneo automatizado de la Juice Shop que falta esa cabecera.

Y otras, como `X-Frame-Options`, que pueden proteger frente a ataques se *clickjacking*.

5.3. A03:2021 – Inyección

Una aplicación web es vulnerable a ataques de inyección cuando:

- Los datos que ha introducido el usuario no se validan, filtran o sanean.
- Se interpretan directamente consultas dinámicas o no parametrizadas sin codificar los parámetros según el contexto.
- Se utilizan datos dañinos en los parámetros de búsqueda en un ORM para acceder a información sensible.

- Se usan datos dañinos directamente o se concatenan. El SQL o el comando contiene la estructura y datos dañinos en consultas dinámicas, comandos o procedimientos almacenados.

Algunas inyecciones comunes son: SQL, NoSQL, comandos del sistema operativo, ORM, LDAP, etc. La mejor forma de detectar si las aplicaciones son vulnerables a inyecciones es la revisión del código fuente. También se recomienda la prueba automática de todos los parámetros, cabeceras, URLs, cookies, JSON, SOAP, etc. Se pueden incluir herramientas de análisis de código (SAST, DAST) en la fase de pruebas antes del paso a producción.

Cómo se previene:

Para prevenir estos ataques se debe mantener los datos separados de los comandos y las consultas:

- Se puede utilizar una API segura que evite utilizar el intérprete por completo y ofrezca una interfaz parametrizada, o migrar a una herramienta ORM.
- Hacer la validación de la entrada en el servidor. No es una defensa completa, pues muchas aplicaciones requieren los caracteres especiales.
- En consultas dinámicas, *escapar* los caracteres especiales, utilizando la sintaxis de *escapado* según el intérprete.
- Usar cláusulas `LIMIT` y otros controles SQL en las consultas para evitar la revelación masiva de registros en caso de inyección SQL.

Ejemplo de explotación:

Podemos utilizar la herramienta Fuzzer de OWASP ZAP para realizar un ataque de inyección sobre la página de inicio de sesión [32]. Esta herramienta se utiliza para enviar muchos datos a un objetivo, a menudo en forma de entradas válidas o inesperadas [33].

Para ello, debemos descargarnos primero el complemento "FuzzDB Offensive", que contiene *payloads* de inyección SQL (entre otros) para la herramienta Fuzzer.

Nos colocamos en la página de inicio de sesión, http://localhost:3000/#/login e introducimos cualquier usuario y contraseña. Una vez ZAP ha registrado la petición en el historial, hacemos clic sobre ella y elegimos la opción: "Atacar → Fuzz…":

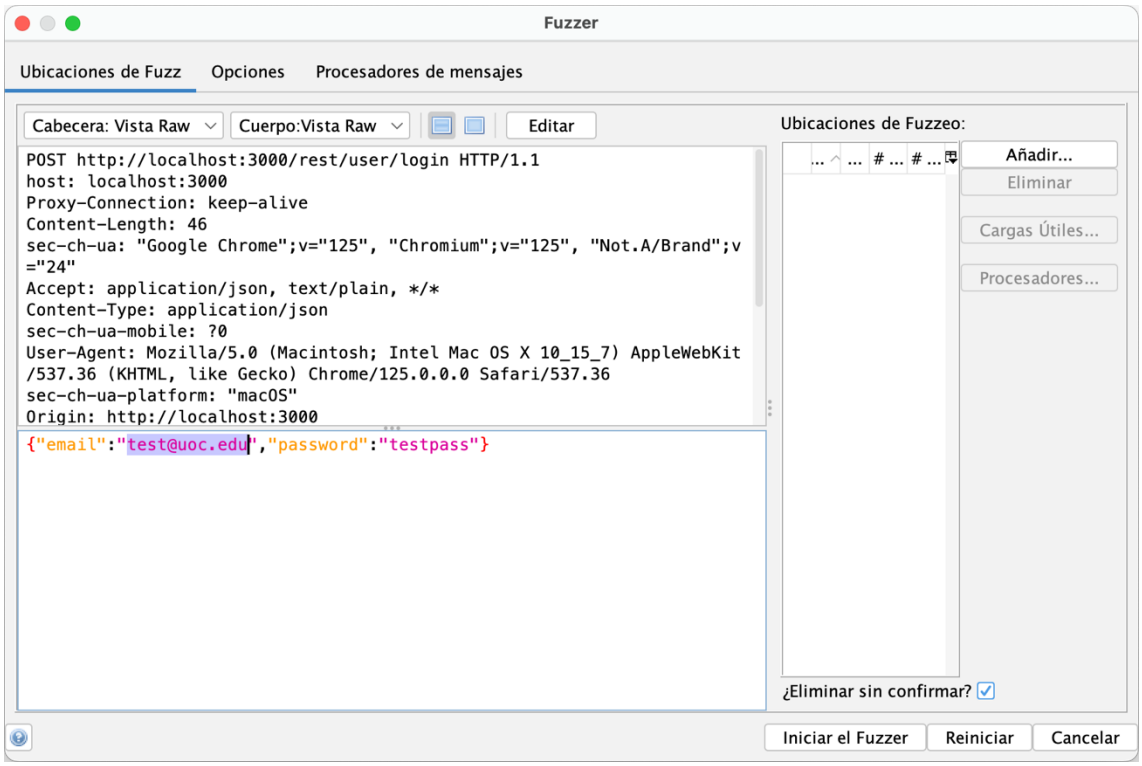


Ilustración 23: Petición de inicio de sesión capturada en la herramienta Fuzzer de ZAP
Fuente: elaboración propia

Se marca el parámetro que se quiere explotar, en este caso el correo electrónico. En la sección "Ubicaciones de Fuzzeo" pulsamos el botón "Añadir…". Se selecciona "Fuzzers de archivo" como tipo y se marca: "fuzzdb → attack → sql-injection":

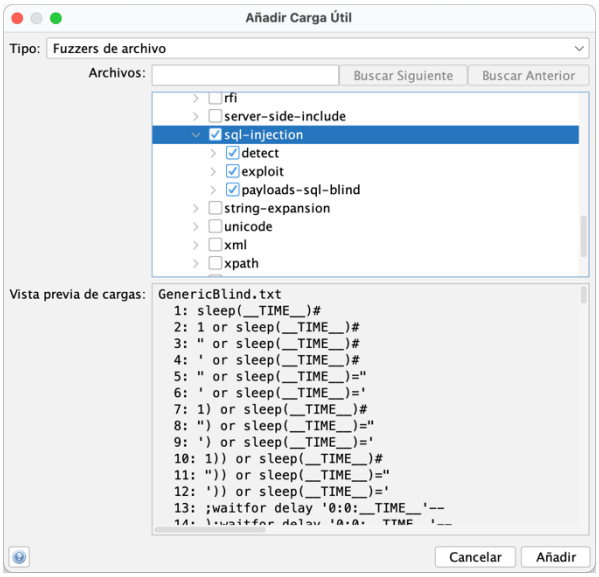


Ilustración 24: Configuración de la utilidad Fuzzer para inyección SQL
Fuente: elaboración propia

Tras pulsar el botón "Añadir", se pulsa el botón "Iniciar el Fuzzer". ZAP prueba los distintos patrones de inyección SQL incluidos en el complemento que hemos descargado y recoge los resultados en la pestaña "Fuzzer":

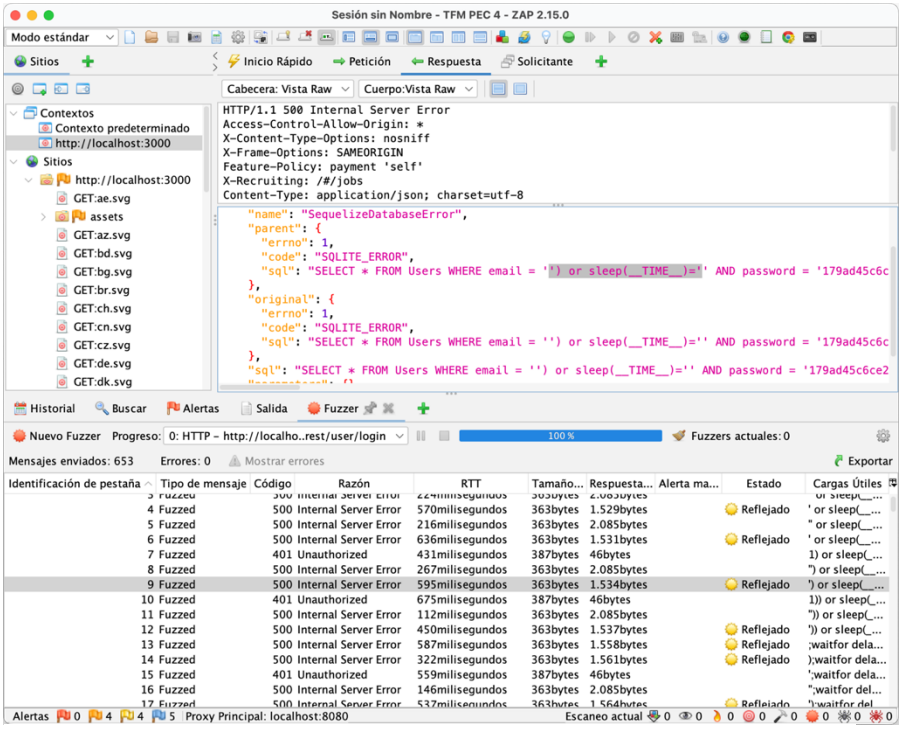


Ilustración 25: Resultado del Fuzzer en ZAP.
Fuente: elaboración propia.

En las peticiones que aparece el estado "Reflejado" significa que el *payload* aparece devuelto en la respuesta, como muestra la pantalla. En este caso vemos que la respuesta revela información importante sobre la consulta SQL que valida el inicio de sesión.

Hay algunas peticiones que devuelven el código de estado 401 (no autorizado), otras el 500 (error interno de servidor) y alguna el 200 (OK). Un ejemplo de esta última se muestra en la siguiente pantalla:

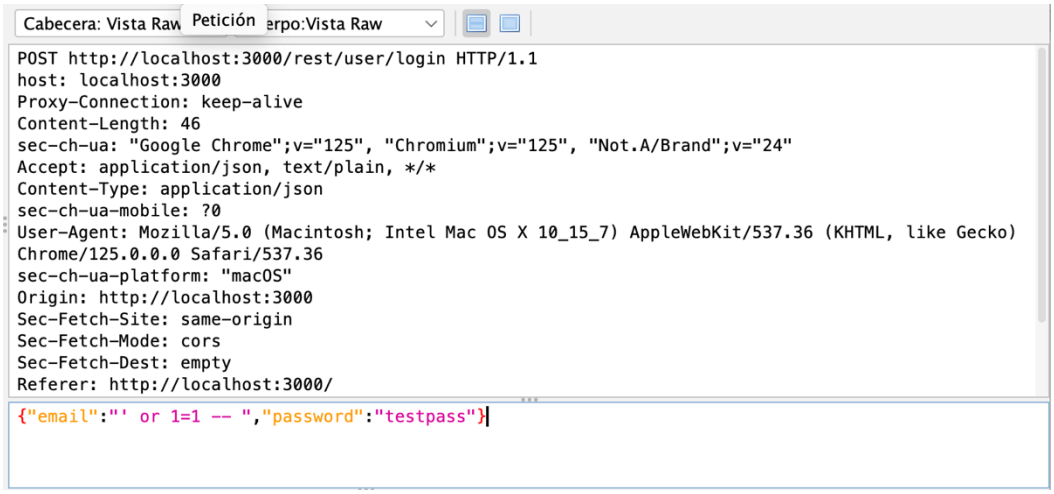


Ilustración 26: Petición realizada con Fuzzer con una inyección SQL exitosa
Fuente: elaboración propia

Como se observa, es una inyección de SQL clásica, `' or 1=1 --`, que demuestra que el formulario de inicio de sesión es vulnerable a inyección de código SQL. Se observa que la respuesta devuelve un token de autenticación:


```
{"authentication":{"token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzd
GF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MSwidXNlcm5hbWUiOiIiLCJlbWFpbCI
6ImFkbWluQGp1aWNlLXNoLm9wIiwicGFzc3dvcmQiOiIwMTkyMDIzYTdiYmQ3MzI1MDUxN
mYwNjlkZjE4YjUwMCIsInJvbGUiOiJhZG1pbiIsImRlbHV4ZVRva2VuIjoiIiwibGFzdEx
vZ2luSXAiOiIiLCJwcm9maWxlSW1hZ2UiOiJhc3NldHMvcHVibGljL2ltYWdlcy91cGxvY
WRzL2RlZmF1bHRBZG1pbi5wbmciLCJ0b3RwU2VjcmV0IjoiIiwiaXNBY3RpdmUiOnRydWU
sImNyZWF0ZWRBdCI6IjIwMjQtMDUtMjggMTA6MDY6MjQuNDU1ICswMDowMCIsInVwZGF0Z
WRBdCI6IjIwMjQtMDUtMjggMTA6MDY6MjQuNDU1ICswMDowMCIsImRlbGV0ZWRBdCI6bnV
sbH0sImlhdCI6MTcxNzAwOTY4Nn0.h5SWyROjLNoonbgfJZJwV9xjkPuFdBzr-
Gj4UvxbOLMWsLTHlTLj_Z8gECRVLNdQrVgH-0nTVDjEdOQq65b3Zxp8MGNJXNN0U-
UIZxvBc3skYHbRkr7kA6pt_OLjJ0mpp8MdfIaaTet7LrX7cgyfaFhqFZj6LkICJJ3X8bcg
O6w","bid":1,"umail":"admin@juice-sh.op"}}
```


Y que corresponde además a la cuenta de administrador. El token JWT podría decodificarse online con herramientas como jwt.io.

Para evitar este tipo de ataques de fuerza bruta contra las páginas de autorización, como el que se acaba de ejecutar en el ejemplo, Express propone desde sus recomendaciones de seguridad [9] el uso de dos métricas:

- El número de intentos fallidos de inicio de sesión por el mismo usuario desde la misma dirección IP.
- El número de intentos fallidos desde una IP durante un periodo de tiempo más largo.

Y se propone el uso del paquete `rate-limiter-flexible` como forma rápida de implementar estas medidas de seguridad.

5.4. A04:2021 – Diseño inseguro

Es una nueva categoría del OWASP Top 10 de 2021 que se centra en los riesgos relativos al diseño y a los fallos de arquitectura. Se hace un llamamiento al uso del modelado de amenazas, los patrones de diseño seguro y las arquitecturas de referencia.

El diseño seguro es una cultura y una metodología que evalúa las amenazas constantemente y se asegura de que el código se diseña de manera robusta y se prueba para evitar los métodos de ataque conocidos.

El software seguro requiere un ciclo de vida de desarrollo seguro, alguna forma de patrones de diseño seguros, una biblioteca de componentes seguros, herramientas y modelado de amenazas.

Cómo evitar el diseño inseguro:

- Establecer y usar un ciclo de vida de desarrollo del software seguro, con profesionales AppSec que ayuden a evaluar y diseñar controles de seguridad y privacidad.
- Establecer y utilizar una biblioteca de patrones de diseño seguro lista para usar.
- Utilizar el modelado de amenazas para flujos críticos de autenticación, control de acceso y lógica de negocio.
- Integrar el lenguaje de seguridad y los controles en las historias de usuario.
- Integrar comprobaciones de viabilidad en cada capa de la aplicación (desde el *frontend* al *backend*).
- Escribir pruebas unitarias y de integración para validar que todos los flujos críticos son resistentes al modelo de amenazas. Compilar casos de uso y de mal uso para cada capa de la aplicación.
- Separar las capas de sistema y las capas de red en función de la exposición y necesidades de protección.
- Limitar el consumo de recursos por usuario o servicio.

Un ejemplo de diseño inseguro que podemos observar en la Juice Shop es el sistema de restablecimiento de contraseña mediante "preguntas de seguridad":

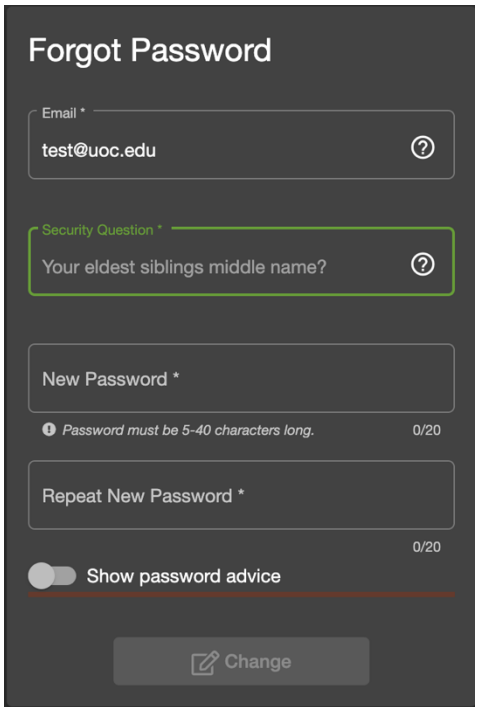


Ilustración 27: Proceso de restablecimiento de cuenta mediante preguntas de seguridad
Fuente: elaboración propia

Esta práctica está totalmente desaconsejada por guías como NIST 800-63b y otras recomendaciones de OWASP, incluido el Top 10. Estas preguntas y respuestas no son de confianza como prueba de identidad, puesto que más de una persona puede conocer las respuestas. Deben sustituirse por un diseño más seguro.

HTTP Request Smuggling en Node.js

La CWE-444 (Inconsistent Interpretation of HTTP Requests), que se incluye en este apartado, se menciona también en la guía de mejores prácticas de seguridad de Node.js [7].

En el caso de Node.js el ataque se produce cuando hay una aplicación web tras un servidor proxy. Este servidor proxy recibe las peticiones y las dirige a un servidor web interno, donde está la aplicación Node.js. El ataque se produce cuando el servidor proxy y el interno interpretan las peticiones HTTP ambiguas de manera diferente, pudiendo un atacante enviar un mensaje malicioso que se salte el servidor proxy y sólo lo reciba el servidor web interno.

Algunas de las mitigaciones posibles son: no utilizar la opción `insecureHTTPParser` al crear el servidor HTTP o configurar el servidor proxy de manera que normalice las peticiones ambiguas.

5.5. A05:2021 – Configuración de seguridad incorrecta

Una aplicación puede ser vulnerable a esta categoría si:

- Le falta el endurecimiento (*hardening*) de seguridad apropiado en cualquier parte de la pila de la aplicación o si hay permisos mal configurados en los servicios en la nube.
- Se habilitan o instalan características innecesarias (por ejemplo, puertos, servicios, páginas, cuentas o privilegios).
- Hay habilitadas cuentas por defecto con sus contraseñas sin cambiar.
- El manejo de errores revela a los usuarios trazas de la pila u otros tipos errores demasiado informativos.
- En sistemas actualizados, las últimas características de seguridad están deshabilitadas o no configuradas de manera segura.
- Los ajustes de seguridad en los servidores de la aplicación, o los *frameworks*, bibliotecas, bases de datos, etc. no están configurados con valores seguros.
- El software está desactualizado o es vulnerable.

Para evitar estas vulnerabilidades se deben implementar procedimientos de instalación seguros, incluyendo:

- Un proceso de endurecimiento (*hardening*) repetible, de forma que sea rápido y sencillo desplegar otro entorno correctamente configurado. Los entornos de desarrollo, QA y producción deberían estar configurados de manera idéntica, con credenciales diferentes en cada entorno. El proceso debería estar automatizado para minimizar los esfuerzos necesarios para configurar un nuevo entorno.
- Una plataforma mínima sin características, componentes, documentación o ejemplos innecesarios. Eliminar o no instalar las características y *frameworks* no utilizados.

- Una tarea para revisar y actualizar las configuraciones apropiadas a todas las actualizaciones y parches, como parte del proceso de gestión de parches. Revisar los permisos de almacenamiento en la nube.
- Una arquitectura de aplicación segmentada que ofrezca una separación segura y efectiva entre componentes o inquilinos, con segmentación, *containerización* o grupos de seguridad (ACLs).
- Enviar directivas de seguridad a los clientes, como cabeceras de seguridad.
- Un proceso automatizado para verificar la efectividad de las configuraciones y ajustes en todos los entornos.

Ejemplos de escenarios de ataque:

En el punto anterior sobre inyección de código se ha visto cómo un error en la consulta tras la inyección retorna en la respuesta información importante sobre la BD subyacente. Se observa que hace referencia a SQLite, Sequelize e incluso se incluye la sentencia SQL completa.

Otro ejemplo similar se produce si hacemos una petición a un *endpoint* del servicio REST que no es válido. Si se observa en ZAP el ejemplo de la petición de inicio de sesión que se vio con Fuzzer, es una petición `POST` a la URL http://localhost:3000/rest/user/login. Si solicitamos esta URL mediante `GET` directamente desde el navegador:

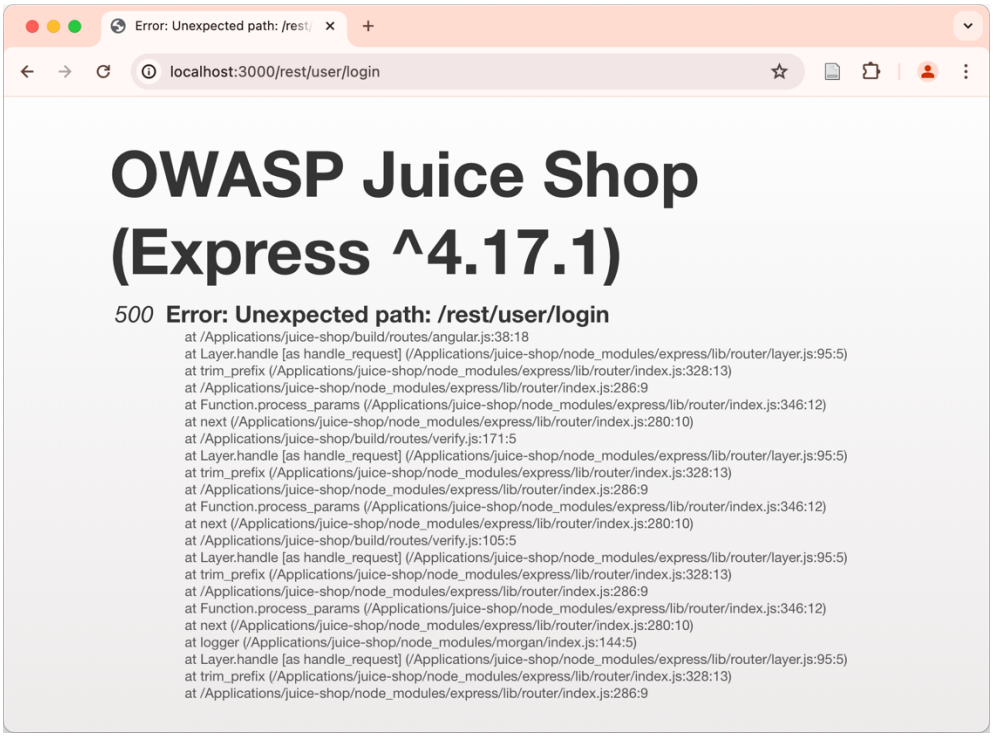


Ilustración 28: Manejador de errores mostrando la traza de la pila
Fuente: elaboración propia

Se observa que se presenta el error del manejador de errores global, que envía un error al cliente con la traza de la pila, que contiene información sensible como rutas de carpetas del servidor web.

Reducción del *fingerprinting* en Express

Desde la guía de mejores prácticas de Express [9] se lanzan varias sugerencias para reducir o mitigar el *fingerprinting*, es decir, la capacidad del atacante de determinar qué software ejecuta un servidor. Este ataque se facilita cuando se dejan opciones de configuración por defecto. En el ejemplo anterior, se ve claramente tras provocar un error que el servidor utiliza Express.

Por ejemplo, por defecto Express envía la cabecera `X-Powered-By`, que se puede desactivar con `app.disable('x-powered-by')`. También tiene su página por defecto para el error `404 Not Found`, que se podría personalizar con un mensaje neutro que no revele la tecnología.

De igual forma, se sugiere no utilizar el nombre por defecto de las cookies y cambiarlo por otro que no favorezca el *fingerprinting*.

5.6. A06:2021 – Componentes Vulnerables y Desactualizados

Se es vulnerable a este punto cuando:

- No se conocen las versiones de todos los componentes utilizados (tanto en el lado del cliente como del servidor). Esto incluye tanto los componentes utilizados directamente como las dependencias anidadas.
- El software es vulnerable, ya no se soporta o está desactualizado. Esto incluye el sistema operativo, el servidor web o de aplicaciones, el gestor de base de datos, las aplicaciones, APIs y todos los componentes, entornos de ejecución y bibliotecas.
- No se escanean las vulnerabilidades de manera periódica y no se siguen los boletines de seguridad relativos a los componentes que se utilizan.
- No se corrigen o actualizan las plataformas, *frameworks* y dependencias utilizadas de manera eficaz y en función del riesgo. Por ejemplo, una empresa donde se parchea el código mensual o trimestralmente puede quedar expuesta a riesgos innecesarios por vulnerabilidades que pueden estar corregidas.
- El equipo de desarrollo no prueba la compatibilidad de la bibliotecas parcheadas o actualizadas.
- No se configuran los componentes de manera segura, tal como se ha visto en el punto anterior.

Para prevenir estas vulnerabilidades, debe existir un proceso de gestión de parches destinado a:

- Eliminar las dependencias no utilizadas, características innecesarias, componentes, ficheros y documentación.
- Hacer un inventario continuo de las versiones, tanto del lado cliente como de los componentes del servidor (*frameworks*, bibliotecas) y sus dependencias, utilizando herramientas como el OWASP Dependency Check, Retire.js, etc. Se deben monitorizar continuamente fuentes como la Common Vulnerability and Exposures (CVE) y la National Vulnerability

Database (NVD) en busca de vulnerabilidades en los componentes. Se pueden utilizar herramientas de composición de software para automatizar el proceso. Y suscribirse a recibir alertas por correo sobre las vulnerabilidades relacionadas con los componentes que se usan.

- Obtener los componentes de fuentes oficiales y vínculos seguros. Se debe mostrar preferencia por los paquetes firmados para reducir la posibilidad de incorporar un componente malicioso (ver el punto: "5.8. A08:2021 – Fallos en el Software y en la Integridad de los Datos").
- Monitorizar las bibliotecas y componentes que ya no se mantienen o cuyas versiones anteriores ya no reciben parches de seguridad.

Retire.js es un escáner que detecta el uso de bibliotecas JavaScript con vulnerabilidades conocidas[9]. Está integrado tanto en ZAP, como en la versión comercial de Burp Suite mediante el uso de extensiones.

En ZAP, los resultados del análisis con Retire.js se muestran en el escaneo automatizado:

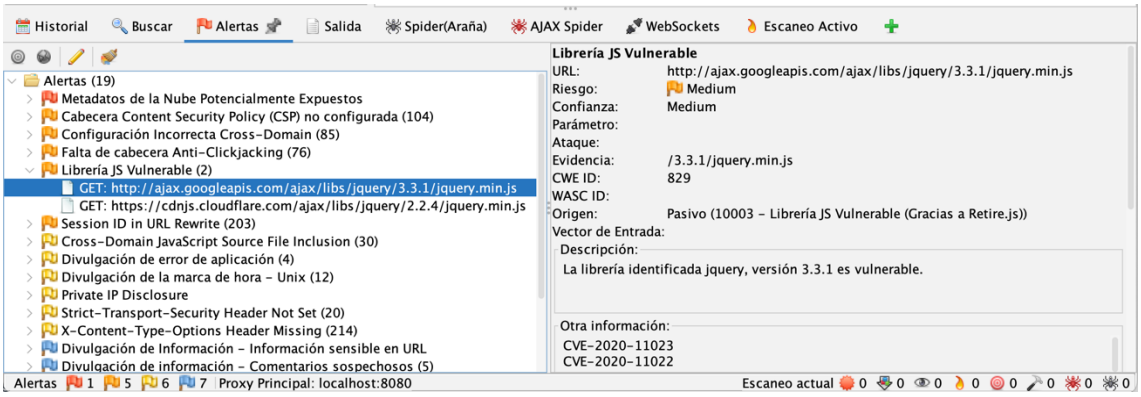


Ilustración 29: Escaneo automatizado con ZAP mostrando bibliotecas vulnerables
Fuente: elaboración propia

En la captura, se muestra que la aplicación Juice Shop incluye dos instancias de bibliotecas JQuery en versiones vulnerables (2.2.4 y 3.3.1). Se incluye las CVEs de referencia y otras referencias web.

Versiones obsoletas de Express

En la guía de mejores prácticas de Express [9] se indica que las versiones 2 y 3 ya no se mantienen. Dentro de la versión 4, la página "Security Updates"[10] lista qué vulnerabilidades concretas se han detectado y corregido en cada una de las versiones concretas. Si se está utilizando una versión vulnerable, se recomienda migrar a la versión más reciente que no sea vulnerable.

[9] Fuente: GitHub - RetireJS/retire.js
[10] https://expressjs.com/en/advanced/security-updates.html

5.7. A07:2021 – Fallos de Identificación y Autenticación

La confirmación de la identidad de usuario, la autenticación y la gestión de sesiones son críticos para proteger frente a los ataques relacionados con la autenticación. Puede haber debilidades de autenticación si la aplicación:

- Permite ataques automatizados como la reutilización de credenciales, donde un atacante tiene una lista de usuarios y contraseñas válidos, que puede haber obtenido de un filtrado de información de otro sitio.
- Permite los ataques automatizados o de fuerza bruta.
- Permite contraseñas débiles del estilo de "123456" o "admin/admin".
- Utiliza procesos de recuperación de contraseña débiles o inefectivos, tales como preguntas de seguridad, que no pueden ser seguras.
- Almacena las contraseñas en texto claro, cifrado o con algoritmos de hash débiles (ver "5.2. A02:2021 – Fallos Criptográficos").
- No tiene autenticación multi-factor o no la implementa bien.
- Expone identificadores de sesión en la URL.
- Reutiliza identificadores de sesión tras un inicio de sesión correcto.
- No invalida correctamente los identificadores de sesión o los tokens de autenticación durante el cierre de sesión o tras un periodo de inactividad.

Formas de prevención:

- Implementar la autenticación multi-factor para evitar los ataques de fuerza bruta y reutilización de credenciales robadas.
- No desplegar el producto con ninguna credencial por defecto, sobre todo para los usuarios administradores.
- Implementar comprobaciones de la debilidad de la contraseña, por ejemplo, utilizando listas de las peores contraseñas.
- Alinear las políticas de longitud de la contraseña, complejidad y rotación con las recomendaciones de la guía NIST 800-63b, sección 5.1.1, o con otra guía moderna similar.
- Asegurarse de que el registro, restablecimiento de contraseña y el uso de APIs no permiten los ataques de enumeración de usuarios y se utilizan los mismos mensajes genéricos en todas las salidas.
- Limitar o retrasar progresivamente los intentos de inicio de sesión fallidos. Registrar todos los fallos y alertar a los administradores cuando se detecten ataques de fuerza bruta o reutilización de credenciales.
- Utilizar un gestor de sesiones seguro en el lado del servidor que genere un nuevo identificador de sesión aleatorio con alta entropía tras el inicio de sesión. Dichos identificadores no deben aparecer en la URL, se deben almacenar de manera segura, invalidarse tras el cierre de sesión o tras un periodo de inactividad.

A modo de ejemplo, vamos a comprobar si la aplicación Juice Shop es susceptible a ataques de fuerza bruta en el proceso de inicio de sesión. Para ello, utilizaremos Burp Suite. Desde la aplicación, iniciamos la captura de tráfico e iniciamos sesión en la aplicación web con cualquier credencial:

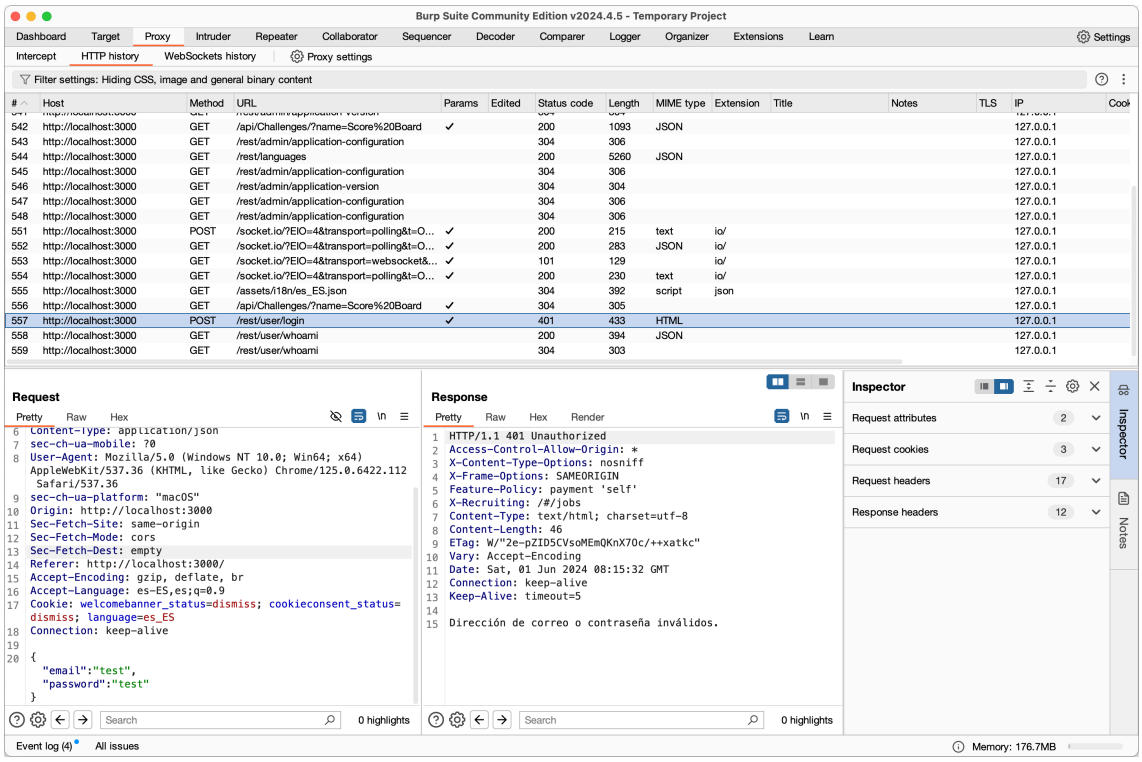


Ilustración 30: Captura de la petición de inicio de sesión en Burp Suite
Fuente: elaboración propia

La pantalla muestra la petición POST al *endpoint* de inicio de sesión `/rest/user/login`. Pulsamos con el botón derecho sobre la petición 557 y elegimos la opción "Send to Intruder". La petición aparece ahora en la pestaña "Intruder":

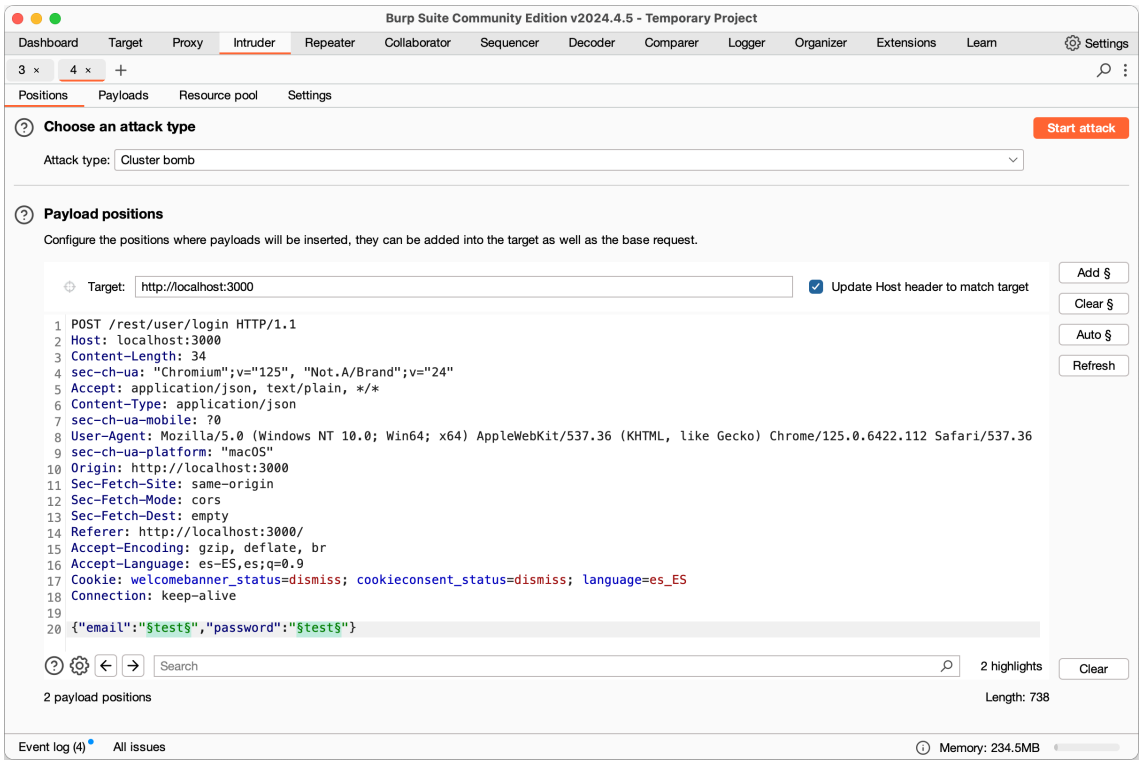


Ilustración 31: Petición capturada en la herramienta Intruder de Burp Suite
Fuente: elaboración propia

A continuación, se selecciona en el cuerpo de la petición POST el correo "test" y se pulsa el botón "Add §". Se hace los mismo con la contraseña, hasta que el cuerpo quede en la forma: `{"email":"§test§","password":"§test§"}`. Después se selecciona como tipo de ataque "Cluster bomb". Este ataque probará todas las permutaciones de usuario y contraseña que especifiquemos en el siguiente paso.

Ahora, en la pestaña "Intruder → Payloads", se selecciona cada uno de los dos *payloads*. En el primero, para el usuario, introducimos admin@juice-sh.op. Es el correo del administrador, que hemos visto anteriormente en la reseña de productos. Se intentará hacer un ataque de fuerza bruta sobre este usuario. En el segundo de los `payloads`, para las contraseñas, introducimos una lista de las diez mil contraseñas más habituales disponible en el repositorio de GitHub SecLists de Daniel Miessler [34].

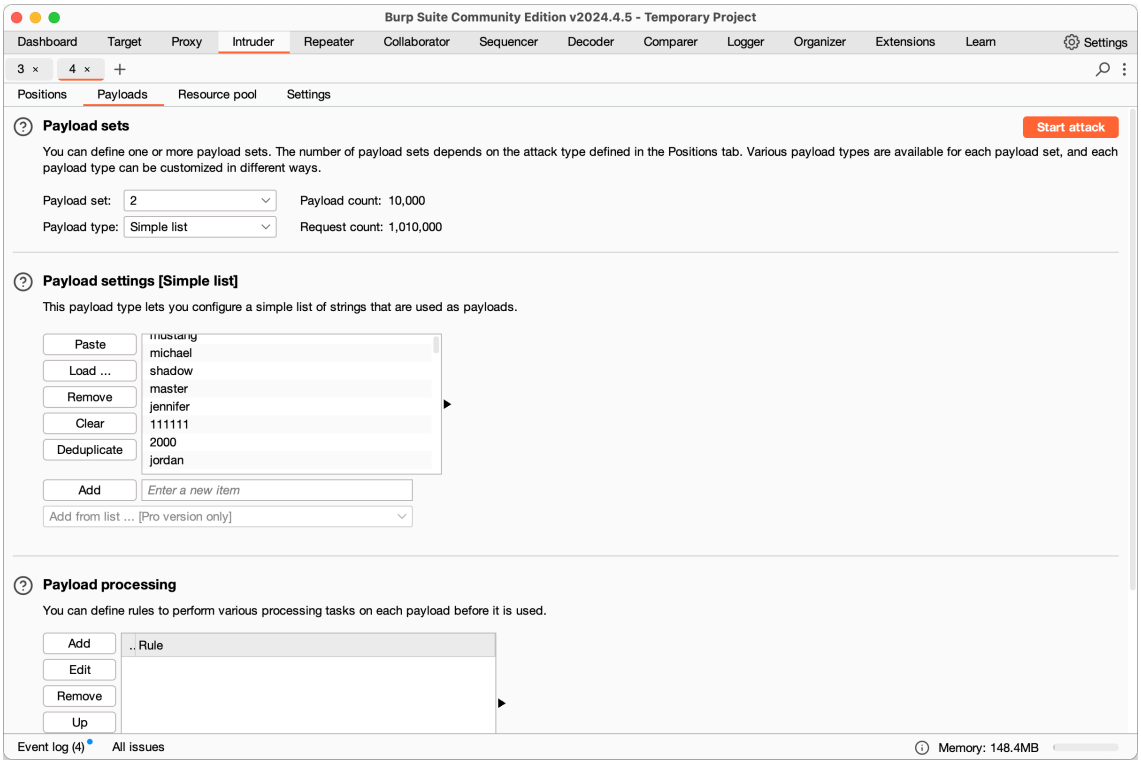


Ilustración 32: Configuración de la herramienta Intruder con *payloads* de contraseñas
Fuente: elaboración propia

Hecho esto, pulsamos el botón "Start attack" para comenzar el ataque. En este ejemplo sólo hemos introducido un usuario, el correo de administrador que conocemos. En un caso más general, podríamos introducir una lista de usuarios o correos de algún volcado de datos filtrado.

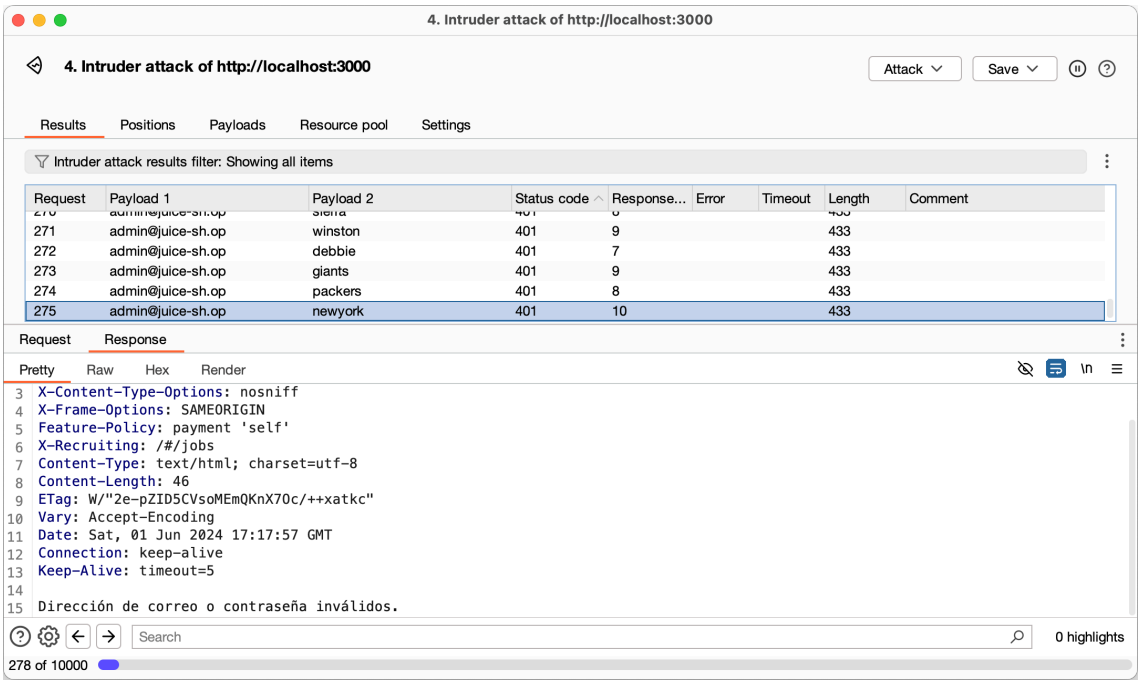


Ilustración 33: Ejemplo de petición del ataque con la herramienta Intruder de Burp Suite
Fuente: elaboración propia

En esta captura puede observarse cómo Burp Suite ya ha podido probar 275 contraseñas de las que hemos indicado. En todos los casos, la respuesta de la aplicación ha sido el código HTTP 401 (Unauthorized), con el mensaje "Dirección de correo o contraseña inválidos".

Se observa que no hay ningún mecanismo que limite el número de veces que un usuario puede iniciar sesión, ni que imponga un tiempo entre intentos fallidos.

DNS *Rebinding* en Node.js

La guía de mejores prácticas de seguridad de Node.js [7] hace referencia a la CWE-346 (Origin Validation Error), que también se incluye en este apartado del Top 10. En el caso de Node.js se refiere a ejecutar la aplicación con el inspector de depuración habilitado (iniciando el proceso con la opción `--inspect`). Esto permite que las herramientas de depuración de los navegadores y otras aplicaciones como Visual Studio Code se conecten con la aplicación web en desarrollo para ayudar en la depuración.

El inspector de depuración puede ser objetivo de los atacantes y las políticas del mismo origen implementadas por los navegadores deberían evitar a los scripts de otros orígenes llegar al inspector de depuración. Sin embargo, mediante un ataque de DNS *rebinding*, un atacante podría hacer que el origen de sus peticiones pareciese que se originan en la IP local.

Como mitigación, Node.js propone asegurarse de no ejecutar el inspector de depuración en entornos de producción.

5.8. A08:2021 – Fallos en el Software y en la Integridad de los Datos

Esta vulnerabilidad se refiere al código e infraestructura que no protege contra las violaciones de integridad. Un ejemplo de ello es una aplicación que se basa en *plugins*, bibliotecas o módulos de fuentes que no son de confianza, repositorios y redes de entrega de contenidos (CDNs). Un pipeline CI/CD puede introducir potencialmente un acceso no autorizado, código malicioso o comprometer el sistema.

Además, ahora muchas aplicaciones incluyen funcionalidad de actualización automática, donde los parches se descargan de manera automática sin la suficiente verificación de integridad y se aplican a lo que era una aplicación de confianza. Los atacantes pueden cargar potencialmente sus propios parches para que se distribuyan y ejecuten en todas las instalaciones. Otro ejemplo se da cuando hay objetos o datos que se codifican o serializan en una estructura que un atacante puede ver y modificar por ser vulnerable a una deserialización insegura.

Formas de prevención:

- Utilizar firmas digitales o mecanismos similares para comprobar que el software o sus datos vienen de la fuente esperada y no han sido modificados.
- Asegurarse de que las bibliotecas y dependencias, tales como npm o Maven, están consumiendo repositorios de confianza. También se puede contemplar alojarlas en un repositorio interno de confianza que haya sido previamente comprobado.
- Se pueden utilizar herramientas de seguridad de la cadena de suministro, tales como el Dependency Check o CycloneDX, ambas de OWASP, para verificar que los componentes no contienen vulnerabilidades conocidas.
- Asegurarse de que existe un proceso de revisión de los cambios en el código y en la configuración que minimice la posibilidad de que se introduzca código o configuraciones maliciosas en el *pipeline* de software.
- Asegurarse de que el pipeline de CI/CD tiene una adecuada separación, configuración y control de acceso que garantice la integridad del código que atraviesa los procesos de *build* y *deploy*.
- Asegurarse de que los datos serializados sin cifrar o sin firmar no se envían a clientes que no son de confianza sin alguna forma de comprobación de integridad o firma digital que detecte si han sido modificados.

En Juice Shop existe la URL http://localhost:3000/api-docs, que se puede descubrir con un análisis manual o automatizado, que presenta una documentación sobre una API B2B de la tienda:

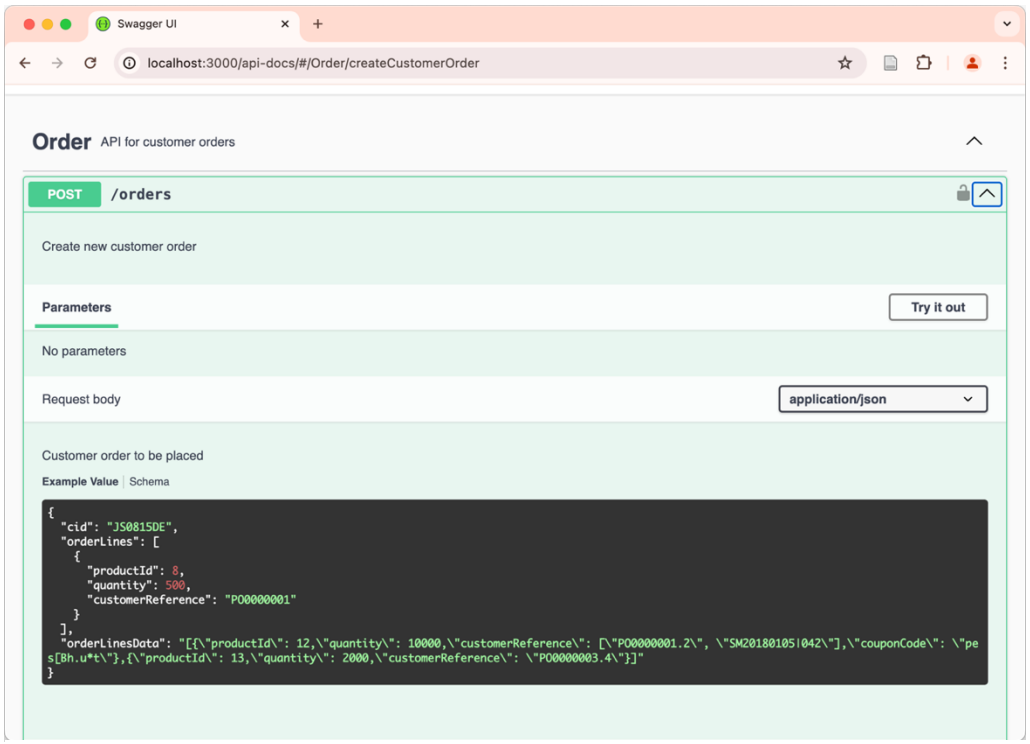


Ilustración 34: Página de ayuda de una API en Juice Shop
Fuente: elaboración propia

Describe una petición POST para hacer pedidos y, en el ejemplo se muestra que la propiedad `orderLinesData` es una cadena que admite un JSON arbitrario para representar cada línea del pedido. Si pulsamos el botón "Try it out", podemos mandar el siguiente JSON malicioso que provoque un bucle infinito: `{"orderLinesData": "(function bucle() { while(true); })()"}`.

Antes de lanzar la petición POST, pulsamos el botón "Authorize" e introducimos un token JWT que obtengamos del inicio de sesión. Después, pulsamos el botón "Execute":

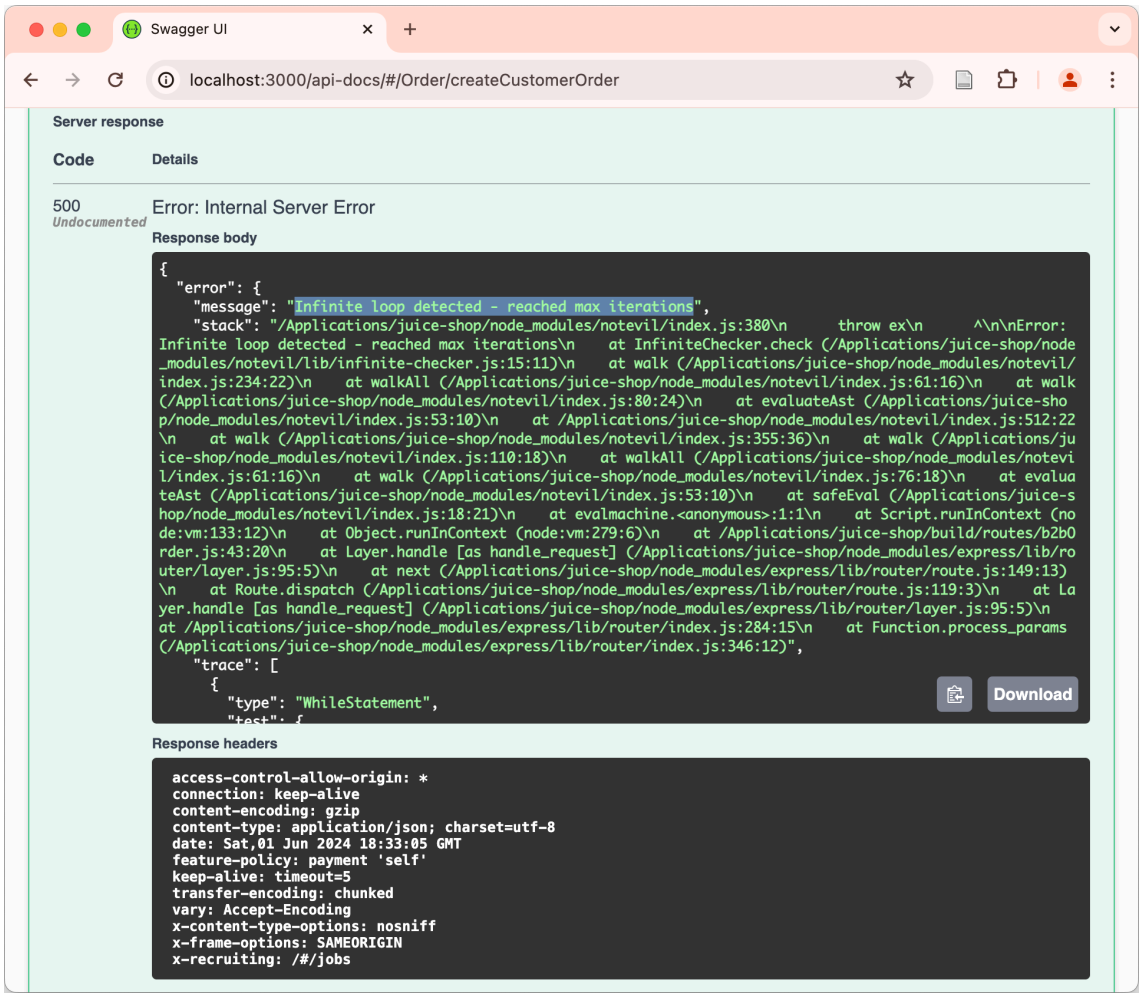


Ilustración 35: Resultado de la ejecución de código arbitrario tras el uso de la API
Fuente: elaboración propia

Se observa, por el mensaje de error, que la aplicación ha ejecutado el bucle infinito y no ha hecho ningún tipo de validación sobre el contenido recibido, a la hora de deserializar las líneas de pedido en objetos dentro del código.

Alucinaciones de paquetes de la IA

El gran auge en el uso de la IA generativa que vivimos actualmente está llevando a los equipos de desarrollo a utilizar herramientas basadas modelos de lenguaje grandes (LLM), tales como ChatGPT y otras, como asistentes en la generación de código.

En su estudio reciente "*Diving Deeper Into AI Package Hallucinations*" [35], Lanyado retoma su estudio anterior [36] sobre las alucinaciones en paquetes propuestas por estos modelos. Una alucinación de un modelo LLM es una información generada por este que, aunque plausible, no es factualmente correcta. En el contexto de ambos estudios, se trata de paquetes de varias plataformas (incluyendo Node.js) que los modelos proponen, pero que no existen realmente.

Estos paquetes inexistentes pueden ser aprovechados por los atacantes de forma que los crean con el mismo nombre propuesto por los LLMs y se suben a los repositorios de paquetes, incluyendo código con fines maliciosos. En el estudio más reciente, Lanyado indica que el paquete que cargó, con el mismo nombre que el de la alucinación del modelo, `huggingface-cli`, había sido descargado treinta mil veces [35].

Las recomendaciones del autor sobre el uso de la información propuesta por los LLMs van en la misma línea que las de OWASP que se han visto en este apartado:

- Cuando un LLM presente una respuesta de la que no se está seguro, realizar una verificación para asegurarnos de que la información es precisa y fiable.
- En general, a la hora de utilizar cualquier software *open source*, especialmente paquetes que no son familiares, visitar la página del repositorio en busca de información del paquete. Revisar el historial de mantenimiento, las vulnerabilidades conocidas, la puntuación, el número de *commits*, la implicación de la comunidad, etc.

5.9. A09:2021 – Fallos en el Registro y Monitorización

Esta categoría es para detectar, escalar y responder a brechas de seguridad. Sin un registro y una monitorización, estas brechas no pueden detectarse. Este registro y monitorización puede ser difícil de comprobar y, a menudo, implica realizar entrevistas con el equipo de desarrollo o preguntar si los ataques se detectaron durante una prueba de penetración.

Hay un registro, detección, monitorización y respuesta insuficientes siempre que:

- No se registran eventos auditables como los inicios de sesión, fallidos o no y transacciones de alto valor.
- Los avisos y los errores generan mensajes que son inadecuados, no son claros o ni siquiera se generan.
- Los registros de aplicaciones y APIs no se monitorizan en busca de actividad sospechosa.
- Los registros solo se almacenan localmente.
- Los umbrales de alerta y los procesos de escalado de respuesta no existen o no son efectivos.
- Los análisis y pruebas de penetración con herramientas de pruebas dinámicas de seguridad de aplicaciones (DAST), tales como OWASP ZAP no lanzan alertas.
- La aplicación no puede detectar, escalar o alertar de ataques activos en tiempo real o casi real.

Formas de evitarlo:

El equipo de desarrollo debería implementar algunos o todos los controles siguientes, en función del riesgo de la aplicación:

- Asegurarse que todos los inicios de sesión, control de acceso y fallos de validación de la entrada en el lado del servidor se pueden registrar con suficiente contexto para identificar cuentas sospechosas o maliciosas. También, comprobar que se mantienen durante el tiempo suficiente como para permitir un análisis forense posterior.
- Asegurarse de que los *logs* se generan en un formato que las soluciones de gestión de *logs* puedan consumir fácilmente.
- Asegurarse de que los datos se codifican correctamente para prevenir inyecciones o ataques en los sistemas de monitorización o de registro.
- Asegurarse de que las transacciones de alto valor tienen una traza de auditoría con controles de integridad que eviten su modificación o el borrado.
- Los equipos de DevSecOps deberían establecer una monitorización y alertas efectivas, de forma que se detecte y responda rápidamente a las actividades sospechosas.
- Establecer o adoptar un plan de recuperación y respuesta, tal como el NIST 800-61r2 o posterior.

Existen *frameworks* de protección de aplicaciones *open-source* y comerciales, tales como el ModSecurity Core Rule Set de OWASP y programas de correlación de logs, como Elasticsearch, Logstash, y la pila Kibana (ELK).

5.10. A10:2021 – Falsificación de Solicitudes del Lado del Servidor (SSRF)

Los fallos SSRF ocurren cuando una aplicación web está recuperando un recurso remoto sin validar la URL suministrada por el usuario. Permite a un atacante forzar a la aplicación a enviar una petición a un destino inesperado, incluso aunque exista protección por firewall, VPN o listas de control de acceso de red.

El equipo de desarrollo puede prevenir las vulnerabilidades de tipo SSRF implementando alguno o todos los controles en profundidad siguientes:

- Desde la capa de red:
    - Segmentar la funcionalidad de acceso a recursos remotos en redes separadas para reducir el impacto de SSRF.
    - Aplicar las políticas de *firewall* de "denegar por defecto" o las reglas de control acceso a la red para bloquear todo el tráfico salvo el necesario.
- Desde la capa de aplicación:
    - Validar y limpiar todos los datos suministrados por el cliente.
    - Hacer cumplir el esquema de la URL, puerto y destino mediante una lista de recursos permitidos.
    - No enviar respuestas en bruto a los clientes.

- Deshabilitar las redirecciones HTTP.
- Ser consciente de la consistencia de la URL para evitar ataques tipo *DNS rebinding* y condiciones de carrera tipo "*time of check, time of use*" (TOCTOU).

No se debe mitigar SSRF con el uso de listas de denegación basadas en expresiones regulares. Los atacantes tienen listas de *payloads*, herramientas y habilidades para saltárselas.

Evitar redirecciones abiertas

Como ejemplo de este tipo de vulnerabilidad, la guía de mejores prácticas de seguridad de Express [9] hace referencia a las redirecciones abiertas. Son aquellas en que una aplicación acepta una URL como entrada del usuario (por ejemplo, en la *querystring*: `?url=http://direccion.com`) y utiliza `res.redirect` para cambiar la dirección de la aplicación devolviendo un código de estado 3xx.

En este caso, para validar la entrada del usuario, se puede comprobar el host de la URL para permitir o no la redirección:

```
if (new Url(req.query.url).host === 'direccion.com') {
  res.redirect(req.query.url)
}
```

6. Consideraciones específicas

Las vulnerabilidades que se acaban de analizar en el apartado anterior afectan a todo tipo de aplicaciones web y no sólo a las aplicaciones web modernas, tal como se han definido al comienzo, que son objetivo de este trabajo.

En este apartado se describen algunas de las vulnerabilidades que son específicas del desarrollo *full-stack* JavaScript y las pilas MExN.

6.1. Node.js [37]

En el apartado "5.3. A03:2021 – Inyección" se han comentado las distintas vulnerabilidades de inyección de código que existen. Algunos de estos ataques de inyección, por ejemplo, *Cross Site Scripting* (XSS), se aplican al código del lado del cliente y tienen como objetivo el propio usuario.

Mientras que otros tipos de inyección son del lado del servidor (como la inyección SQL) y el objetivo es el propio servidor y la aplicación. Este apartado se centra en este tipo de ataques, concretamente en el llamado *Server-Side JavaScript Injection* (SSJI).

A diferencia de un ataque XSS en el lado del cliente, cuyo alcance puede estar limitado por el *sandbox* del navegador web, un ataque SSJI puede potencialmente ejecutar código en un entorno sin restricciones y tener acceso al sistema donde se ejecuta. Y relacionado con la inyección de código, la inyección

de comandos permite la ejecución de comandos del sistema operativo del servidor que aloja la aplicación.

Por tanto, un ataque SSJI es un tipo de ataque de inyección de código que se produce en el código JavaScript que se ejecuta en el servidor. El objetivo es inyectar código JavaScript arbitrario que se ejecute en la aplicación. Como en el resto de las vulnerabilidades de este tipo, se producen cuando las aplicaciones aceptan, sin ningún tipo de validación o filtrado, datos originados por el usuario que luego se procesan dinámicamente por un intérprete de JavaScript.

En el caso concreto de Node.js, algunas de las funciones que pueden introducir estos problemas de seguridad son:

- `eval()`, que recibe una cadena como parámetro y la interpreta como JavaScript.
- `Function()`, que crea funciones dinámicamente.
- Las funciones del módulo `ChildProcess`, como `exec()`, `execfile()`, `spawn()` y `fork()`, que permiten a una aplicación Node.js acceder a los recursos de bajo nivel del SO que aloja la aplicación. Estas funciones no están disponibles en el JavaScript tradicional del cliente.
- Las funciones del módulo `fs` como `readFileSync()` y `writeFileSync()` que leen y escriben a fichero, respectivamente.

Hay varias formas en la que un atacante puede explotar este tipo de vulnerabilidades:

1. Denegación de Servicio (DoS):
   Si el atacante consigue inyectar en la aplicación la línea de código `while(true)`, forzará al servidor web a usar toda su CPU para el bucle infinito y no podrá atender a las peticiones de otros clientes. Comparado con ataques de DoS clásicos, este es mucho más efectivo para el atacante, que no tiene que enviar a la víctima millones de peticiones.

   En el apartado "5.8. A08:2021 – Fallos en el Software y en la Integridad de los Datos" se ha visto un ejemplo de cómo Juice Shop es vulnerable a este tipo de inyección.

2. Acceso al sistema de ficheros:
   El atacante puede inyectar una función del módulo `fs` para conseguir acceso al sistema de ficheros del servidor web. Por ejemplo, con el *payload*: `response.end(require('fs').readFileSync('/etc/passwd'))`.

3. Creación y ejecución de ficheros binarios:
   Además de leer ficheros, el atacante podría escribir ficheros. Por ejemplo, mediante el código:

   ```
   require('fs').writeFileSync(nombreFichero, datos, 'base64')
   ```

   Podría enviar un programa malicioso codificado en Base64 para después darle permisos de ejecución:

```
require('fs').chmodSync(nombreFichero, '755')
```

Y, por último, ejecutarlo:

```
require('child_process').exec(nombreFichero)
```

4. Shell inversa: mediante el uso de las funciones `spawn()`, `connect()` y `pipe()` del módulo `net`.

Ntantogian et al., autores del estudio de referencia en este apartado, presentan la herramienta *open source* NodeXP (NOde.js JavaScript injection vulnerability DEtection and eXPloitation). Escrita en Python, implementa una metodología para automatizar la detección y explotación de vulnerabilidades SSJI en aplicaciones Node.js.

Mediante el uso de diversos vectores de ataque, intenta inyectar estos secuencialmente en campos potencialmente vulnerables (parámetros GET o POST) y evalúa la respuesta HTTP para detectar si la aplicación ha ejecutado el vector y es vulnerable a SSJI, o no. En caso de serlo, el módulo de explotación intentará ataques de enumeración de directorios, acceso a ficheros sensibles y carga o ejecución de ficheros.

El estudio compara la herramienta NodeXP frente a otras herramientas de escaneo de aplicaciones web[11], buscando vulnerabilidades SSJI en varias aplicaciones Node.js deliberadamente vulnerables[12]. Como resultado, ZAP no detectó ninguna de las vulnerabilidades y Burp Suite detectó cuatro de las cinco.

Por tanto y debido a la gravedad de los ataques que permite la vulnerabilidad SSJI, se propone la herramienta NodeXP como complemento a las herramientas analizadas en el trabajo, especialmente si sólo se utiliza ZAP. Pero, por limitaciones de tiempo, no se ha podido evaluar la herramienta frente a Juice Shop.

Como mejor práctica a la hora de prevenir este tipo de ataques, se recomienda evitar pasar la entrada del usuario a funciones que ejecuten código JavaScript de manera dinámica en el lado del servidor, como las comentadas `eval()` o `Function()`.

Si es necesario utilizar dicha entrada del usuario, el equipo de desarrollo se debe asegurar de validar la entrada, filtrando caracteres peligrosos. Esto puede hacerse mediante listas negras (eliminando caracteres tales como el `&`, `'`, `;`, etc.) o mediante listas blancas (aceptar las entradas que coinciden con patrones predefinidos considerados seguros).

Node.js dispone de varios paquetes que permiten validar la entrada del usuario, por ejemplo, `validator`, que permite reducir el esfuerzo de crear esos filtros.

[11] Acunetix, Burp Suite, ZAP, Vega y W3af.
[12] Nodegoat, Express TestBench, XVNA (Extreme Vulnerable Node Application), node.nV y Appsecco.

6.2. Angular [12]

Los ataques de tipo Cross-site scripting (XSS) permiten a un atacante inyectar código malicioso en las páginas web. Son, por tanto, ataques de inyección de código similares a los vistos en la sección "5.3. A03:2021 – Inyección" y permiten, por ejemplo, robar datos como el usuario o la contraseña y realizar acciones que suplanten al usuario. Son uno de los ataques más comunes en la web.

En el caso de Angular y *frameworks* similares, se debe evitar que llegue código malicioso al modelo de objetos de la aplicación (DOM: *Document Object Model*). Por ejemplo, si el atacante consigue insertar una marca `<script>` en el DOM, logrará ejecutar código arbitrario en la aplicación.

Para mitigar este tipo de vulnerabilidades, Angular considera como no confiable cualquier valor que vaya a insertar en DOM desde el enlace de una plantilla o la interpolación de cadenas. Por tanto, limpia y escapa los valores que no son de confianza, eliminando, por ejemplo, el elemento `<script>`. También se le puede indicar a Angular que el valor sí es de confianza si ha sido limpiado previamente.

Además de esta limpieza automática que realiza Angular, se pueden utilizar APIs específicas para ello dentro del módulo `DomSanitizer`. Concretamente, llamando al método `sanitize` con el contexto de seguridad apropiado. Angular dispone de varios contextos, pues lo que puede ser seguro para uno (por ejemplo, CSS) puede no serlo para otro.

Por otra parte, para evitar la limpieza automática cuando se sabe que se confía en un valor, se pueden utilizar los métodos `bypassSecurityTrustHtml`, `bypassSecurityTrustScript`, `bypassSecurityTrustStyle`, `bypassSecurityTrustUrl` y `bypassSecurityTrustResourceUrl`, según el contexto de seguridad también.

De manera similar a Angular, Vue escapa por defecto las interpolaciones de cadenas en las plantillas y los enlaces de atributos dinámicos [13].

En la Juice Shop podemos provocar un ataque de XSS desde el campo de búsqueda de productos si introducimos este código: `<iframe src="javascript:alert('hola')">`:

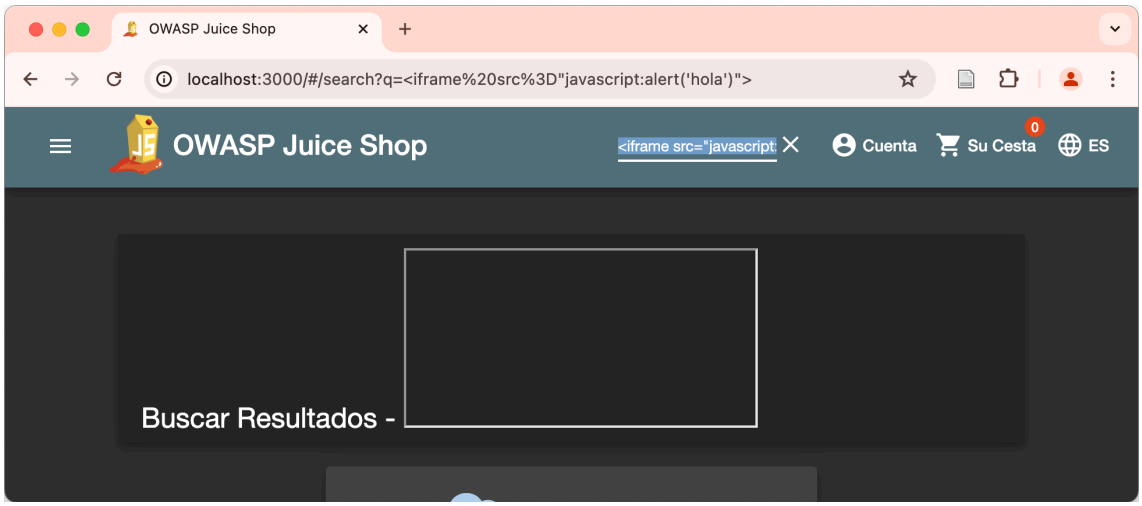


Ilustración 36: Ataque XSS contra la búsqueda de productos en Juice Shop
Fuente: elaboración propia

En la figura se observa que se ha construido un `iframe` junto a "Buscar Resultados" y que el navegador ejecuta el código JavaScript, pues se muestra la alerta:

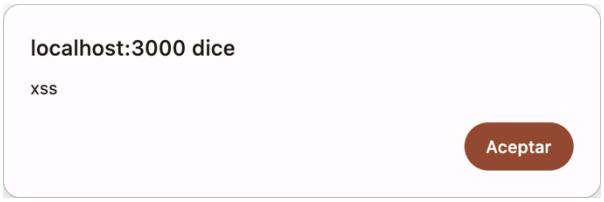


Ilustración 37: Ejecución del código JavaScript del ataque XSS
Fuente: elaboración propia

Y, si observamos el DOM con las herramientas de desarrollo del navegador, vemos que se ha introducido el código vulnerable sin ninguna validación:

```
▼<span _ngcontent-dyb-c46 id="searchValue">
  ▼<iframe src="javascript:alert('xss')"> == $0
    ▼#document (about:blank)
      ▼<html>
          <head></head>
          <body></body>
        </html>
    </iframe>
  </span>
</div>
```

Ilustración 38: DOM modificado tras el ataque XSS
Fuente: elaboración propia

Se debe hacer notar que la aplicación Juice Shop se construye de manera deliberadamente insegura y tiene fines educativos. Si se observa el código fuente que gestiona la búsqueda, se observa que se han utilizado las funciones comentadas con anterioridad que indican a Angular que la entrada se considera segura:

```
  this.dataSource.filter = queryParam.toLowerCase()
  this.searchValue =
    this.sanitizer.bypassSecurityTrustResourceUrl(queryParam)
```

Content-Security-Policy

Ya se ha aludido a la cabecera Content-Security-Policy en el apartado "5.2. A02:2021 – Fallos Criptográficos" al hablar del complemento Helmet para Express. Esta cabecera puede ayudar a prevenir los ataques de tipo XSS.

La guía de mejores prácticas de Angular propone también el uso de esta cabecera e incluso sugiere la configuración mínima para una aplicación Angular nueva:

```
default-src 'self'; style-src 'self' 'nonce-randomNonceGoesHere';
script-src 'self' 'nonce-randomNonceGoesHere';
```

`default-src 'self';` permite que la página cargue los recursos que necesita desde el mismo origen. `style-src 'self' 'nonce-randomNonceGoesHere';` permite a la página cargar los estilos desde el mismo origen (`'self'`) y los estilos insertados por Angular con `nonce-randomNonceGoesHere`. Por último, `script-src 'self' 'nonce-randomNonceGoesHere';` permite que Angular cargue JavaScript desde el mismo origen y los scripts insertados con `nonce-randomNonceGoesHere`.

El *nonce* es un número aleatorio que se debe incluir en la cabecera en cada petición. Este número debe ser único por petición y no puede ser predecible, de forma que un atacante no pueda saltarse las protecciones de la cabecera CSP.

Cross-Site Request Forgery (XSRF)

Angular tiene mecanismos para evitar vulnerabilidades de tipo CSRF o XSRF (*cross-site request forgery*).

Con CSRF un atacante puede engañar al usuario para que visite una página diferente a la de la aplicación y que esa página envíe una petición maliciosa al servidor de la aplicación web. Si el usuario tiene la sesión iniciada en la aplicación web y no está protegida contra CSRF, el atacante podría ejecutar acciones en nombre del usuario. Por ejemplo, una transferencia bancaria si se trata de una aplicación de banca.

La técnica común para prevenir este tipo de ataques es que el servidor envíe un token aleatorio de autenticación en una cookie. El código cliente lee la cookie y añade una cabecera en las peticiones siguientes que incluye el valor que ha recibido. El servidor puede después comparar los valores y, si coinciden, verificar que se trata del cliente legítimo ya que, por la política del mismo origen solo el código del sitio web legítimo podría leer esa cookie. Este mecanismo está implementado en Angular, con la cookie `XSRF-TOKEN` y la cabecera `X-XSRF-TOKEN`.

6.3. MongoDB

En el apartado "5.3. A03:2021 – Inyección" se llevó a cabo un ataque de inyección SQL clásica contra la página de inicio de sesión de Juice Shop. Incluso se pudo ver, en la "Ilustración 25: Resultado del Fuzzer en ZAP." cómo la propia sentencia SQL empleada por la aplicación se filtraba como error en la respuesta.

Si observamos el código fuente de la aplicación web:

```
models.sequelize.query(`SELECT * FROM Users
  WHERE email = '${req.body.email || ''}'
  AND password = '${security.hash(req.body.password || '')}'
  AND deletedAt IS NULL`, { model: UserModel, plain: true })
```

Se observa que se está concatenando con la consulta el valor del correo electrónico que se recibe del cliente (`req.body.email`), sin ningún tipo de validación o filtro.

Además de SQLite, la aplicación Juice Shop también utiliza MarsDB, un SGDB NoSQL derivado de MongoDB. Este tipo de BD también son vulnerables a ataques de inyección, aunque no utilicen el lenguaje SQL. Incluso al ejecutarse dentro de un lenguaje procedimental en lugar de declarativo, como SQL, pueden tener mayores impactos que la inyección SQL clásica [38].

En la Juice Shop podemos explotar un ejemplo de inyección NoSQL [26]. Para ello, capturamos el tráfico con Burp Suite a la hora de modificar la reseña de un producto que hayamos hecho con anterioridad:

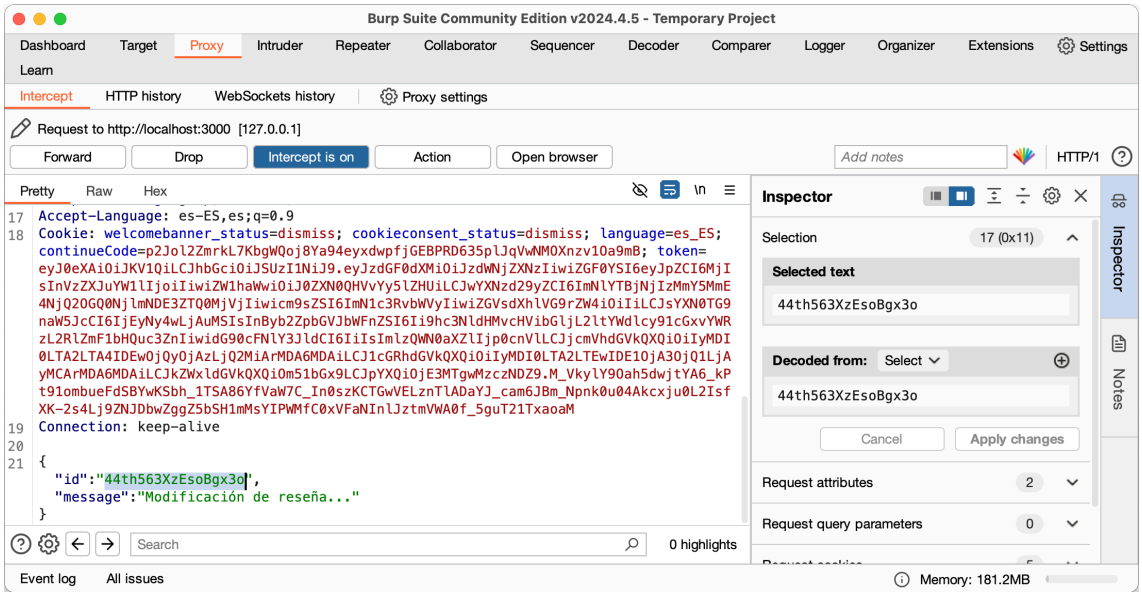


Ilustración 39: Captura en Burp Suite de la petición de modificación de una reseña
Fuente: elaboración propia

Se trata de una petición `PATCH` al *endpoint* `/rest/products/reviews`. Como se observa en el cuerpo de la petición, el mensaje es un JSON con el `id` de la reseña a actualizar y el nuevo mensaje. Fijándonos en los selectores de consulta de

MongoDB[13], podemos manipular en Burp Suite el cuerpo del mensaje de la siguiente manera:

```
{"id": { "$ne": -1 },"message":"Producto muy bueno"}
```

Es decir, se introduce un filtro a la consulta para que busque las reseñas cuyo `id` sea distinto de -1 y se cambia el mensaje de la reseña.

Si se pulsa el botón "Forward" para enviar la petición `PATCH` actualizada, puede observarse en la tienda que se han cambiado las reseñas de todos los productos:

Ilustración 40: Ejemplo de reseña modificada tras la inyección NoSQL
Fuente: elaboración propia

Si se observa el código fuente de la Juice Shop encargado de la actualización de reseñas:

```
db.reviewsCollection.update(
  { _id: req.body.id },
  { $set: { message: req.body.message } },
  { multi: true }
)
```

Se aprecia que, aunque se esté utilizando un ORM, el descuido en el desarrollo ha hecho que se pase como filtro del parámetro `id` una entrada del usuario sin validar ni filtrar.

[13] https://www.mongodb.com/docs/manual/reference/operator/query/#query-selectors

Para mitigar este tipo de ataques, además de nunca utilizar la entrada del usuario sin filtrar o validar, es posible deshabilitar la ejecución de JavaScript en el lado del servidor en una instancia MongoDB [39]. Si se usan, se debe tener especial cuidado con las expresiones que permiten ejecutar código JavaScript en el servidor: `$where`, `mapReduce`, `$accumulator` y `$function`.

7. Metodología de análisis de aplicaciones web

La guía de OWASP sobre las principales vulnerabilidades de las aplicaciones web es un recurso de referencia en el desarrollo web seguro. La fundación dispone de muchos más recursos y herramientas destinadas a mejorar la seguridad, no solo de aplicaciones web, sino de otras tecnologías.

Otros recursos interesantes respecto a las aplicaciones web y que complementan el OWASP Top 10 son:

- La guía de pruebas de seguridad web (OWASP Web Security Testing Guide). Es un *framework* de mejores prácticas para la prueba de la seguridad de aplicaciones y servicios web. Se crea con el esfuerzo colaborativo de profesionales de la ciberseguridad y la utilizan *pentesters* y empresas por todo el mundo [40].
- OWASP Cheat Sheet Series: siguiendo el formato de guía rápida (*chuleta*), ofrece una colección de información concisa sobre ciertos temas de seguridad de aplicaciones [41]. Incluso existe una sección que indica, para cada categoría del OWASP Top 10 que se ha repasado, la *chuleta* con información para aplicar directa y concretamente las recomendaciones que da para cada vulnerabilidad.

Toda esta información, aunque muy valiosa, puede resultar abrumadora para un pequeño equipo de desarrollo que aún no puede dedicar medios a analizar y mejorar la ciberseguridad de sus aplicaciones web.

Hay algunos aspectos, como se ha visto en alguna de las vulnerabilidades del Top 10 de OWASP, que pueden requerir una revisión manual del código de la aplicación o entrevistas con el equipo de desarrollo. Incluso puede resultar necesario plantearse un cambio de mentalidad o cultura en la empresa para adoptar ciertas prácticas o un ciclo de desarrollo del software seguro.

Pero, como se ha comentado también, los programas de análisis dinámicos de seguridad de aplicaciones web, como OWASP ZAP y Burp Suite, pueden ser el primer paso para detectar ciertas vulnerabilidades típicas de las que el equipo de desarrollo puede no ser consciente.

Por ejemplo, en OWASP ZAP, el escaneo automatizado dispone de reglas específicas para siete de los diez riesgos del Top 10[14]. Lo que se propone aquí es incorporar este escaneo al flujo de trabajo del equipo de desarrollo, de manera

[14] Son: A01, A02, A03, A04, A05, A06 y A08. Fuente: ZAP - ZAPping the OWASP Top 10 (2021).

que se puedan detectar las vulnerabilidades más importantes antes de desplegar una aplicación web.

Para evitar el trabajo manual que supondría este análisis previo, se utilizarán las capacidades de automatización de las herramientas para integrar ese escaneo en un flujo de trabajo de integración/entrega continua (CI/CD).

7.1. Jenkins

Jenkins es un servidor de automatización *open source* que se utiliza para flujos de trabajo de CI/CD [42]. Se propone el uso de esta herramienta, debido a su popularidad, pues tendría en 2023 una cuota de mercado del 44%[15], y por su facilidad de uso. Si nuestro equipo de desarrollo no utiliza este tipo de herramientas, puede comenzar a hacerlo de manera sencilla con Jenkins. Y, si utiliza otra herramienta, es fácil adaptar lo que se propone aquí a la herramienta empleada.

Se propone crear un *pipeline* en Jenkins que incluya, como parte de sus fases:

- Un escaneo automatizado junto a un ataque activo contra la aplicación web desplegada en un entorno de desarrollo, antes de su despliegue a producción. Se podrá configurar el nivel de riesgo que se quiere asumir en las vulnerabilidades que se detecten. El flujo de trabajo puede fallar, por ejemplo, si se encuentra alguna vulnerabilidad clasificada como tipo Alto y evitar desplegar una aplicación insegura.
- Un escaneo automatizado de los paquetes Node.js empleados, detectando si nuestra aplicación utiliza algún paquete vulnerable y evitando, como en el caso anterior, el despliegue de la aplicación si se detectan vulnerabilidades de tipo crítico o alto.

7.2. Configuración de la automatización con ZAP

OWASP ZAP dispone de un *framework* para la automatización, de forma que se puede ejecutar su funcionalidad desde fuera de la aplicación. Se utilizará dicho *framework* para integrar el escaneo en un pipeline de CI/CD [43].

El primer paso para ello es definir un contexto, donde configuraremos varias opciones del análisis. En este caso, se crea un contexto "Juice Shop" y se define, en primer lugar, las URLs que se procesarán, en este caso, http://localhost:3000/:

[15] Fuente: Jenkins Project Reports Growth of 79% in Jenkins Pipeline, Used to Speed Software Delivery - CD Foundation

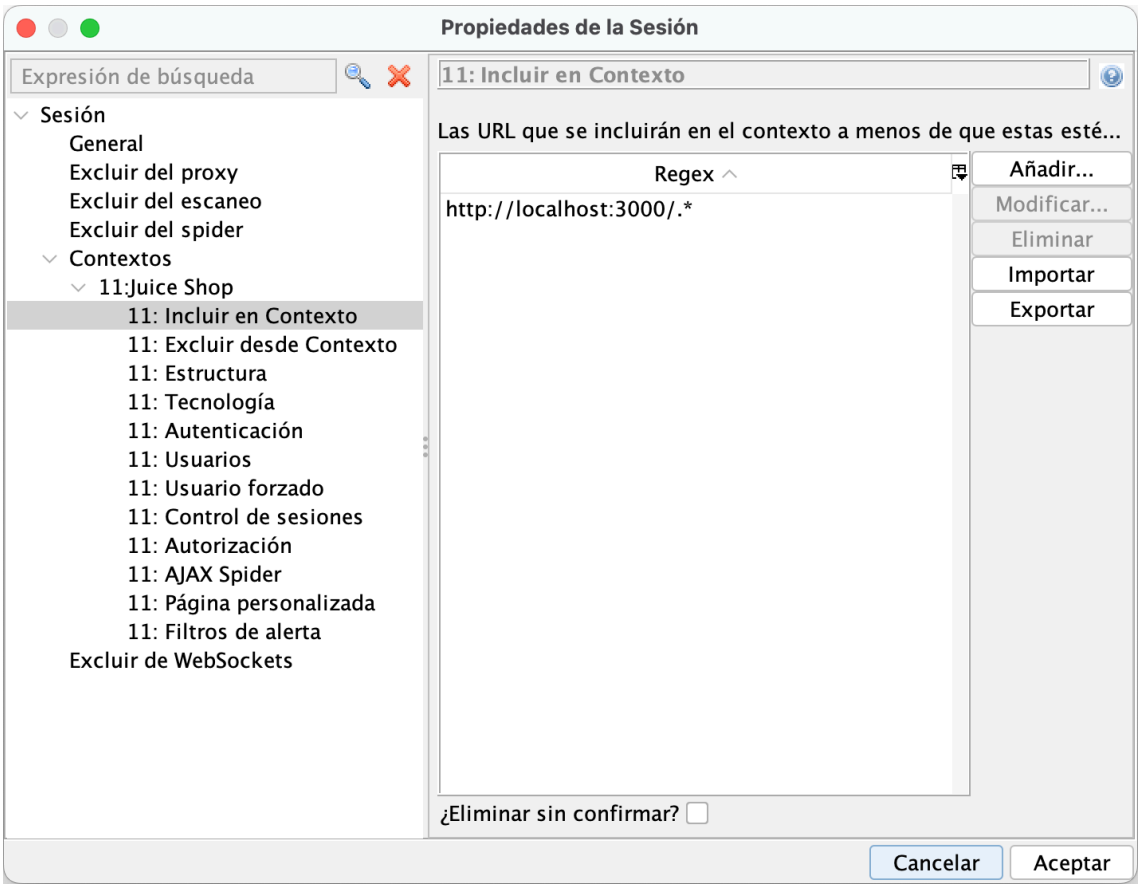


Ilustración 41: Configuración de URLs en el contexto
Fuente: elaboración propia

En la pestaña "Tecnología", dejamos marcadas únicamente aquellas que utiliza nuestra aplicación, pues ahorrará tiempo en el análisis activo, evitando que se intentes explotar vulnerabilidades servidores, *frameworks* o sistemas de BD que no se emplean:

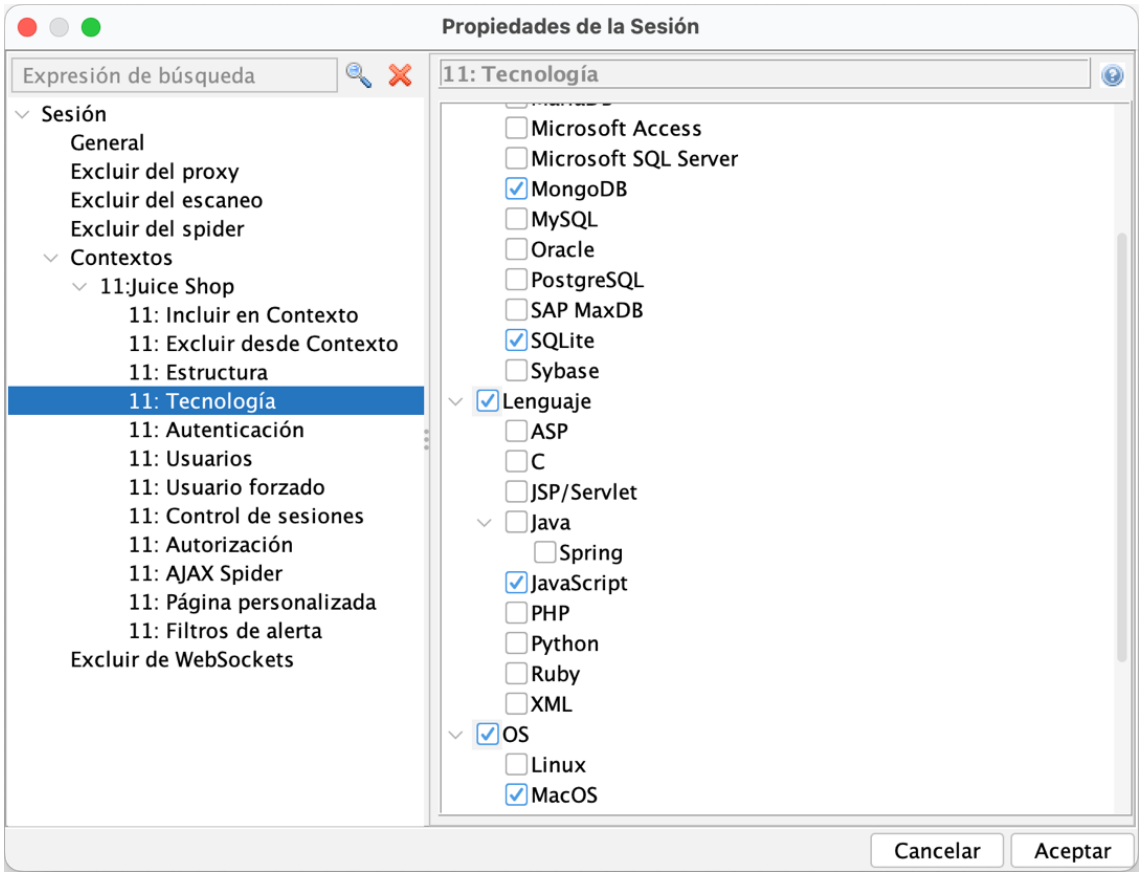


Ilustración 42: Configuración de tecnologías utilizadas en nuestra aplicación
Fuente: elaboración propia

La pestaña de "Autenticación" es importante ya que gran parte de la funcionalidad de la aplicación se encuentra disponible sólo para usuarios con cuenta en la aplicación. Cuando ZAP realice el análisis activo deberá poder iniciar sesión de manera automática para acceder a esas partes y poder buscar las vulnerabilidades.

Como se ha visto en puntos anteriores, podemos analizar con ZAP una petición de autenticación en la aplicación. Se trata de una petición `POST` al *endpoint* del servicio REST http://localhost:3000/rest/user/login pasando como cuerpo de la petición el JSON `{"email":"usuario","password":"contraseña"}`.

Se configura el tipo de autenticación como "Autenticación basada en JSON" [44] como método de autenticación en la pestaña mencionada:

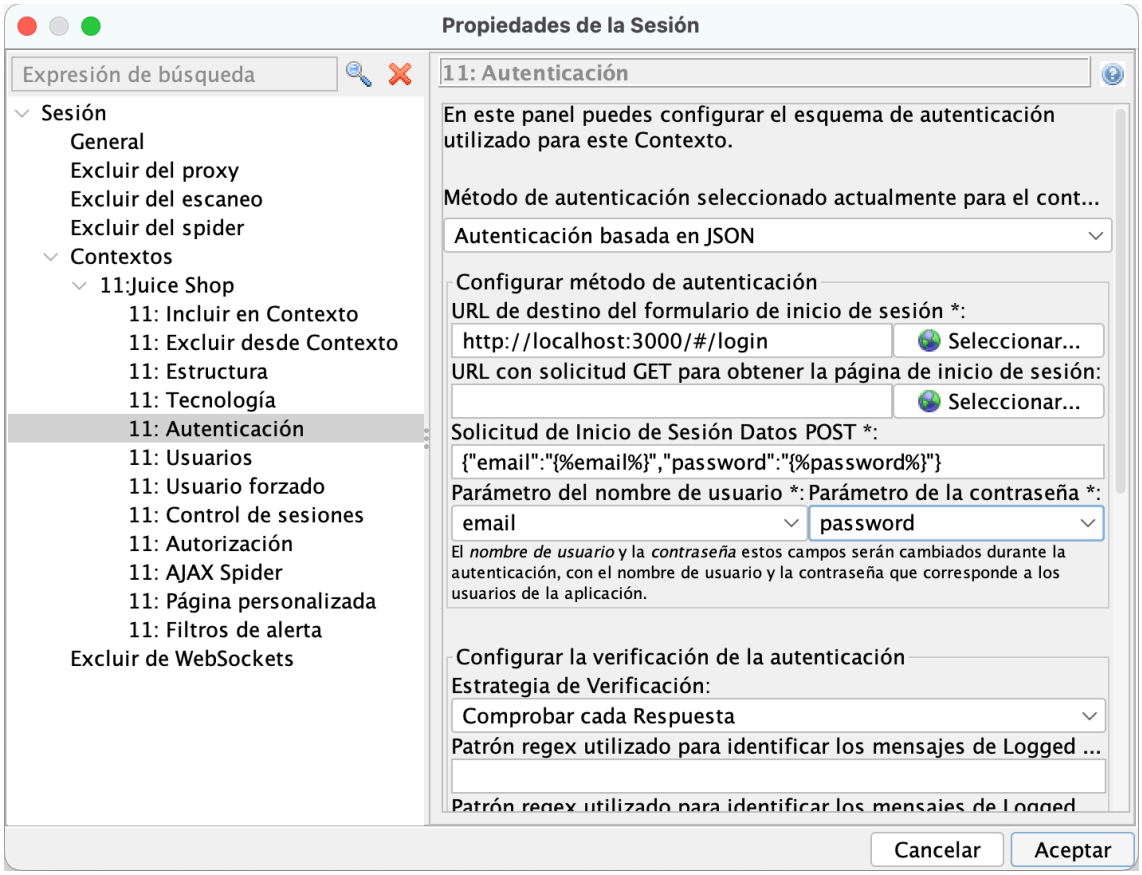


Ilustración 43: Configuración de las opciones de autenticación del contexto
Fuente: elaboración propia

En la sección "Usuarios" se configurará un usuario que hayamos dado de alta en la aplicación para que ZAP inicie sesión con este en la aplicación.

Una vez configurado el contexto del análisis, nos colocamos en la sección "Automatización" de la mitad inferior de ZAP y añadimos un nuevo plan.

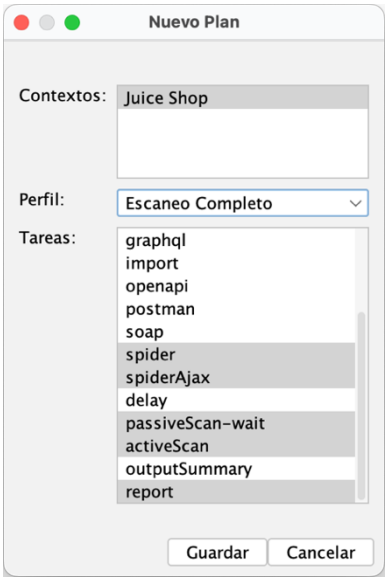


Ilustración 44: Nuevo plan de automatización
Fuente: elaboración propia

En este, se selecciona el contexto que se acaba de crear y se elige "Escaneo Completo" como perfil. La ventana muestra las tareas que se ejecutarán como parte del plan, que incluye escaneos pasivos con los *spider* normal y Ajax y un análisis activo atacando la aplicación.

El plan creado se muestra en la sección de "Automatización":

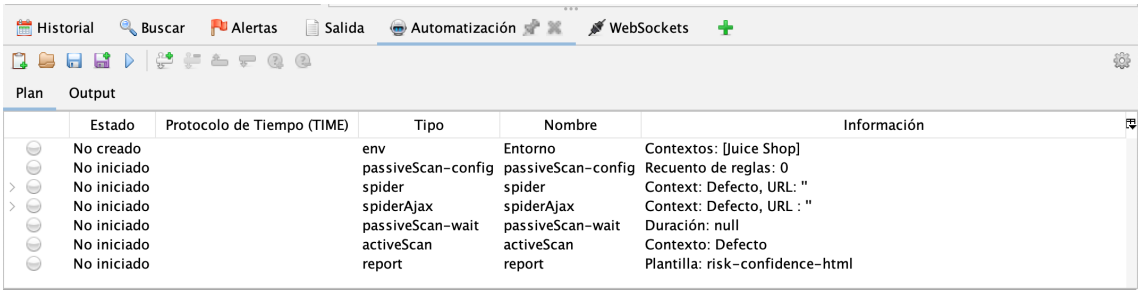


Ilustración 45: Plan de automatización recién creado
Fuente: elaboración propia

Una vez generado, se puede guardar el plan en formato YAML y esta será la configuración que se aplicará cuando se ejecute ZAP de manera automatizada desde el pipeline de Jenkins.

7.3. Integración del escaneo de ZAP en Jenkins

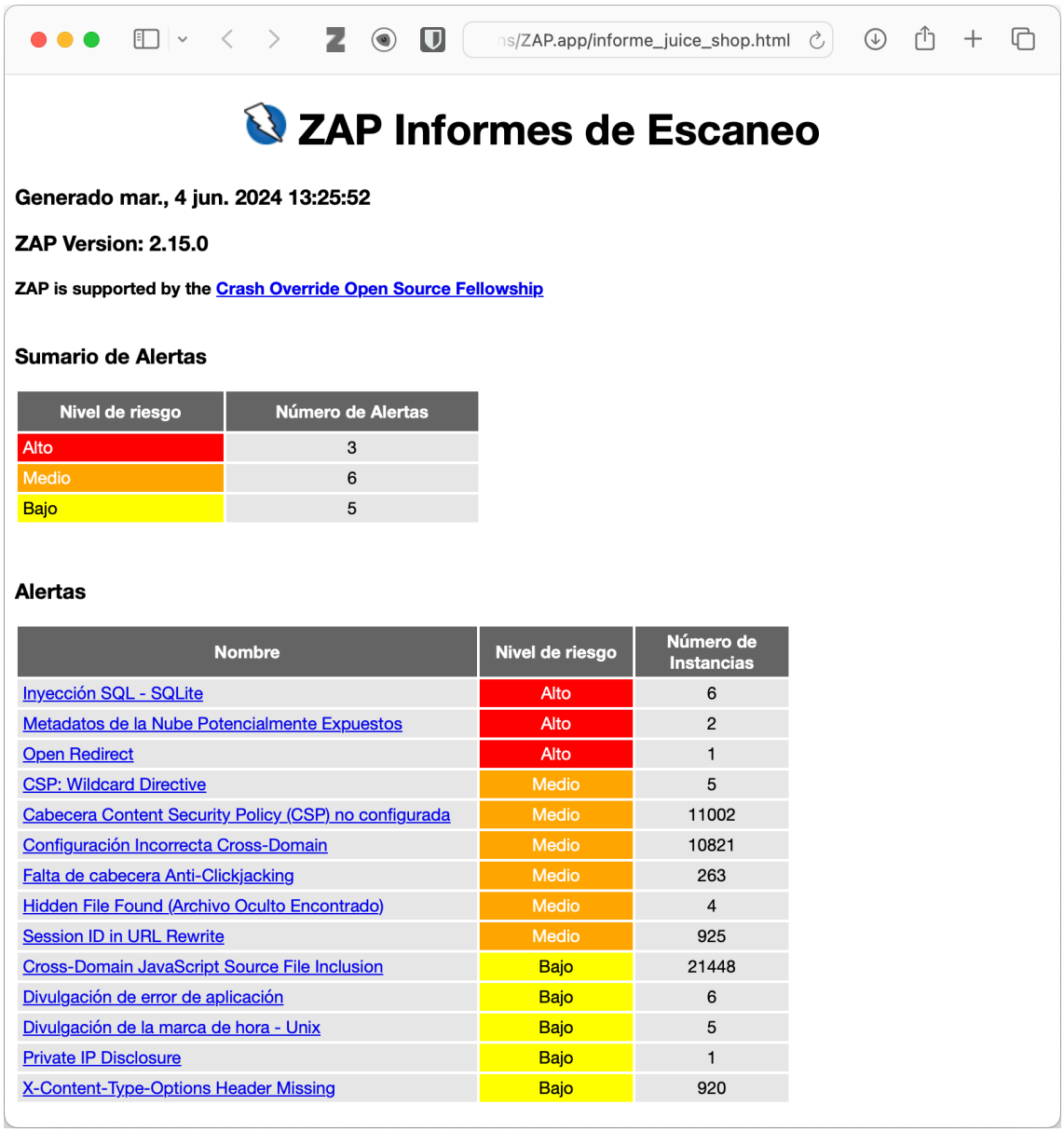
Se supone que existe ya en Jenkins una tarea de tipo *pipeline* para construir la aplicación web, probarla y desplegarla a un entorno, por ejemplo, de integración.

Para incorporar al proceso el escaneo mediante ZAP, se crea una nueva etapa en el *script* del *pipeline*. Por ejemplo:

```
stage('ZAP: Análisis de seguridad') {
  steps {
    sh '/Applications/ZAP.app/Contents/Java/zap.sh -cmd
       -autorun /Applications/JuiceShopAutomation.yaml'
  }
}
```

El paso consiste en ejecutar un comando (`sh`), en este caso, `zap.sh`, que se ejecuta con el parámetro `-autorun` indicando la ruta del fichero de configuración YAML que se ha creado en el paso anterior. Con la opción `-cmd` se indica que se devolverá 0 en la línea de comandos si el plan se ejecuta correctamente [45].

Pero la ejecución correcta del plan no significa que no se hayan encontrado vulnerabilidades en la aplicación. Para detectarlas, se debe analizar el informe que se genera en el último paso del plan, en este ejemplo, en HTML. Se debe buscar el texto que indique que se han encontrado vulnerabilidades importantes.



ZAP Informes de Escaneo

Generado mar., 4 jun. 2024 13:25:52

ZAP Version: 2.15.0

ZAP is supported by the Crash Override Open Source Fellowship

Sumario de Alertas

Nivel de riesgo	Número de Alertas
Alto	3
Medio	6
Bajo	5

Alertas

Nombre	Nivel de riesgo	Número de Instancias
Inyección SQL - SQLite	Alto	6
Metadatos de la Nube Potencialmente Expuestos	Alto	2
Open Redirect	Alto	1
CSP: Wildcard Directive	Medio	5
Cabecera Content Security Policy (CSP) no configurada	Medio	11002
Configuración Incorrecta Cross-Domain	Medio	10821
Falta de cabecera Anti-Clickjacking	Medio	263
Hidden File Found (Archivo Oculto Encontrado)	Medio	4
Session ID in URL Rewrite	Medio	925
Cross-Domain JavaScript Source File Inclusion	Bajo	21448
Divulgación de error de aplicación	Bajo	6
Divulgación de la marca de hora - Unix	Bajo	5
Private IP Disclosure	Bajo	1
X-Content-Type-Options Header Missing	Bajo	920

Ilustración 46: Informe HTML resultado de la ejecución automatizada
Fuente: elaboración propia

Se podría establecer, por ejemplo, que el trabajo de Jenkins falle si se encuentran alertas con nivel de riesgo Alto o Medio. En ese caso, en el HTML del informe aparecerían las marcas `<div>Alto</div>` y `<div>Medio</div>`.

Con la siguiente etapa en el pipeline de Jenkins se podría detectar este hecho:

```
stage('Evaluación del informe') {
    steps {
        script {
            def reportFile =
            readFile('/Applications/ZAP.app/informe_juice_shop.html')

            // Definir las cadenas a buscar en el informe HTML
            def vuln = ['<div>Alto</div>', '<div>Medio</div>']
            def encontrado = false
```

```
            // Buscar las cadenas en el informe HTML
            vuln.each { vulnerability ->
                if (reportFile.contains(vulnerability)) {
                    encontrado = true
                }
            }

            // Cambiar el estado del pipeline si se encuentran
            // vulnerabilidad de riesgo medio o alto
            if (encontrado) {
                error "Se han encontrado vulnerabilidades de tipo
                       medio o alto en el informe de ZAP"
            }
        }
    }
}
```

De esta forma cuando se ejecute el trabajo en Jenkins, si el informe incluye este tipo de alertas, se marcará el *build* como erróneo:

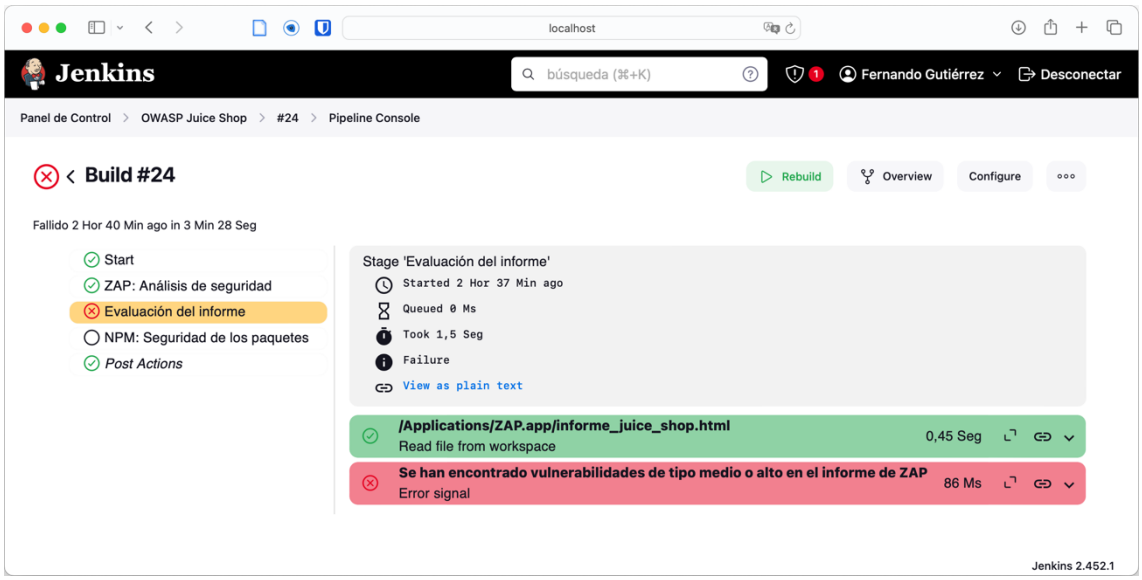


Ilustración 47: *Build* fallido tras el fallo en el paso de evaluación del informe ZAP
Fuente: elaboración propia

7.4. Integración del escaneo de paquetes `npm`

En el apartado "5.8. A08:2021 – Fallos en el Software y en la Integridad de los Datos", se ha comentado sobre la importancia de incorporar al desarrollo módulos y bibliotecas que sean de confianza. El gestor de paquetes de Node.js, `npm`, tiene su propio comando para evaluar la seguridad de los módulos que incorpora un proyecto, en busca de vulnerabilidades conocidas.

Para ello se utiliza el comando `npm audit`[16]. Si no se encuentra ninguna vulnerabilidad, el comando retorna 0. Esto es útil para automatizar también este paso en el pipeline de Jenkins.

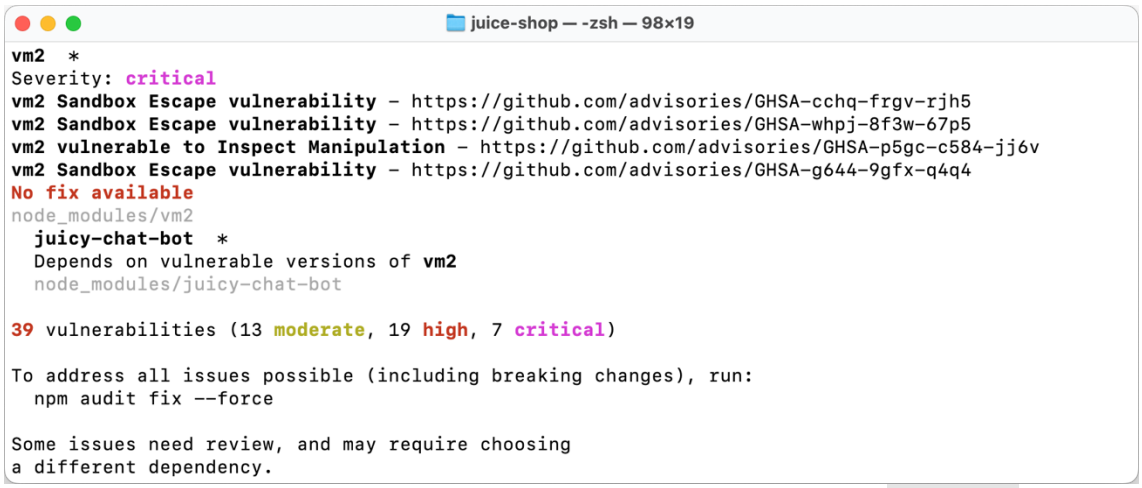


Ilustración 48: Evaluación de la seguridad de los paquetes con `npm audit`
Fuente: elaboración propia

En el ejemplo anterior, se ha lanzado el comando sobre la Juice Shop y se observa que tiene 7 vulnerabilidades críticas y 19 altas. En este caso, se muestran incluso paquetes para los que no existe corrección en ese momento.

Para incorporar esta comprobación en el *pipeline* de Jenkins, introducimos este paso en el *script*:

```
stage('NPM: Seguridad de los paquetes') {
  steps {
    sh 'npm audit'
  }
}
```

Cuando se ejecute esta etapa dentro de Jenkins, el *build* fallará si se detectan vulnerabilidades en los paquetes, evitando el despliegue de una aplicación insegura.

La guía de mejores prácticas de seguridad de Express [9] propone el uso de Snyk[17], que también dispone de una base de datos de vulnerabilidades de dependencias. El uso sería similar a `npm audit`, siendo el comando equivalente, `snyk test`.

[16] Fuente: npm-audit | npm Docs
[17] https://snyk.io

8. Conclusiones

Objetivos principales

El primer objetivo principal del trabajo presentado era diseñar una metodología de análisis de seguridad de aplicaciones web modernas. Para medir el grado de éxito de este objetivo, hay que tener en cuenta el ámbito en que se ha enmarcado el trabajo y que se ha mencionado a lo largo de este: el de una pyme de desarrollo de software sin un presupuesto específico de ciberseguridad o formación específica en la materia. Estos dos factores determinan el tipo de metodología que se puede aplicar, haciendo que deba ser sencilla, o de lo contrario el equipo de desarrollo no la podrá asumir y no la implementará.

Por mi experiencia en este tipo de empresas, pienso que se ha conseguido muy bien este objetivo principal, dado el ámbito y las restricciones comentadas. Ciertamente, en un ciclo de vida de desarrollo de software donde no se tiene en cuenta la ciberseguridad en ninguna de las fases, añadir las comprobaciones de seguridad descritas a un flujo de CI/CD es una mejora importante en comparación con la situación anterior de no tener nada.

Para elaborar este trabajo y la metodología ha sido de especial importancia disponer de los recursos de la fundación OWASP, ya sea en forma de documentación, aplicaciones vulnerables para evaluar la seguridad de las estas y las herramientas *open source* para hacer tal análisis. Nuevamente, recursos valiosos para nuestra empresa objetivo. La dificultad ha sido, precisamente, decidir qué información presentar y cuál descartar por limitaciones de espacio. Debiendo atender, como no podía ser de otra manera, a las vulnerabilidades más frecuentemente explotadas en la actualidad.

El segundo objetivo principal consistía en aplicar la metodología desarrollada para detectar las vulnerabilidades de aplicaciones web concretas. En el trabajo se ha presentado una única aplicación vulnerable, Juice Shop de OWASP, ya que encaja en cuanto a tecnología con lo que se ha establecido aquí como aplicación web moderna e incluye todas las vulnerabilidades más frecuentes.

La metodología que se ha presentado tiene un alto grado de automatización. Aplicarlo a otra aplicación web distinta es tan sencillo como cambiar en el código del *pipeline* de Jenkins el repositorio de código fuente donde está almacenada y la URL del entorno donde se despliega. Por eso, aunque por limitaciones de tiempo y espacio sólo se haya podido analizar una aplicación web, se puede considerar cumplido este segundo objetivo principal.

Objetivos secundarios

El primero de los objetivos secundarios era investigar cuáles eran en la actualidad los *frameworks* de desarrollo web y los lenguajes de programación que utilizan. El segundo objetivo, analizar qué características de seguridad incorporan, frente a las que hay que implementar aparte.

Para el primer objetivo, ha resultado de ayuda encontrar el estudio comparativo sobre popularidad de *frameworks* de desarrollo web citado en el segundo capítulo. Eso ha permitido centrar el trabajo en el desarrollo *full-stack* JavaScript con la pila MExN. Para el segundo objetivo, ha sido útil referenciar las guías de mejores prácticas de seguridad de los distintos *frameworks*, como Node.js, Express, Angular o MongoDB.

El reto para estos dos objetivos ha sido familiarizarse con los aspectos modernos de JavaScript ya que, por mi experiencia profesional, había desarrollado en JavaScript hace más tiempo, cuando se utilizaba principalmente en el cliente para dotar de interactividad a las páginas web. Para el segundo objetivo, el reto ha sido encajar cada una de las propuestas de seguridad de las guías de mejores prácticas en cada una de las vulnerabilidades web descritas, de forma que el trabajo tuviera una mejor estructura y coherencia que simplemente referenciar las guías sin más.

Se considera que ha habido una buena consecución de estos dos primeros objetivos secundarios, aunque no ha sido posible analizar, por falta de tiempo, las mejores prácticas de *frameworks* de *frontend* como React o Vue.

Otro objetivo secundario ha sido elegir las técnicas de análisis de vulnerabilidades de aplicaciones web. Se decidió desde el primer momento ir por vía de las pruebas de seguridad dinámica de aplicaciones web porque puede ser el primer contacto que tenga nuestra pyme con la ciberseguridad. Efectivamente, en mi experiencia laboral reciente, nuestras aplicaciones web fueron sometidas por alguno de nuestros clientes más grandes a un análisis de seguridad tipo DAST. Y esta es la realidad de muchas pymes que no han incorporado la ciberseguridad a su ciclo de desarrollo de software.

Derivado de este objetivo se eligieron las herramientas de análisis de seguridad web que se han utilizado en el trabajo. La herramienta OWASP ZAP creo que ha cumplido con el objetivo de ayudar al tipo de empresa que se viene mencionando, al ser gratuita y estar bien soportada por la comunidad. El software Burp Suite, edición Community se eligió para incorporar al conjunto de herramientas una que fuera comercial pero gratuita. Pero, en mi opinión, ha resultado demasiado limitada.

Al no disponer de análisis automatizado y requerir una exploración manual, aporta menos a una empresa si ésta todavía no tiene la experiencia en ciberseguridad para saber cómo buscar vulnerabilidades que explotar de forma manual. Además, no se ha podido incluir su uso en la metodología de análisis de seguridad, ya que sólo la versión comercial soporta la automatización. En este aspecto entonces, considero que se ha cumplido el objetivo de manera parcial y tal vez habría resultado más efectivo elegir otra herramienta *open source*, en lugar de una versión tan limitada de Burp Suite.

Por último, los dos últimos objetivos secundarios que se plantaban eran estudiar las principales vulnerabilidades de las aplicaciones web y buscar aplicaciones para analizar en ellas aquellas vulnerabilidades. Gracias a los recursos *open-source* de la fundación OWASP ya comentados, se ha podido conseguir un

cumplimiento adecuado de los objetivos. Pero, aunque se han explorado las diez vulnerabilidades más importantes en la tienda Juice Shop, cada una de ellas puede manifestarse en la aplicación de diversas formas y no ha sido posible analizarlas todas por las limitaciones de espacio y tiempo.

9. Trabajo futuro

Como ya se avanzaba en el punto "3.3. Limitaciones de las herramientas de análisis de vulnerabilidades", las herramientas de análisis de seguridad dinámico de aplicaciones (DAST) pueden ayudar a encontrar problemas genéricos (algunos importantes) en las aplicaciones. Pero, como describe en [18], no son infalibles, pues generan tanto falsos positivos (clasificando vulnerabilidades de manera incorrecta), como falsos negativos (no identificando vulnerabilidades que existen). La misma herramienta ZAP incluye en el informe de vulnerabilidades un grado de seguridad respecto al resultado, que no siempre es alto. Por tanto, aunque sirven de ayuda inicial, estas herramientas requieren de la experiencia del equipo de desarrollo, especialmente para confirmar si una vulnerabilidad reportada de verdad lo es.

Un aspecto que considerar para trabajos futuros es ampliar la metodología para que incluya herramientas de tipo análisis estático de seguridad de aplicaciones (SAST: *Static Application Security Testing*) y así complementar la metodología presentada con el análisis del código fuente. Este tipo de análisis puede ayudar a detectar problemas derivados de la aplicación concreta que el análisis dinámico no ha conseguido detectar. Esto implica introducir la ciberseguridad en las primeras fases del ciclo de vida del desarrollo del software lo que, para algunas empresas, puede suponer un cambio de mentalidad importante.

Por otra parte, en la propuesta inicial de este trabajo se consideraban también aplicaciones de escritorio junto con las aplicaciones web. Efectivamente, tecnologías como Electron[18] permiten utilizar las mismas herramientas de la pila JavaScript moderna que se han visto para desplegar aplicaciones de escritorio. De manera similar, React Native permitiría desarrollar aplicaciones móviles con React.

Analizar este tipo de aplicaciones no ha sido posible por falta de tiempo. Otra vía de investigación interesante sería analizar, de forma similar a como se ha hecho aquí, estas aplicaciones de escritorio o móviles y comprobar si las herramientas de análisis que se han presentado aquí también son efectivas para detectar vulnerabilidades en ellas.

[18] https://www.electronjs.org

10. Glosario

Adaptación de impedancias objeto-relacional: Conjunto de problemas conceptuales y técnicos derivados de trabajar con SGBD relacionales desde lenguajes de programación orientados a objetos. Fuente: Wikipedia.

Ataques de criptoanálisis por modificación de relleno *(padding oracle attack)*: ataque que utiliza la validación del relleno de un mensaje criptográfico para descifrar el contenido. Fuente: Wikipedia.

Cifradores de confidencialidad adelantada *(FS: forward secrecy)*: propiedad de los sistemas criptográficos que garantiza que el descubrimiento de las claves utilizadas actualmente no compromete la seguridad de las claves usadas con anterioridad. Fuente: Wikipedia.

***Clickjacking* (secuestro de clics)**: técnica que, mediante el uso de múltiples capas opacas o transparentes puede inducir a un usuario a hacer clic en un botón o enlace a otra página, cuando en realidad la intención del usuario no es navegar a esas páginas. Fuente: Clickjacking | OWASP Foundation.

CSRF (*Cross-site request forgery* o falsificación de petición en sitios cruzados): *Exploit* malicioso de un sitio web en que se transmiten comandos no autorizados a través de un usuario validado por el sitio web. Se conoce también como XSRF, ataque de un clic, secuestro de sesión y ataque automático. Fuente: Wikipedia.

CWE (Common Weakness Enumeration): lista desarrollada por la comunidad de debilidades comunes de software y hardware. La lista se actualiza varias veces al año y se puede consultar desde https://cwe.mitre.org/data/index.html. Fuente: CWE - About CWE.

DevOps: Acrónimo inglés de *development* (desarrollo) y *operations* (operaciones). Es un conjunto de prácticas que agrupan el desarrollo de software y las operaciones de IT. El objetivo es acelerar el ciclo de vida del desarrollo de software y proporcionar una entrega continua de calidad. Fuente: Wikipedia.

DevSecOps: Supone integrar las prácticas de seguridad en el enfoque anterior de DevOps.

DNS *rebinding*: Ataque que consiste en que un sitio web malicioso hace que un visitante ejecute un script en el cliente que ataca otras máquinas de la red. Fuente: Wikipedia.

DOM *(Document Object Model)*: El Modelo de Objetos del Documento es una API de programación para documentos HTML, que proporciona un conjunto estándar de objetos para representar documentos HTML, XHTML, XML y SVG, acceder a ellos y manipularlos. Fuente: Wikipedia.

ENISA: Agencia de la Unión Europea para la Ciberseguridad. Es la agencia de la Unión Europea a la que se le ha encomendado la misión de velar por un alto nivel común de ciberseguridad en toda Europa. Fuente: Acerca de la ENISA.

Finale-rest: paquete para Node.js que permite crear *endpoints* REST flexibles y controladores a partir de modelos Sequelize en una aplicación Express. Fuente: finale-rest - npm.

Fundación OWASP: Organización sin ánimo de lucro que trabaja por la mejora de la seguridad del software. Fuente: About the OWASP Foundation.

GitHub: Plataforma que permite alojar y compartir código fuente en línea. Se basa en el programa de control de versiones Git. https://github.com/.

JWT (JSON Web Token): Estándar basado en JSON para la creación de tokens de acceso que permiten la propagación de identidad y privilegios (*claims*). Fuente: Wikipedia.

HSTS (HTTP Strict Transport Security): Mecanismo por el cual un servidor web declara a los clientes (los navegadores) que solo puede interactuar con ellos mediante conexiones HTTP seguras (TLS/SSL). Fuente: Wikipedia.

ORM *(Object–relational mapping)*: Técnica de programación para convertir datos entre su representación en una BD relacional y el sistema de tipos de un lenguaje de programación orientado a objetos. Fuente: Wikipedia.

OWASP Dependency Check: Herramienta de análisis de composición de software (SCA: Software Composition Analysis) de la fundación OWASP para la detección de vulnerabilidades en las dependencias de un proyecto. Fuente: OWASP Dependency-Check | OWASP Foundation.

Payload (o carga útil): En seguridad informática, se refiere a la parte del malware que realiza la acción maliciosa tras una penetración exitosa. Fuente: Wikipedia.

Payment Card Industry Data Security Standard (PCI DSS): Estándar de seguridad de la información desarrollado por las compañías más importantes de medios de pago. Sirve como guía de ayuda a las organizaciones que procesan, almacenan y transmiten datos de los titulares de tarjetas, para asegurar dichos datos y evitar el fraude. Fuente: Wikipedia.

Phishing: Conjunto de técnicas que persiguen engañar a una víctima, haciéndose pasar por una persona, empresa o servicio de confianza, para manipularla y hacer que realice acciones como revelar información confidencial o hacer clic sobre un enlace. Fuente: Wikipedia.

Principios SOLID: Acrónimo mnemónico de los cinco principios básicos de la programación orientada a objetos y el diseño. Son: 1) *Single responsibility principle* (principio de responsabilidad única), 2) *Open/closed principle* (principio de abierto/cerrado), 3) *Liskov substitution principle* (principio de sustitución de Liskov), 4) *Interface segregation principle* (principio de segregación de la interfaz)

y 5) *Dependency inversion principle* (principio de inversión de la dependencia). Fuente: Wikipedia.

Proxy (o servidor *proxy*): En una red informática es un servidor, programa o dispositivo que hace de intermediario en las peticiones de recursos entre un cliente y un servidor. Fuente: Wikipedia.

Spider (rastreador o araña web): Es un programa informático que inspecciona las páginas web de forma metódica y automatizada. Comienzan visitando una URL para identificar los hiperenlaces y añadirlos a la lista de URL a visitar. Fuente: Wikipedia.

Sequelize: ORM moderno TypeScript y Node.js para MySQL y MariaDB, entre otros sistemas de BD. Fuente: Sequelize | Feature-rich ORM for modern TypeScript & JavaScript.

Stack Overflow: Plataforma de preguntas y respuestas donde se pueden formular cuestiones de programación y recibir respuestas de la comunidad. https://stackoverflow.com/.

Time-of-check to time-of-use (TOCTOU): Tipo de error en el desarrollo de software causado por una condición de carrera que implica comprobar el estado de una parte de un sistema (como una credencial de seguridad) y el uso de los resultados de tal comprobación. Fuente: Wikipedia.

11. Referencias

1. ENISA. ENISA Threat Landscape - The year in review [Internet]. 2020 oct [citado 5 de marzo de 2024]. Disponible en: https://www.enisa.europa.eu/publications/year-in-review/@@download/fullReport

2. Raible M. History of Web Frameworks Timeline [Internet]. 2019 [citado 11 de marzo de 2024]. Disponible en: https://github.com/mraible/history-of-web-frameworks-timeline/blob/master/history-of-web-frameworks-timeline-2019.jpg

3. Swacha J, Kulpa A. Evolution of Popularity and Multiaspectual Comparison of Widely Used Web Development Frameworks [Internet]. Basel: MDPI AG; 2023 p. 3563. Disponible en: https://www.mdpi.com/2079-9292/12/17/3563

4. G. Blinowski, A. Ojdowska, A. Przybyłek. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. 2022 p. 20357-74. Disponible en: https://ieeexplore.ieee.org/document/9717259

5. Lewis J, Fowler M. Microservices [Internet]. 2014 [citado 21 de marzo de 2024]. Disponible en: https://www.martinfowler.com/articles/microservices.html

6. Holmes S, Harber C. Getting MEAN with Mongo, Express, Angular, and Node, Second Edition [Internet]. Manning Publications; 2019. Disponible en: https://learning.oreilly.com/library/view/getting-mean-with/9781617294754/

7. Node.js — Security Best Practices [Internet]. [citado 24 de marzo de 2024]. Disponible en: https://nodejs.org/en/learn/getting-started/security-best-practices

8. Nodejs - Threat Model [Internet]. Google Docs. [citado 24 de marzo de 2024]. Disponible en: https://docs.google.com/document/d/10so8HJdNVYr9q66tyl6caK9s56c4ucNIUs_J1BzjZLo/edit?usp=embed_facebook

9. Security Best Practices for Express in Production [Internet]. [citado 24 de marzo de 2024]. Disponible en: https://expressjs.com/en/advanced/best-practice-security.html

10. Weber N. Evaluation and Comparison of Full-Stack JavaScript Technologies [Internet]. [citado 22 de marzo de 2024]. Disponible en: https://opus.hs-offenburg.de/frontdoor/deliver/index/docId/6125/file/22_08_31-Bachelor_Thesis-Weber-Nadine_Final.pdf

11. Subramanian V. Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node [Internet]. Apress; 2019. 552 p. Disponible en: https://learning.oreilly.com/library/view/pro-mern-stack/9781484243916/

12. Angular - Security [Internet]. [citado 24 de marzo de 2024]. Disponible en: https://angular.dev/best-practices/security

13. Vue.js - Security [Internet]. [citado 24 de marzo de 2024]. Disponible en: https://v2.vuejs.org/v2/guide/security

14. Designing Data-Intensive Applications [Internet]. [citado 8 de abril de 2024]. Disponible en: https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/

15. JSON And BSON [Internet]. MongoDB. [citado 8 de abril de 2024]. Disponible en: https://www.mongodb.com/json-and-bson

16. MongoDB Security Architecture [Internet]. MongoDB. [citado 8 de abril de 2024]. Disponible en: https://www.mongodb.com/resources/products/capabilities/mongodb-security-architecture

17. OWASP Foundation. Vulnerability Scanning Tools | OWASP Foundation [Internet]. [citado 28 de abril de 2024]. Disponible en: https://owasp.org/www-community/Vulnerability_Scanning_Tools

18. Amankwah R, Chen J, Kudjo PK, Towey D. An empirical comparison of commercial and open-source web vulnerability scanners [Internet]. 2020 [citado 28 de abril de 2024] p. 1842-57. Disponible en: https://www.researchgate.net/publication/342683707_An_empirical_comparison_of_commercial_and_open-source_web_vulnerability_scanners

19. OWASP Foundation. ZAP [Internet]. ZAP. [citado 27 de abril de 2024]. Disponible en: https://www.zaproxy.org/

20. PortSwigger. Download Burp Suite Community Edition - PortSwigger [Internet]. [citado 30 de abril de 2024]. Disponible en: https://portswigger.net/burp/communitydownload

21. PortSwigger. Burp Suite documentation [Internet]. [citado 30 de abril de 2024]. Disponible en: https://portswigger.net/burp/documentation

22. ZAP – Burp to ZAP Feature Map [Internet]. [citado 30 de abril de 2024]. Disponible en: https://www.zaproxy.org/docs/burp-to-zap-feature-map/

23. OWASP Foundation. WSTG - v4.2 | OWASP Foundation [Internet]. [citado 26 de abril de 2024]. Disponible en: https://owasp.org/www-project-web-security-testing-guide/v42/

24. OWASP Vulnerable Web Applications Directory | OWASP Foundation [Internet]. [citado 4 de mayo de 2024]. Disponible en: https://owasp.org/www-project-vulnerable-web-applications-directory/

25. OWASP Juice Shop | OWASP Foundation [Internet]. [citado 4 de mayo de 2024]. Disponible en: https://owasp.org/www-project-juice-shop/

26. Bjoern Kimminich. Pwning OWASP Juice Shop [Internet]. [citado 4 de mayo de 2024]. Disponible en: https://pwning.owasp-juice.shop/companion-guide/latest/index.html

27. The ZAP Dev Team. ZAP – Getting Started [Internet]. [citado 5 de mayo de 2024]. Disponible en: https://www.zaproxy.org/getting-started/

28. Intercepting HTTP traffic with Burp Proxy [Internet]. [citado 6 de mayo de 2024]. Disponible en: https://portswigger.net/burp/documentation/desktop/getting-started/intercepting-http-traffic

29. OWASP Foundation. OWASP Top Ten | OWASP Foundation [Internet]. [citado 28 de mayo de 2024]. Disponible en: https://owasp.org/www-project-top-ten

30. OWASP Top 10:2021 [Internet]. [citado 3 de junio de 2024]. Disponible en: https://owasp.org/Top10/

31. Bjoern Kimminich. Pwning OWASP Juice Shop [Internet]. [citado 3 de junio de 2024]. Disponible en: https://pwning.owasp-juice.shop/companion-guide/latest/index.html

32. SQL Injection Attacks Using OWASP Zap Fuzzer [Internet]. 2022 [citado 29 de mayo de 2024]. Disponible en: https://www.youtube.com/watch?v=S2GggY6tcZs

33. ZAP – Fuzzing [Internet]. [citado 29 de mayo de 2024]. Disponible en: https://www.zaproxy.org/docs/desktop/addons/fuzzer/

34. GitHub - danielmiessler/SecLists. [Internet]. [citado 1 de junio de 2024]. Disponible en: https://github.com/danielmiessler/SecLists

35. Diving Deeper into AI Package Hallucinations [Internet]. 2024 [citado 2 de junio de 2024]. Disponible en: https://www.lasso.security/blog/ai-package-hallucinations

36. Lanyado B. Can you trust ChatGPT's package recommendations? [Internet]. Vulcan Cyber. 2023 [citado 2 de junio de 2024]. Disponible en: https://vulcan.io/blog/ai-hallucinations-package-risk/

37. Ntantogian C, Bountakas P, Antonaropoulos D, Patsakis C, Xenakis C. NodeXP: NOde.js server-side JavaScript injection vulnerability DEtection and eXPloitation [Internet]. 2021 may [citado 8 de junio de 2024] p. 102752. Disponible en: https://www.sciencedirect.com/science/article/pii/S221421262100003X

38. WSTG - v4.2 | OWASP Foundation [Internet]. [citado 10 de junio de 2024]. Disponible en: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05.6-Testing_for_NoSQL_Injection

39. FAQ: MongoDB Fundamentals - MongoDB Manual v7.0 [Internet]. [citado 10 de junio de 2024]. Disponible en: https://www.mongodb.com/docs/manual/faq/fundamentals/

40. OWASP Web Security Testing Guide | OWASP Foundation [Internet]. [citado 3 de junio de 2024]. Disponible en: https://owasp.org/www-project-web-security-testing-guide/

41. Introduction - OWASP Cheat Sheet Series [Internet]. [citado 3 de junio de 2024]. Disponible en: https://cheatsheetseries.owasp.org/

42. Jenkins [Internet]. Jenkins. [citado 4 de junio de 2024]. Disponible en: https://www.jenkins.io/

43. How to Automate OWASP ZAP [Internet]. Jit. 2023 [citado 4 de junio de 2024]. Disponible en: https://www.jit.io/resources/owasp-zap/how-to-automate-owasp-zap

44. ZAP – Session Context Authentication screen [Internet]. [citado 4 de junio de 2024]. Disponible en: https://www.zaproxy.org/docs/desktop/ui/dialogs/session/context-auth/

45. ZAP – Automation Framework [Internet]. [citado 4 de junio de 2024]. Disponible en: https://www.zaproxy.org/docs/desktop/addons/automation-framework/