UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PLATAFORMA PARA EL SEGUIMIENTO DE VULNERABILIDADES EN APLICACIONES WEB

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

BASTIÁN ANDRÉS PEZOA VERGARA

PROFESOR GUÍA:
EDUARDO GODOY VEGA

MIEMBROS DE LA COMISIÓN:
ALEJANDRO HEVIA ANGULO
PATRICIO INOSTROZA FAJARDIN

SANTIAGO DE CHILE
2024

Resumen

El trabajo consiste en el desarrollo de una plataforma web dedicada al seguimiento y gestión de vulnerabilidades en aplicaciones web. Mediante el uso de tecnologías como Django, Nginx y Docker, la plataforma permite a los usuarios cargar informes de escaneo de vulnerabilidades generados por herramientas de escaneo para aplicaciones web, a través de una API, y realizar un seguimiento de las vulnerabilidades detectadas a lo largo del tiempo. Además, se implementa un panel de control que permite visualizar el nivel de seguridad mediante diferentes gráficos y métricas, calculando un puntaje en función del nivel de apetito de riesgo designado por el usuario y las características de las vulnerabilidades encontradas, asignando una nota en consecuencia.

La metodología de trabajo utilizada se basa en un enfoque iterativo e incremental. Inicialmente, se definieron los requisitos del sistema y se seleccionaron las tecnologías adecuadas para su implementación. Luego, se procedió con el diseño y desarrollo de los componentes principales de la plataforma, entre los que se incluyen la API para la integración de informes de escaneo de vulnerabilidades y el sistema de puntuación de seguridad. Finalmente, se desplegó la aplicación, se realizaron pruebas de usuario para ajustar el diseño y la funcionalidad de la plataforma, ayudando a validar la solución propuesta.

El objetivo principal es proporcionar a los usuarios una herramienta centralizada, gratuita y de código abierto, para monitorear y mejorar la seguridad de sus aplicaciones web de manera efectiva y sencilla. Se espera que la plataforma ayude a los equipos de desarrollo y seguridad a gestionar el ciclo de vida de las vulnerabilidades presentes en sus aplicaciones de manera oportuna, contribuyendo al aspecto de seguridad de sus desarrollos y mejorando la integridad y confianza en las aplicaciones web.

A mis padres, por apoyarme y animarme durante toda mi formación como profesional. A Scarlette, tu amor y compañía incondicional han sido mi faro en los momentos más difíciles. A los que conocí durante estos años y que ahora son parte importante de mí, los quiero.

Tabla de Contenido

Índice de Ilustraciones

Listings

Capítulo 1

Introducción

En este capítulo se presenta una introducción al trabajo realizado en esta memoria de título. Se explora el contexto actual de las tecnologías en el ámbito global y los problemas que surgen de ellas, enfocándose en la necesidad de gestionar las vulnerabilidades en aplicaciones web. Se detallan las motivaciones para abordar estos problemas, los objetivos planteados para el trabajo, y se esboza la solución propuesta sin profundizar en los detalles. Además, se describe la metodología empleada y se ofrece una visión general de la estructura del documento que compone esta memoria.

1.1. Contexto actual

Actualmente, vivimos en la denominada "era digital" la cual ha revolucionado diversos aspectos de la vida. Internet y la tecnología se encuentran presentes en diversas actividades de nuestro día a día, transformando la manera en que nos relacionamos, comunicamos, trabajamos y aprendemos. Si bien esto ha logrado agilizar procesos, permitir el acceso instantáneo a la información, descubrir nuevas industrias y oportunidades económicas, entre otros beneficios, también nos plantea desafíos y preocupaciones. La privacidad en línea, la brecha digital, la desinformación, la dependencia tecnológica y, en particular, la seguridad informática, son algunas de estas.

Seguridad informática, o ciberseguridad, se entiende de muchas formas diferentes, de hecho, no posee una definición específica en el idioma español, bajo la "Real Academia Española" (o RAE, por sus siglas), y en el idioma inglés, según el Oxford English Dictionary, se define, desde el año 1990, como la "seguridad relacionada con sistemas informáticos o Internet... que pretende proteger contra virus o fraude" [22]. Por otro lado, los especialistas en el área, fuentes especializadas en tecnología y empresas líderes en el rubro, también dan sus propias definiciones. Por ejemplo, Amazon Web Services (o AWS por sus siglas en inglés) la define como la "práctica de proteger equipos, redes, aplicaciones de software, sistemas críticos y datos de posibles amenazas digitales. ..." [2], mientras que Microsoft entrega su definición de una manera más simple como "la práctica de proteger su información digital, dispositivos y activos." [16].

Esta diversidad de perspectivas al momento de intentar definir, o conceptualizar, qué es la seguridad informática refleja como se ha convertido en un elemento crucial en el desarrollo de aplicaciones y software. Si bien el aspecto crítico, o principal, de esta es resguardar la integridad de la información, asegurando que esta no sea objeto de manipulaciones maliciosas, también busca garantizar la disponibilidad continua de los servicios, prevenir pérdidas financieras y proporcionar una defensa sólida y constante frente a ataques y amenazas cibernéticas emergentes. Asimismo, desempeña un papel fundamental en la construcción y preservación de la confianza de los usuarios, manteniendo una reputación en términos de seguridad. Es decir, la seguridad informática es un pilar esencial para salvaguardar los activos digitales y mantener la integridad de las operaciones tecnológicas.

Sin embargo, en este contexto arduo y desafiante de la seguridad informática, existe una compensación entre el nivel de seguridad de una aplicación y la eficacia de las operaciones o el negocio de la organización. Este *"trade-off"* se conoce como el apetito de riesgo y se define como la "cantidad y tipo de riesgo que una organización está dispuesta a perseguir o retener" [12]. Se debe considerar el deseo de los usuarios de poder definir un nivel de apetito de riesgo, y también un nivel de tolerancia a este, si lo que se busca es mejorar el aspecto de seguridad de la aplicación, u organización, considerando la eficiencia de esta y/o su negocio.

1.2. Motivación

Con el crecimiento que ha tenido la tecnología y la información en esta era digital, también se ha observado un incremento en las amenazas cibernéticas [14, 29]. El riesgo para las empresas, organizaciones y usuarios, partiendo por la explotación de vulnerabilidades en el código, hasta la filtración de datos sensibles, es constante. Lo antes expuesto plantea la necesidad de desarrollar estrategias efectivas para analizar y mejorar la seguridad en las aplicaciones y software creados. De aquí nace el propósito de esta memoria de título, la cual se enfoca en abordar una problemática muy común en el área de la ciberseguridad: la evaluación y seguimiento eficiente del aspecto de seguridad en aplicaciones, en particular en las basadas en la web.

El estudio de algoritmos destinados a la identificación de vulnerabilidades y la evaluación de su grado de criticidad ha entregado como resultado una gran variedad de herramientas especializadas que automatizan este proceso. Esto se manifiesta en diversas áreas de desarrollo, como la infraestructura como código, el desarrollo de aplicaciones web, desarrollo de aplicaciones móviles, redes, entre otras. Cada una de estas áreas tiene sus propias particularidades y desafíos, a los que las herramientas se adaptan. Esto permite que, debido a la constante evolución de las amenazas cibernéticas, las herramientas se adapten a las nuevas demandas en función de un área de desarrollo específica.

Pero algo muy interesante que queda por trabajar es el saber cuántas de las vulnerabilidades encontradas por estas herramientas han sido mitigadas, cuántas persisten, cuántas nuevas han surgido, cómo las aún presentes logran definir un grado de seguridad sobre la aplicación escaneada y si este grado de seguridad es aceptable, se puede admitir o debe ser rechazado en su totalidad. Toda esta información que se puede obtener de los chequeos es esencial para conseguir evaluar la efectividad de las medidas de seguridad implementadas,

para tomar decisiones informadas sobre la protección de la aplicación o software y para no afectar la eficacia de las operaciones.

Este escenario conlleva la integración de dos terrenos fundamentales: la Seguridad Informática, con un enfoque en la gestión de vulnerabilidades inherentes a la aplicación, y la Gestión de Riesgos, encargada de establecer un marco de apetito de riesgo para esta. En la primera, se busca desempeñar un papel activo al rastrear y abordar las vulnerabilidades detectadas, y, simultáneamente, la segunda, se embarca en la tarea de definir un nivel de apetito de riesgo para la aplicación, acompañado de su respectivo nivel de tolerancia.

Esto no solo proporciona un marco claro para determinar si la aplicación posee un nivel de seguridad aceptable, sino que también permite alinear de manera efectiva los objetivos del negocio y/o la eficacia de las operaciones con las estrategias de seguridad propuestas. Así, la intersección entre ambos terrenos se convierte en un componente fundamental para la toma de decisiones informadas sobre la seguridad de la aplicación, garantizando la protección de esta contra amenazas.

1.3. Objetivos

Objetivo General

El objetivo principal de este trabajo es desarrollar un sistema para el seguimiento y análisis de vulnerabilidades en aplicaciones web. Este sistema se implementará mediante un tablero de control que tendrá en cuenta el apetito de riesgo de la aplicación para calcular un nivel de seguridad sobre esta. Su función principal será recibir informes generados por herramientas de escaneo de vulnerabilidades en sitios web y proporcionar una representación visual y cuantitativa de la evolución de la seguridad de los sitios web escaneados a lo largo del tiempo.

Objetivos específicos

1. Identificar y seleccionar dos herramientas de análisis de vulnerabilidades de código para aplicaciones web ampliamente conocidas, para ser integradas en el sistema de seguimiento.
2. Diseñar detalladamente el tablero de mando, especificando las métricas y visualizaciones a utilizar para presentar la evolución de las vulnerabilidades de la aplicación.
3. Diseñar y desarrollar el algoritmo encargado de analizar los reportes generados por las herramientas y realizar la integración de la información recopilada por estas con el sistema de seguimiento.
4. Diseñar y crear un sistema de almacenamiento de datos eficiente que registre los resultados de los análisis para cada aplicación, permitiendo la posterior comparación y procesamiento de estos.

5. Crear las interfaces de programación de aplicaciones, o API por sus siglas en inglés, para las herramientas seleccionadas, que permitan cargar los archivos generados en los reportes por estas.
6. Diseñar y desarrollar las vistas de la plataforma web, contemplando la implementación de la lógica necesaria, el estilo visual consistente y la integración de scripts interactivos para asegurar una experiencia de usuario fluida y efectiva.
7. Integrar los diferentes componentes creados del sistema
8. Realizar pruebas exhaustivas del sistema, apoyándose del uso de aplicaciones inseguras de testeo, para asegurar el correcto funcionamiento de este.
9. Generar la documentación detallada del sistema, describiendo arquitectura utilizada, funcionalidades, uso del sistema, facilitando la adopción y mantención de este.
10. Validar la implementación de la solución propuesta mediante el uso de la aplicación en un ambiente de pruebas real.

1.4. Solución propuesta

La solución propuesta consiste en una plataforma web desarrollada con el *framework* Django, que permitirá a los usuarios hacer seguimiento de las vulnerabilidades presentes en sus aplicaciones y visualizar el estado de seguridad de estas en un tablero de mando mediante gráficos y métricas. Los usuarios podrán subir los reportes de seguridad generados por herramientas de escaneo de vulnerabilidades mediante una API implementada con Django REST Framework, la cual procesará los archivos y almacenará la información en una base de datos gestionada con MySQL.

La plataforma ofrecerá vistas basadas en las operaciones *CRUD* (Crear, Leer, Actualizar y Borrar), permitiendo a los usuarios gestionar todos los datos asociados a su cuenta. Es decir, el usuario podrá crear aplicaciones para su monitoreo en la plataforma, visualizar las vulnerabilidades asociadas a estas aplicaciones, actualizar el estado de las vulnerabilidades (por ejemplo, marcarlas como mitigadas) y eliminar escaneos importados junto con las vulnerabilidades reportadas, entre otros.

Las interfaces de la página se crearán utilizando HTML, CSS y JavaScript nativos, junto con librerías de código abierto para agilizar el desarrollo y mejorar las funcionalidades. Entre estas librerías se incluyen BootStrap, para la creación de componentes y estilo de la página; Chart.js, para la creación de los gráficos interactivos del tablero de mando; y DataTables, para generar tablas interactivas en las cuales el usuario pueda visualizar y manipular la información asociada a su aplicación en seguimiento.

En la figura 1.1 se observa el diagrama de flujo para la plataforma. En este se puede apreciar cómo un usuario puede crear aplicaciones para hacer seguimiento, importar reportes de herramientas de escaneo de vulnerabilidades y obtener los análisis de seguridad de estas mediante un tablero de mando, o *dashboard*.

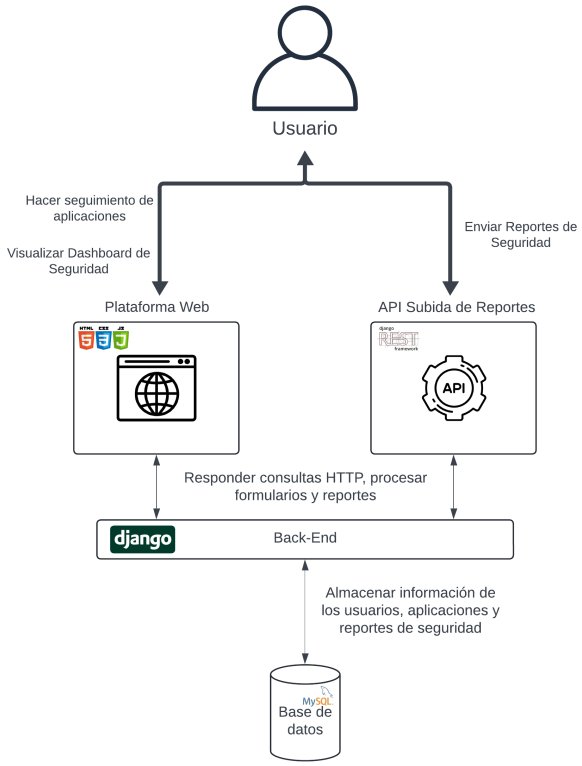

Figura 1.1: Diagrama de flujo de la plataforma propuesta

1.5. Metodología

El trabajo se lleva a cabo mediante el uso de una metodología de tipo Scrum. Esta metodología ágil permite organizar el desarrollo del proyecto en iteraciones o "*sprints*" de corta duración, lo que facilita la adaptación a los cambios y la entrega constante de valor. Se establecen objetivos y tareas cada semana en función de cumplir tres sprints de cinco semanas cada uno.

Para dar inicio al desarrollo de la plataforma, el primer *sprint* abarcó el diseño de esta. En esta primera etapa se seleccionaron las herramientas de escaneo de vulnerabilidades, se diseñaron los *mockups* para las interfaces de la plataforma, se creó el algoritmo para *parsear* los reportes generados por las herramientas de escaneo, se diseñó el modelo matemático encargado de asignar el puntaje y nota de seguridad para la aplicación y se establecieron los esquemas de relación para las entidades de la base de datos.

El segundo *sprint* se concentró en el desarrollo del código fuente de la plataforma. En esta segunda etapa, se confeccionaron los diferentes componentes de la plataforma, se creó la lógica y las vistas de la plataforma. Se desarrolló la API y se integraron las métricas y visualizaciones en el tablero de control. Se realizaron pruebas unitarias para garantizar que los desarrollos funcionaran correctamente y se solucionaron los errores que surgieron durante la implementación. Esto estableció una base sólida sobre la cual poder construir y refinar las características de la plataforma.

Por último, el tercer *sprint* se enfocó en la validación del trabajo desarrollado. En esta etapa, se llevó a cabo una prueba de usuarios en la que desarrolladores y profesionales del área de ciberseguridad interactuaron con la plataforma para evaluar sus funcionalidades y aspecto visual. Con ello, se recopilaron comentarios para realizar ajustes y mejoras en la plataforma. Esta prueba de usuarios demostró la utilidad de la plataforma, así como el valor y relevancia del proyecto, lo que la convierte en una solución con mucho potencial.

1.6. Estructura de la memoria

El objetivo del presente informe es documentar y describir detalladamente el proyecto titulado "Plataforma para el seguimiento de vulnerabilidades en aplicaciones web" que ha sido desarrollado como tema de memoria. A continuación, se detalla la estructura que comprende este documento.

- En el capítulo 2 se analizan algunas de las herramientas del seguimiento de vulnerabilidades en aplicaciones web actuales y se justifica la necesidad de una nueva solución.
- El capítulo 3 describe el diseño de la solución, abarcando el front-end y back-end de la aplicación, la gestión de los datos, las tecnologías empleadas y cómo se implementará.
- En el capítulo 4 se detalla cómo se desarrolló la solución diseñada, incluidos los aspectos técnicos del código fuente, las configuraciones de las tecnologías utilizadas y las medidas de seguridad implementadas.
- En el capítulo 5 se evalúa el desarrollo llevado a cabo, validando su exactitud, funcionalidad y utilidad.
- El capítulo 6 presenta las conclusiones del trabajo realizado y propone posibles mejoras y desarrollos futuros.

Capítulo 2

Marco teórico

Este capítulo examina las herramientas actuales utilizadas para el seguimiento de vulnerabilidades en aplicaciones web. Se analizan sus características, ventajas y limitaciones, lo que lleva a identificar la necesidad de una nueva solución.

2.1. Herramientas actuales

En la actualidad, existen numerosas herramientas para llevar a cabo análisis de vulnerabilidades sobre aplicaciones y software. Muchas de estas son ampliamente difundidas y conocidas, sin embargo, una limitación a la cual se enfrentan los profesionales en el área es la ausencia de una solución que permita evaluar el progreso de la seguridad frente a sucesivas revisiones y reportes de vulnerabilidades de manera efectiva, y que también considere un apetito de riesgo sobre la protección de la aplicación. Esta carencia no solo dificulta el trabajo de los profesionales del área de ciberseguridad, sino que también pone en riesgo lo que la seguridad informática busca garantizar.

Se pueden encontrar diversas herramientas que buscan solucionar esta problemática, en el presente trabajo analizaremos 4 de las más utilizadas:

2.1.1. DefectDojo

DefectDojo [7] es, según la OWASP Foundation [21], una herramienta de gestión de vulnerabilidades de código abierto. Esta ofrece plantillas, generación de informes, métricas y herramientas básicas de autoservicio, permite la integración con diversas herramientas de escaneo de vulnerabilidades y en diferentes extensiones de archivo para los reportes. Ofrece características adicionales suscritas bajo planes de pago mediante un modelo de *Software as a Service* (o SaaS, por sus siglas).

Esta herramienta está desarrollada utilizando Python como lenguaje de programación. También hace uso del *framework* Django, lo que integra plantillas de lenguaje de marcado

de hipertexto (HTML) junto al uso de JavaScript como lenguaje para programar scripts en estas. Proporciona archivos de extensión YAML, llamados manifiestos de Kubernetes, en los cuales se declaran los recursos necesarios para desplegar la aplicación dentro de un clúster utilizando el sistema orquestador de contenedores Kubernetes (K8s), y archivos Dockerfile para realizar la instalación utilizando contenedores Docker mediante el uso de la herramienta Docker Compose.

Utiliza lo que sus creadores denominan como un modelo de productos y compromisos [8]. Los productos representan las aplicaciones o sistemas que se encuentran bajo evaluación dentro de DefectDojo y los compromisos, o *"engagements"*, son los momentos en el tiempo en que se realizan pruebas sobre la aplicación. Cada engagement posee las pruebas, o *"test"*, las cuales corresponden a aquellas actividades realizadas para descubrir vulnerabilidades y defectos en el producto, por ejemplo, un escaneo de la herramienta Burp Suite. Por último, están los hallazgos, o *"findings"*, que corresponden a las vulnerabilidades específicas que han sido reportadas, encapsuladas en cada uno de los tests realizados. [6]

Con ello, DefectDojo, logra establecer un modelo enormemente jerárquico, que favorece la categorización de los resultados y el orden de los reportes importados, e incorpora un enfoque temporal mediante la introducción de sus *"engagement"*, permitiendo evaluar la seguridad en un análisis detallado e histórico a lo largo del tiempo. Esto nace bajo el foco de buscar reducir la cantidad de tiempo que los profesionales de seguridad dedican a registrar vulnerabilidades.

A pesar de los beneficios de este enfoque jerárquico implementado por Defectdojo, los altos niveles de jerarquización no siempre son la opción más adecuada. Esta estructura conlleva limitaciones en términos de flexibilidad y adaptabilidad de la herramienta. La rigidez heredada del modelo puede convertirse en una restricción en situaciones donde se necesite reconfigurar el producto, incorporar nuevos elementos, entre otros, lo que podría llegar a ser crucial en entornos de cambio constante como lo es la seguridad informática. Además, genera una sobrecarga cognitiva para los usuarios el navegar a través de múltiples niveles de categorías.

A esto se suma el hecho de que las métricas y estadísticas acerca del nivel de seguridad son características secundarias en el diseño de DefectDojo, pues este centra la atención en la documentación de las vulnerabilidades encontradas. Por ello, las métricas e información que logra levantar respecto a los avances de seguridad del sitio web escaneado son insuficientes, o poco valiosas.

2.1.2. Tenable Nessus

Nessus [23] es otra herramienta para la evaluación de vulnerabilidades sobre las aplicaciones desarrollada por la compañía especializada en ciberseguridad, Tenable. Esta herramienta es ampliamente utilizada debido a su completitud, pues cuenta con auditorías de configuración, cumplimiento y seguridad, con chequeos para aplicaciones web y redes, posibilidad de realizar ataques externos y configurar reportes de vulnerabilidad. Además, presenta un apartado de resultados en vivo en donde se pueden visualizar métricas y gráficos simples sobre los análisis realizados.

Nessus se lanzó como herramienta de código abierto y uso gratuito en 1998, bajo el nombre de Proyecto Nessus, siendo pioneros en el área. Se utilizó C como lenguaje de programación para su desarrollo en ese entonces. El año 2005, con el lanzamiento de Nessus 3, se convirtió en un producto comercial con su versión empresarial y su desarrollo pasó a ser privado, perdiéndose el rastro sobre las tecnologías utilizadas en su avance. No obstante, el código fuente de Nessus 2 que quedó disponible fue utilizado por otros desarrolladores, convirtiéndolo en una base para nuevas herramientas de seguridad tales como OpenVAS [11].

Si bien no se tienen muchos detalles acerca de la arquitectura y tecnologías utilizadas para desarrollar Nessus hoy en día, podemos ver que su modelo parece ser excesivo para abordar la problemática específica que se intenta solucionar. Esto dado que Tenable propone Nessus como el estándar global en evaluación de vulnerabilidades construido para la superficie de ataque moderna [23], lo cual la convierte en una herramienta muy completa. La naturaleza exhaustiva de esta herramienta es valiosa en su conjunto, pero genera una complejidad innecesaria y redundante para la tarea particular de evaluar la evolución de la seguridad de manera histórica en aplicaciones web.

Nessus se destaca por una serie de características que lo convierten en una herramienta multifuncional para la detección de vulnerabilidades en diversos componentes. La capacidad de poder ser desplegado en cualquier plataforma gracias a su portabilidad, junto a la existencia de diversos *plugins* que se compilan dinámicamente para optimizar el rendimiento según la acción realizada, son atributos considerables y de alto nivel de complejidad. La presencia de plantillas y políticas pre configuradas, agiliza la auditoría de configuraciones, permitiendo una comprensión más profunda de las áreas vulnerables en los sistemas. Estas propiedades hacen de Nessus una opción atractiva, especialmente para grandes organizaciones con recursos financieros más amplios y requisitos de seguridad más complejos.

2.1.3. Tenable Vulnerability Management & Security Center

También existen diferentes herramientas de usos más específicos desarrolladas por Tenable. En particular, Tenable Vulnerability Management [25] y Security Center [24]. Ambas son plataformas de gestión de vulnerabilidades que permiten identificar, evaluar y gestionar las vulnerabilidades de los sistemas y aplicaciones. La gran diferencia entre ambas es la forma que estas son gestionadas, Vulnerability Management se gestiona en la nube de Tenable, mientras que Security Center es una solución *On-Premise*, es decir, se instala en los servidores propios del cliente. [25]

Estas herramientas proporcionan una interfaz centralizada para la administración de vulnerabilidades con tableros de control pre configurados y personalizables, ofrecen lo que Tenable denomina como su "priorización predictiva", un sistema inteligente que utiliza ciencia de datos basada en el riesgo y explotación de las vulnerabilidades para proporcionar un puntaje de riesgo a estas, y la opción de generar reportes sobre el estado de seguridad actual. Cuentan con herramientas y sensores programables que escanean los activos constantemente y en tiempo real, realizan monitoreos pasivos y permiten la integración de otras herramientas de seguridad.

Tanto Security Center como Vulnerability Management se presentan como soluciones in-

tegrales para abordar la problemática planteada en este trabajo. Sin embargo, una desventaja significativa radica en el enfoque con el que estas fueron concebidas. Ambos softwares están diseñados principalmente para satisfacer las necesidades de entornos grandes con recursos considerables para invertir en seguridad. El modelo de suscripción que Tenable propone para estos productos resulta prohibitivo para usuarios individuales, empresas con recursos limitados o aquellos que simplemente buscan realizar un seguimiento de seguridad sobre sus aplicaciones sin incurrir en gastos sustanciales. Esto limita la accesibilidad y utilidad de estas herramientas para una amplia gama de usuarios, lo que podría beneficiar enormemente la seguridad informática de las aplicaciones desarrolladas hoy en día.

2.1.4. Brinqa

Brinqa [3] es una plataforma inteligente para la gestión de riesgos cibernéticos en toda la superficie de ataque de una organización. Se denomina como un centro de operaciones de riesgo en el cual se identifican, gestionan y mitigan vulnerabilidades y riesgos en la infraestructura digital, facilitando la priorización y remediación eficiente. Se presenta como una solución para cuatro problemas comunes al desarrollar esta tarea: El manejo de enormes volúmenes de vulnerabilidades provenientes de múltiples fuentes. La falta de centralización de la información acerca de los riesgos provenientes de diferentes herramientas con distintos conceptos de riesgo. La complejidad de la propiedad cuando los activos tienen múltiples propietarios o carecen de uno claro. Las carencias que existen al momento de comunicar y presentar los riesgos cibernéticos hallados a ejecutivos y juntas directivas.

Para ello, implementan lo que llaman "Centro de Operaciones de Riesgo (ROC)", que gestiona proactivamente las amenazas en infraestructura, nube y aplicaciones. Esto se convierte en una plataforma integral para gestionar los riesgos de toda la superficie de ataque, unificando diferentes fuentes de datos y herramientas mediante conectores de datos, creando un inventario consolidado de activos y vulnerabilidades. Además, permite orquestar remediaciones mediante la automatización de estas, priorizar vulnerabilidades basándose en el riesgo y generar análisis de impacto en el negocio.

Según la "The Forrester Wave: Vulnerability Risk Management, Q3 2023" los usuarios catalogaron Brinqa como una plataforma personalizable y extensible pero de implementación compleja y muy extensa.[9] Lo establecen como una opción para aquellas organizaciones que requieran de un alto nivel de personalización para los factores de su entorno, especialmente en el desarrollo de aplicaciones. A pesar de ello, este estudio lo logra catalogar como un fuerte contendiente frente a las herramientas de esta área.

Sin embargo, Brinqa, al igual que las herramientas de Tenable, debido a su integridad y versatilidad, está orientada para clientes con presupuestos significativos destinados a la seguridad informática. Al ser una solución con una amplia gama de herramientas, con características para abordar diversas áreas de la ciberseguridad y capaz de adaptarse a múltiples casos gracias a su personalización, hacen que sea una opción atractiva para grandes organizaciones con mayores recursos financieros y necesidades complejas. Esto termina limitando su accesibilidad para usuarios individuales y/u organizaciones con recursos financieros ajustados.

2.2. La necesidad de una nueva solución

Este estudio del arte nos muestra una problemática presente en el área de la seguridad informática: la existencia de diversas herramientas para la gestión de vulnerabilidades con altos niveles de personalización y completitud, bajo planes de suscripción de grandes cantidades de dinero, versus la casi nula existencia de herramientas de uso gratuito, o de bajo costo, que permitan hacer el seguimiento del aspecto de seguridad de aplicaciones de buena manera.

Al analizar las soluciones actuales, observamos que estas herramientas comerciales ofrecen una serie de características avanzadas, tales como análisis en tiempo real, informes detallados y alertas automatizadas. Sin embargo, estas funcionalidades vienen acompañadas de costos prohibitivos para muchas organizaciones pequeñas y medianas, así como para desarrolladores independientes. Esta brecha en el mercado crea una barrera significativa para la adopción de buenas prácticas de seguridad en sectores con recursos limitados.

Además, las soluciones gratuitas disponibles carecen de funcionalidades necesarias para realizar un seguimiento continuo de las vulnerabilidades, pues su enfoque principal es el registro de estas. Las interfaces suelen ser poco intuitivas, complicando el uso para usuarios o equipos que no son expertos en el área. Como resultado, las organizaciones que utilizan estas herramientas, o dependen de ellas, pueden estar en desventaja al intentar mantener su entorno seguro.

Esto permite visualizar la necesidad de una nueva solución con un enfoque moderno y específico, y de uso gratuito, que permita realizar el seguimiento de vulnerabilidades en aplicaciones web. Un sistema novedoso debe generar métricas y visualizaciones no triviales sobre la información entregada por los reportes, a la vez que se visualiza el progreso de la postura de seguridad del sitio web y se considera el apetito de riesgo de esta.

Adicionalmente, esta herramienta debe ser intuitiva y accesible para usuarios con diferentes niveles de experiencia en el área informática. Esto implica el diseño de una interfaz de usuario clara y proporcionar documentación detallada que ayude a los usuarios a comprender y utilizar la herramienta de manera efectiva. De este modo, se generan herramientas de alto estándar para uso gratuito y se promueve una cultura de seguridad más inclusiva.

Capítulo 3

Diseño de la solución

En este capítulo se detalla la arquitectura de la plataforma, describiendo el proceso de selección de las herramientas de escaneo y la confección del modelo para calcular la nota de seguridad. Se presentan las métricas y gráficos seleccionados para ser mostrados en el panel de control, se especifican las tecnologías utilizadas tanto en el *back-end* como en el *front-end*. Se describen las entidades y relaciones de la base de datos. Por último, se aborda el método de despliegue de la solución.

3.1. Arquitectura de la solución

La arquitectura de la solución desarrollada consiste en una plataforma web construida con el *framework* Django, que gestiona tanto los datos como la lógica del *back-end*. Esta utiliza NGINX para levantar un servidor *proxy* encargado de servir el contenido estático, y MySQL para la gestión de la base de datos. Todo esto se implementa utilizando Docker, donde cada componente de la aplicación se despliega como un servicio independiente en un contenedor, lo que facilita la administración y el escalado de la solución. Su función es gestionar y hacer seguimiento de las vulnerabilidades en aplicaciones web, para ello se estructura en torno a una API robusta que permite la carga de informes generados por herramientas de escaneo de seguridad para aplicaciones web.

La aplicación presenta un panel de control, o *dashboard*, interactivo que ofrece métricas detalladas sobre el estado de seguridad de las aplicaciones monitoreadas. Entre las métricas desarrolladas, se incluye una función matemática que asigna una nota de seguridad basada en una suma ponderada de la cantidad de vulnerabilidades, clasificadas según su nivel de severidad. Además, se permite visualizar la cantidad de vulnerabilidades por severidad a lo largo del tiempo, la comparación entre vulnerabilidades activas y mitigadas, gráficos de tendencias y frecuencia de vulnerabilidades. Este panel de control proporciona a los usuarios una visión clara y actualizada de la postura de seguridad, permitiéndoles seguir el estado de las vulnerabilidades y actualizarlas como mitigadas una vez resueltas.

Los datos almacenados y utilizados por la aplicación en el *back-end* se almacenan en

una base de datos relacional, administrada mediante el sistema de código abierto MySQL. Esta base de datos está estructurada en cuatro tablas principales: *User*, *Application*, *Scan* y *Vulnerability*. Estas tablas se encargan de almacenar, respectivamente, los datos del usuario, de la aplicación, de los escaneos de seguridad y de las vulnerabilidades.

El *front-end* de la aplicación está configurado utilizando NGINX como servidor *proxy*, este redirige eficientemente las comunicaciones hacia el *back-end* gestionado por Django. NGINX también se encarga de servir los archivos estáticos, lo cual permite separar esta tarea de las funciones del *back-end*, optimizando así el rendimiento general de la aplicación. Esta arquitectura asegura una gestión eficaz del tráfico y mejora la capacidad de respuesta, proporcionando una experiencia más fluida y confiable al usuario.

Se puede encontrar una representación visual y más detallada de la arquitectura de la solución en la figura 3.1.

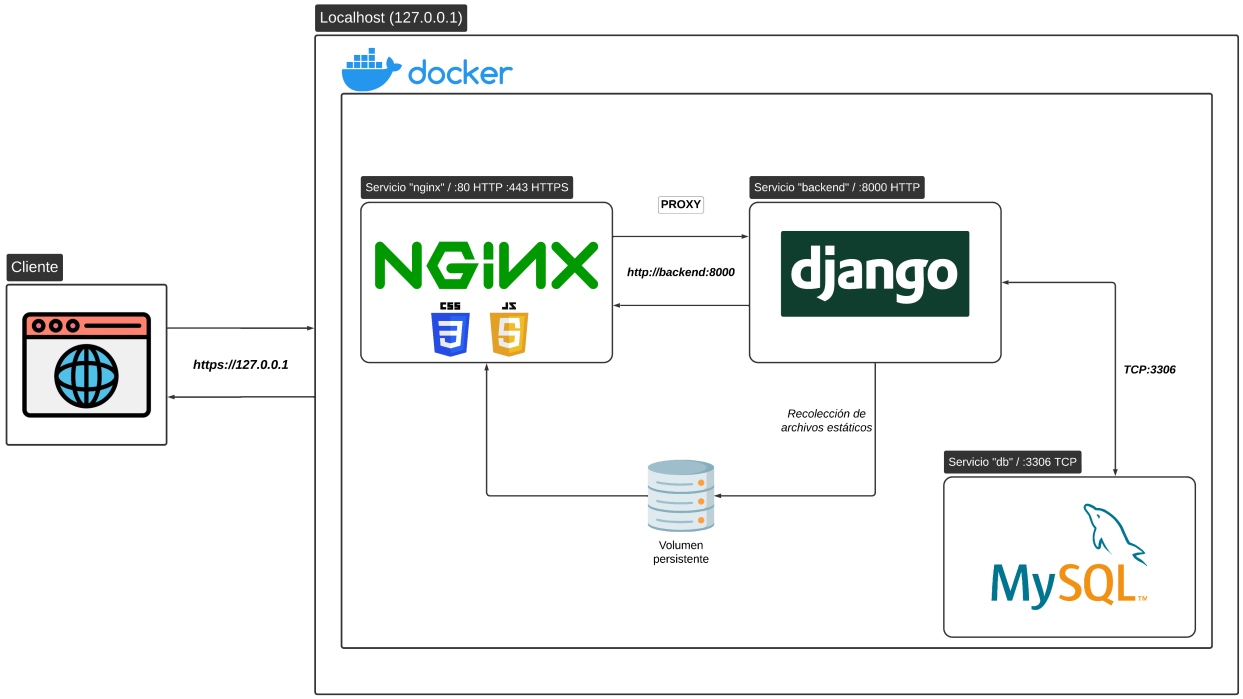

Figura 3.1: Arquitectura de la solución

3.2. Selección de las herramientas de escaneo

Debido a la existencia de múltiples herramientas de escaneo de vulnerabilidades para aplicaciones web y que cada una de ellas posea su propio estándar y formato para generar los reportes de hallazgos, se hace necesaria la elección de al menos dos de ellas para poder importar información a la plataforma.

Se examinaron varias opciones considerando tres aspectos principales: que permitieran generar reportes en formato JSON, que fueran de uso gratuito y que estuvieran disponibles para múltiples plataformas. Se decidió JSON como el formato estándar para los importes

debido a su fácil uso y gran compatibilidad con la mayoría de tecnologías modernas gracias a su popularidad. Se consideró la gratuidad de las herramientas para que la plataforma pueda ser utilizada por una amplia gama de usuarios, sin la necesidad de incurrir en costos adicionales. Por último, se evaluó la disponibilidad en múltiples plataformas para garantizar que las herramientas puedan ser utilizadas en diversos entornos de desarrollo, demostrando su versatilidad y ampliando el rango de usuarios potenciales.

Con estos tres aspectos, se decidió habilitar las siguientes herramientas para el importe de escaneos de vulnerabilidad:

OWASP Zed Attack Proxy (o ZAP por sus siglas) [26]. ZAP es una herramienta de uso gratuito y código abierto diseñada para realizar pruebas de penetración en, específicamente, aplicaciones web. Funciona como un servidor intermediario, o *proxy*, entre el cliente y el servidor de la aplicación para interceptar, inspeccionar y modificar el contenido de los mensajes entre estos. Se seleccionó debido a su soporte multiplataforma (Linux, MacOS, Windows), su fácil utilización y al amplio abanico de alertas que puede reportar. Su gran popularidad y amplia comunidad de usuarios garantizan un soporte constante y actualizaciones regulares, lo que la convierte en una herramienta confiable. Además, posee respaldo por parte de OWASP (Open Web Application Security Project), una autoridad en la comunidad de seguridad web.

Wapiti [18] es una herramienta de código abierto gratuita que realiza escaneos de tipo caja negra funcionando como un *web crawler* y *fuzzer*. Navega a través de las páginas de la aplicación web buscando donde poder inyectar datos y una vez obtiene las direcciones, actúa como *fuzzer* introduciendo datos inválidos en busca de vulnerabilidades. Permite detectar una amplia variedad de vulnerabilidades gracias a sus modelos de búsqueda. Además, es de fácil uso al ser una aplicación de línea de comando y es ampliamente conocida y usada para auditar aplicaciones web.

3.3. Asignación de nota de seguridad

Para lograr asignar una nota de seguridad a cada aplicación se decidió utilizar la cantidad de vulnerabilidades presentes para crear un puntaje de seguridad. Este puntaje se calcula mediante una suma ponderada, asignando un peso a cada nivel de severidad, de la siguiente forma:

$$Puntaje_{seguridad} = Peso_l \cdot Cantidad_l + Peso_m \cdot Cantidad_m + Peso_h \cdot Cantidad_h$$

Donde $Peso_l$, $Peso_m$ *y* $Peso_h$ son los pesos (mayores o iguales a cero) asignados a cada nivel de criticidad de vulnerabilidad, permitiendo establecer una importancia relativa a cada categoría en el cálculo del puntaje total. Y $Cantidad_l$, $Cantidad_m$ *y* $Cantidad_h$ representan la cantidad de vulnerabilidades identificadas en la aplicación para cada nivel de criticidad (Bajo, Medio, Alto y Crítico), ofreciendo un indicador cuantitativo para establecer el nivel de seguridad de la aplicación.

Se puede ver que esta función entrega valores en un rango de 0 a infinito, por lo que se vuelve necesario emplear una estandarización sobre los valores obtenidos para trasladar el resultado al rango [0, 10]. Sin embargo, debido a la naturaleza de la ciberseguridad, no

hay un límite, o cota, superior de vulnerabilidades, por lo que no existe un valor máximo de vulnerabilidades a hallar en un código, ni se posee un set de datos sobre el cual obtener métricas que permita realizar la estandarización de los valores. Es por ello que se optó por diseñar una función que se acerque al valor 10 a medida que el puntaje se hace infinito y que sea creciente para el dominio de puntajes ($\Re^+$): $Puntaje_{estandarizado} = min(10, \frac{10 \cdot x}{K+x} \cdot Apetito_{riesgo})$, donde x es el puntaje obtenido en la función previa, K es una constante positiva que permite aumentar o disminuir el ritmo en que la función crece. Y, por último, $Apetito_{riesgo}$ refleja el nivel de tolerancia al riesgo configurado para la aplicación, permitiendo adaptar el puntaje estandarizado según las preferencias específicas de seguridad de cada usuario.

Con las funciones de puntaje y nota elaboradas, resta únicamente definir los coeficientes de peso asociados a cada categoría de vulnerabilidad y establecer el valor de la constante K. Para el caso de la constante K, se decidió el valor 20, basándose en diferentes gráficos creados para la función con variaciones de valor para el parámetro, obtenidos a partir de posibles valores de puntaje. Valores muy pequeños para K hacen que el crecimiento de la función sea muy rápido, mientras que valores demasiado altos hacen que el crecimiento sea desmedidamente lento, el valor 20 para K es un punto bastante central en este *trade-off*.

Para asignar los pesos a cada tipo de severidad de vulnerabilidad se consideraron dos conjuntos de valores: [1, 2, 3, 4] y [1, 3, 6, 10], correspondientes a las severidades Baja, Media, Alta y Crítica, respectivamente. La elección de estos conjuntos depende de como se desea reflejar cada severidad en la nota final de seguridad de la aplicación. La primera opción ofrece un crecimiento lineal entre los pesos, indicando que cada severidad tiene un impacto incremental consistente. Sin embargo, la diferencia entre las severidades no es pronunciada, lo que no refleja adecuadamente el riesgo que representan las vulnerabilidades de tipo Crítico o Alto en comparación con las de tipo Bajo o Medio. El segundo conjunto, con un incremento más pronunciado y casi exponencial, ofrece una representación más dramática y posiblemente más fiel a la realidad de las diferencias de riesgo entre las severidades. Aunque esto puede reflejar de manera más precisa la gravedad de las vulnerabilidades de tipo Alto y Crítico, también puede exagerar su impacto, ya que una baja cantidad de estas vulnerabilidades podría resultar en una nota de seguridad significativamente alta.

Bajo esta comparación, se optó por utilizar la segunda opción para calcular el puntaje de seguridad, ya que esta solución busca mejorar el estado de seguridad de las aplicaciones desarrolladas por organizaciones y usuarios. Por lo tanto, es fundamental representar de la manera más fiel posible la realidad del riesgo presente en los diferentes tipos de severidad. Utilizando el conjunto de pesos más pronunciado, se puede reflejar con mayor precisión la gravedad de las vulnerabilidades críticas y altas, incentivando así una mayor atención y esfuerzo en su mitigación.

Es relevante señalar que el proceso de asignación de valores a los coeficientes y la formulación de las funciones de puntaje presentadas son áreas propensas a mejoras en el futuro. Se contempla la posibilidad de perfeccionar y ajustar estos aspectos mediante el empleo de técnicas de ciencia de datos cuando se disponga de una base de datos debidamente poblada. Gracias a la metodología de trabajo utilizada y la modularización de los componentes de la aplicación, es posible modificar la función de asignación de la nota de seguridad sin riesgo de afectar otros componentes o el funcionamiento de la aplicación. Esto permite una refinación continua de los modelos y una adaptación más precisa a las características de los datos

recopilados.

3.4. Panel de control

Dentro del panel de control de la aplicación, se planea proporcionar *insights* al usuario sobre el nivel de seguridad de su aplicación. Además de la nota de seguridad, se busca que el usuario pueda ver la cantidad de vulnerabilidades de diferentes severidades que su aplicación contiene, la cantidad de vulnerabilidades nuevas ingresadas en una fecha específica y la cantidad de vulnerabilidades mitigadas a lo largo del tiempo. También se planea identificar las vulnerabilidades más recurrentes dentro de la aplicación, entre otros aspectos. Con este objetivo en mente, se definieron las siguientes métricas y gráficos a implementar:

3.4.1. Gráficos

- **Gráfico de barras de severidad a lo largo del tiempo:** Este gráfico es crucial para entender cómo ha evolucionado la seguridad de la aplicación con el tiempo y como las severidades de las vulnerabilidades aumentan y disminuyen en determinados periodos. Permite identificar patrones y tendencias de la distribución de las vulnerabilidades,
- **Gráfico de torta de vulnerabilidades activas versus mitigadas:** Un gráfico de torta proporciona una representación clara y concisa del estado actual de la seguridad de la aplicación, permitiendo al usuario visualizar las proporciones de vulnerabilidades que aún no han sido abordadas y cuántas han sido mitigadas en un solo vistazo. Su simplicidad y claridad lo hace efectivo para comunicar el estado de seguridad, facilitando la toma de decisiones informada.
- **Gráfico de tendencia de vulnerabilidades en el tiempo:** Un gráfico de tendencia permite hacer seguimiento de la evolución de la cantidad de vulnerabilidades a lo largo del tiempo. Esto lo vuelve crucial para entender cómo varía el riesgo de seguridad de la aplicación en el tiempo y si hay tendencias específicas que necesiten ser abordadas. Permite evaluar la efectividad de las prácticas de seguridad implementadas a lo largo del tiempo mediante la identificación de patrones o ciclos. Por ejemplo, se podrían notar picos de vulnerabilidades en ciertos periodos de tiempo que coincidan con nuevas versiones de la aplicación, u observar una disminución constante de estas luego de implementar medidas de seguridad o realizar auditorías. La visualización clara de la evolución temporal de las vulnerabilidades ayuda a destacar la importancia de la seguridad y respalda la toma de decisiones informada.
- **Gráfico de torta de severidades:** Con esta visualización se suministra una representación rápida de la proporción de severidad de las vulnerabilidades. Con ello, el usuario podría estimar un nivel de seguridad basado en severidad, considerando un porcentaje mínimo, o máximo, que un tipo de severidad podría llegar a ocupar en su aplicación.
- **Gráfico de barras apiladas de vulnerabilidades activas versus mitigadas en el tiempo:** Un gráfico de barras apiladas permite visualizar como va el proceso de

mitigación de vulnerabilidades para un reporte dado en una fecha específica. Con esto, el usuario puede tener una vista general del estado de mitigación de todos los reportes que se han importado a la aplicación.

- **Gráfico de barras de tipo de vulnerabilidad:** Permite visualizar qué tipos de vulnerabilidades son las más comunes en la aplicación, lo que puede indicar a su vez qué área de la aplicación es la más débil. Con ello, se pueden priorizar medidas de seguridad específicas para abordar las vulnerabilidades más prevalentes.

Los gráficos mencionados se pueden visualizar en el anexo A.1.

3.4.2. Métricas

- **Nota de seguridad:** Nota de seguridad, entre 0 y 10, basada en la cantidad de vulnerabilidades de la aplicación. Mientras más alta, peor es la seguridad de la aplicación. Permite al usuario tener una referencia del nivel de seguridad de su aplicación, basándose únicamente en las vulnerabilidades presentes en esta.
- **Tasa de mitigación:** Tasa de vulnerabilidades mitigadas por día en un periodo de tiempo designado por el usuario. Esta tasa permite calcular la cantidad de días necesarios para mitigar todas las vulnerabilidades activas en ese momento y estimar la fecha en que se lograría dicha mitigación total.
- **Índice de severidad:** Tipo de severidad con la mayor frecuencia (moda estadística).
- **Severidad máxima:** La severidad más alta de las vulnerabilidades activas de la aplicación.

3.5. Back-end

Para el desarrollo del *back-end* de la plataforma se utilizó Django, un *framework* de desarrollo web de código abierto basado en el lenguaje de programación Python. Django es conocido por su robustez y flexibilidad, permitiendo un desarrollo rápido y eficiente de aplicaciones web. Pero la elección de esta herramienta no se debe solo a sus capacidades técnicas, sino también a la gran popularidad de Python en el desarrollo web hoy en día. Siendo este uno de los lenguajes de programación más populares como se muestra en el TIOBE Index [27] y la Stack Overflow Developer Survey [20]. Esta popularidad da la seguridad de una amplia comunidad y una abundancia de recursos y bibliotecas de código, facilitando el desarrollo, mantenimiento y evolución del proyecto. En esta sección se detallan las principales funcionalidades del *back-end*, incluyendo la estructura del proyecto y los servicios que proporcionan la lógica de negocio necesaria para el seguimiento de vulnerabilidades en aplicaciones web.

El *back-end* de la plataforma se desarrolló mediante la creación de dos aplicaciones principales: **“App-Web”** y **“Users”**.

3.5.1. App-Web

Esta aplicación contiene todo lo relacionado con los servicios y la lógica de negocio de la aplicación. Se encarga de gestionar las vistas, modelos, formularios y permisos necesarios para el funcionamiento de la plataforma. Mediante el uso de las vistas de Django, se procesan las solicitudes web y se devuelven las respuestas adecuadas, ya sea en páginas HTML o datos en formato JSON. Dentro de esta aplicación se gestionan las operaciones "críticas" relacionadas con el seguimiento de las vulnerabilidades, como la recepción y procesamiento de los reportes generados por las herramientas de escaneo, la actualización del estado de las vulnerabilidades y la generación de métricas a partir de los datos levantados por los escaneos. Esta aplicación también integra los *endpoints* de la API necesarios para la carga de los reportes por parte del usuario. En resumen, esta aplicación actúa como el núcleo de la plataforma, orquestando las operaciones esenciales que permiten al usuario realizar el seguimiento de las vulnerabilidades presentes en sus aplicaciones.

3.5.2. Users

La aplicación *"Users"* se encarga de gestionar todo lo relacionado con el registro, inicio/-cierre de sesión y autorización de los usuarios en la plataforma. Garantiza que solo usuarios autenticados y autorizados puedan acceder a las funcionalidades críticas de la aplicación, proporcionando control y seguridad.

Separar esta lógica en una nueva aplicación de Django da permiso a una mayor escalabilidad de la aplicación. Se vuelve más fácil realizar mejoras, agregar nuevas funcionalidades o escalar partes específicas de la plataforma sin afectar al resto del sistema. Esta modularización facilita la reutilización del código en otros proyectos o aplicaciones, pues puede ser fácilmente adaptada en otros desarrollos, ahorrando tiempo y esfuerzo en el trabajo. También permite auditar de manera más eficiente los controles de acceso y genera de forma más ágil políticas de seguridad dirigidas para los usuarios, beneficiando el desarrollo y mantenimiento de la plataforma.

3.5.3. API

Para que los usuarios puedan importar sus escaneos de vulnerabilidad de manera eficiente, se decide implementar una API. Esta tecnología ofrece la comodidad de permitir a los usuarios mantener sus desarrollos y hacer seguimiento de sus aplicaciones sin necesidad de ingresar directamente a la aplicación web. Al utilizar una API, se promueve el desarrollo continuo, ya que los usuarios pueden subir sus archivos de escaneo mediante el uso de una terminal o cualquier software de preferencia que permita hacer una solicitud POST al *endpoint* correspondiente, junto con un *token* de acceso. Esto facilita la integración con flujos de trabajo existentes y herramientas personalizadas, como *pipelines* de CI/CD, mejorando significativamente la eficiencia y la experiencia del usuario.

Para desarrollar la API, se ha optado por utilizar Django REST Framework (DRF), una

herramienta que facilita la creación de APIs *RESTful* con Django. DRF es conocida por su flexibilidad y robustez, permitiendo la creación de *endpoints* de manera rápida y eficiente. En este caso, se crearán los siguientes tres *endpoints* necesarios para interactuar con la aplicación:

- Carga de Archivos: Permite a los usuarios subir archivos de escaneo directamente a la plataforma.
- Obtención de Token de Acceso: Proporciona a los usuarios un *token* de acceso necesario para autenticar las solicitudes a la API.
- Actualización de Token de Acceso: Permite a los usuarios actualizar su *token* de acceso cuando lo requieran.

Para garantizar la seguridad y control de acceso a los datos sensibles que los escaneos suponen, se implementa un sistema de autenticación basado en JSON Web Tokens (o JWT, por sus siglas) [19]. Un JWT es un estándar abierto (RFC 7519) [13] que define una forma compacta y autónoma de transmitir información de manera segura entre partes como un objeto JSON. Este método de autenticación asegura que solo los usuarios autenticados puedan acceder a los *endpoints*, proporcionando un token de acceso que se envía en cada solicitud. La implementación de JWT no solo simplifica la gestión de sesiones, sino que también facilita la autorización de usuarios, permitiendo una verificación eficiente en el lado del cliente y favoreciendo el desarrollo de los usuarios.

Al utilizar JWT, se garantiza que cada solicitud a la API esté acompañada de un *token* válido, lo que protege los datos sensibles y asegura que solo los usuarios autorizados puedan subir y gestionar sus escaneos. Esta capa adicional de seguridad es crucial para mantener la integridad y confidencialidad de los datos.

En resumen, la implementación de esta API *RESTful* proporciona un recurso seguro, escalable y fácil de mantener para que los usuarios puedan subir sus reportes y realizar el seguimiento de vulnerabilidades. Al integrar herramientas como Django REST Framework y sistemas de autenticación basados en JWT, se asegura una experiencia de usuario óptima y una gestión eficiente de la seguridad.

3.6. Modelo de datos

Dada la naturaleza del problema que se aborda, se pueden notar cuatro entidades esenciales:

1. User: Usuario que busca realizar el seguimiento de vulnerabilidades en sus aplicaciones.
2. Application: La(s) aplicación(es) que el usuario busca hacer seguimiento.
3. Scan: El/los escaneo(s), o reporte(s), de seguridad que una, o más, herramienta(s) de escaneo de vulnerabilidades realiza(n) sobre la aplicación.

4. Vulnerability: La(s) vulnerabilidad(es) que el/los escaneo(s) reporta(n) en el proceso.

Para gestionar la información proporcionada por estas entidades se requiere de un sistema que garantice la integridad de los datos almacenados y defina un esquema relacional fijo para estos. Los datos que la plataforma deberá manejar son altamente estructurados y con una estrecha relación, su escalado no es "horizontal" ni requiere de flexibilidad en la estructura, por lo que un esquema no relacional no sería la mejor opción para la naturaleza del caso. Este tipo de diseño, en donde las entidades están altamente relacionadas en un entorno altamente estructurado, da un indicio de que una base de datos relacional, que respete los principios *ACID*, es la solución adecuada para gestionar la información.

3.6.1. Estableciendo la relación entre los datos

Teniendo en mente la utilización de una base de datos relacional, falta establecer el tipo de relación entre las diferentes entidades para asegurar la gestión eficiente y coherente de los datos. En este contexto, la relación de las entidades puede ser descrita de la siguiente forma: un usuario puede crear su cuenta y no asociar ninguna aplicación, o bien, puede crear una o más aplicaciones para hacer seguimiento. Esto define una relación de cero a muchos entre las entidades *"User"* y *"Application"*. Del mismo modo, cada aplicación puede ser sometida a múltiples escaneos de seguridad en su desarrollo y mantenimiento, o a ninguno, estableciendo así una relación de cero a muchos entre *"Application"* y *"Scan"*. Por último, al realizar un escaneo sobre una aplicación, puede darse la posibilidad de que no se encuentren vulnerabilidades debido a la seguridad implementada en esta, o se pueden reportar una o más vulnerabilidades. Nuevamente, se tiene una relación de cero a muchos, esta vez entre la entidad *"Scan"* y *"Vulnerability"*.

En la figura 3.2 se puede ver un diagrama de entidad-relación que muestra la relación previamente definida para las entidades de la base de datos de la solución y los campos que cada una de estas define.

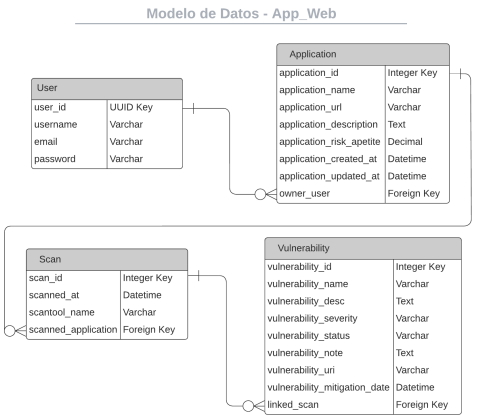

Figura 3.2: Esquema entidad-relación de la base de datos

3.6.2. Sistema de bases de datos

Para la administración de la base de datos de esta aplicación se optó por MySQL debido a sus múltiples ventajas. Si bien Django ofrece por defecto una base de datos gestionada con SQLite, esta opción presenta algunas limitaciones que no la hacen del todo adecuada para esta aplicación. SQLite es un gestor ideal para proyectos de pequeña escala, pero carece de la robustez y escalabilidad necesaria para un alto rendimiento y gestión de grandes volúmenes de datos.

Una vez descartado SQLite, las opciones principales fueron PostgreSQL y MySQL. Ambas tecnologías son altamente utilizadas por los desarrolladores, pero para este proyecto, se decidió por MySQL. Las razones se basan en las necesidades específicas del proyecto, como frecuentes lecturas de datos con consultas simples, en las que MySQL tiene un mejor desempeño que PostgreSQL. Además, MySQL es conocido por su facilidad de configuración, lo que facilita su implementación y mantenimiento. Por más que PostgreSQL propone un enfoque más empresarial ideal para proyectos de gran escala, ofreciendo características como almacenamiento de datos geométricos, direcciones de red, rangos, XML, escritura de datos optimizada, entre otros. MySQL ofrece la fiabilidad *ACID*, alto rendimiento y escalabilidad necesaria para esta plataforma, sin la sobrecarga de configuración y gestión que podría no ser necesaria para este proyecto en particular ni en su futuro.

Por estas razones, se prefirió utilizar MySQL, un sistema de gestión de bases de datos relacionales de código abierto. MySQL provee transacciones, vistas y procedimientos almacenados útiles para el manejo de los datos del proyecto, y su amplia adopción en la industria asegura una robusta comunidad de soporte, abundancia de recursos y amplia documentación. Esto hace que MySQL sea una elección idónea para gestionar la base de datos de la aplicación.

3.7. Front-end

Se comenzó con el diseño de las vistas preliminares para tener una base al momento de comenzar el desarrollo del *front-end* de la aplicación. Para esta tarea se utilizó Uizard [28], una herramienta para el diseño de interfaces de usuario. Con dicha herramienta se diseñaron mediante *wireframes* las diferentes páginas principales de la aplicación: página de bienvenida, *dashboard* de seguridad, página de aplicaciones del usuario, entre otras, formando así una base para el desarrollo de la interfaz de la aplicación. En el anexo A.2 se ahonda más y se muestran los *mockups* creados.

3.7.1. Tecnologías

El *front-end* de toda aplicación web cuenta con tres tecnologías principales: HTML, CSS y JavaScript. Cada una de estas dispone de un gran abanico de herramientas para agilizar su desarrollo debido al gran ecosistema que el desarrollo de aplicaciones web describe.

Para el caso de CSS se optó por utilizar Bootstrap [15], un *framework* CSS de código abierto desarrollado por Twitter en el año 2010. Este combina CSS, JavaScript y HTML para estilizar los elementos de una página web, proporcionando elementos personalizables ya construidos, requiriendo únicamente de la importación de la biblioteca de JavaScript y el archivo de extensión CSS para su uso en la aplicación del usuario. La elección de esta herramienta se debe a su diseño altamente responsivo y la fácil utilización y adaptación, con una instalación extremadamente rápida y una curva de aprendizaje muy empinada (en poco tiempo se logra aprender mucho).

Para HTML se decidió por la utilización del motor de Templates incluido dentro del *back-end* que Django suministra. Este motor proporciona un pequeño lenguaje para definir la interfaz de cara al usuario, manteniendo la separación entre la lógica de la aplicación y la de presentación, o *back-end* y *front-end*. El lenguaje permite generar documentos HTML, XML, CSV, entre otros, a partir de archivos de texto plano de una manera sencilla y poderosa mediante la utilización de tags, variables y filtros. Con estas herramientas que el lenguaje proporciona se pueden generar *templates* dinámicos de forma limpia. La razón para utilizar esta herramienta que Django proporciona y no otro *framework*, como podría ser Vue, React o Angular, se debe a que la aplicación únicamente busca presentar los datos de una manera clara para el usuario para facilitarle el proceso de seguimiento de las vulnerabilidades en su aplicación. Las interacciones que el usuario realiza con los datos son mínimas (operaciones *CRUD* y gráficos interactivos del panel de control), por lo que la utilización de otro *framework* para este apartado se convertiría en un *"overkill"*.

Por último, para JavaScript se decantó por la utilización de librerías del lenguaje en lugar de *frameworks*. Esta decisión se justifica de la misma forma que ocurre para la tecnología HTML. La aplicación requiere de JavaScript únicamente para la validación y securitización de *inputs* del usuario en la interfaz de la aplicación, junto al desarrollo de gráficos interactivos para el panel de control. Si bien, el uso de un *framework* acelera el desarrollo del software, proporcionando múltiples herramientas para la construcción de la aplicación, una pequeña parte de este kit que proporciona sería utilizado.

3.7.2. Servir contenido estático

El contenido estático es "cualquier archivo que se almacena en un servidor y es el mismo cada vez que se entrega a los usuarios"(Cloudflare, Inc., 2021).[4] Dentro de esta categoría se incluyen archivos HTML, *scripts* de JavaScript, imágenes y estilos CSS que una aplicación web utiliza. Estos archivos rara vez se editan o actualizan y no cambian durante la conexión con el usuario. Según Cloudflare, una práctica muy común en el desarrollo web es la utilización de un servidor *proxy* inverso para aumentar el rendimiento de la aplicación, o servidor origen (Cloudflare, Inc., 2021).[5] En particular, esto se logra haciendo que el servidor *proxy* almacene en caché los contenidos estáticos de la aplicación. De este modo, al momento en que el cliente envía su solicitud, el servidor *proxy* responde inmediatamente con los archivos estáticos, aliviando así al servidor origen de esa carga adicional. Esto no solo disminuye el tráfico de la red, sino que también mejora el rendimiento de las respuestas.

Dado lo anterior, se ha decidido utilizar NGINX, un servidor web de código abierto, para

configurar un servidor *proxy* inverso que contenga los archivos estáticos de la aplicación y redirija las solicitudes al servidor origen, o *back-end*. De este modo, se asegura un mejor rendimiento de la aplicación frente a múltiples solicitudes y permite mantener los principios *SOLID* del desarrollo de software, separando esta tarea del servidor *back-end*. NGINX es conocido por su eficiencia en la gestión de grandes volúmenes de conexiones simultáneas y su capacidad de servir contenido estático de manera rápida y eficiente. Al implementar esta configuración se logra un equilibrio entre rendimiento y escalabilidad, garantizando que la aplicación pueda manejar picos de tráfico sin comprometer la calidad del servicio ofrecido.

3.8. Despliegue de la Aplicación

Se decidió evitar poner a disposición la aplicación como un modelo *SaaS* (Software as a Service) mediante un servidor expuesto a Internet para poder dar alcance a mayor cantidad de usuarios y preservar la naturaleza *open source* del desarrollo. La aplicación maneja datos sensibles y críticos, ya que trata con información sobre vulnerabilidades en los desarrollos de los usuarios. Esto la convierte en un posible objetivo para ciberataques, pues es una fuente de información importante, y exponerla a Internet requeriría una implementación exhaustiva de sistemas de seguridad, lo cual excede los objetivos y recursos disponibles para este trabajo. Utilizar un modelo *SaaS* puede ser riesgoso, ya que cualquier filtración de datos podría perjudicar gravemente a los usuarios, disuadiendo a muchos de utilizar la aplicación debido al *trade-off* entre los beneficios que el seguimiento de vulnerabilidades ofrece versus la exposición de información crítica en Internet. Al evitar este modelo, se incrementa el alcance potencial de la aplicación, ya que los usuarios pueden desplegarla en sus propios entornos controlados, asegurando que la información crítica se mantenga segura bajo sus propios métodos y parámetros.

Bajo esta perspectiva, se eligió realizar el despliegue utilizando Docker, una tecnología que empaqueta software en unidades estandarizadas denominadas contenedores. Docker facilita la portabilidad y replicabilidad de la aplicación al permitir que esta se ejecute de manera consistente en diferentes entornos. Al utilizar contenedores, se encapsulan todas las dependencias, bibliotecas y configuraciones necesarias, asegurando que la aplicación funcione del mismo modo, sin importar en donde se ejecute. El uso de esta tecnología no solo facilita el despliegue, sino que también simplifica el proceso de mantenimiento y actualización de la aplicación, favoreciendo la naturaleza *open source* que se busca adoptar.

El despliegue de la aplicación se realizará mediante la creación de imágenes Docker para los tres componentes clave: *back-end*, Base de Datos y *front-end*. Para el *back-end*, se utilizará una imagen basada en la distribución Alpine de Linux, con Python3 instalado para ejecutar el proyecto de Django, y Gunicorn para manejar las solicitudes. La base de datos se desplegará utilizando una imagen estándar de MySQL. El *front-end* se implementará con una imagen de Nginx, que servirá como servidor *proxy* y se encargará de los archivos estáticos. Estas tres imágenes se utilizarán en tres servicios separados, los cuales se configurarán y levantarán utilizando Docker Compose, una herramienta que permite orquestar múltiples contenedores. Se establecerán dos redes: una red interna, que no tendrá exposición a Internet, y una red externa, que sí estará expuesta a Internet. La base de datos residirá exclusivamente en la red

interna, lo que la hará accesible únicamente desde los otros contenedores en la misma red. El *front-end* se ubicará únicamente en la red externa, exponiendo la aplicación web para que los usuarios puedan acceder a ella desde sus navegadores. Por último, el *back-end* actuará como un nexo entre las dos redes, ya que estará presente en ambas. Esto le permitirá comunicarse con la base de datos para gestionar la información y responder a las solicitudes que lleguen a través del servidor *proxy* de Nginx.

Capítulo 4

Implementación

En este capítulo se detalla el desarrollo de la plataforma desde el ámbito técnico. Se abarcan los aspectos fundamentales de cada sección, describiendo las decisiones de diseño adoptadas, las configuraciones utilizadas y los desafíos encontrados. También, se profundiza el proceso de despliegue de la aplicación, explicando como se llevó a cabo este, describiendo los servicios creados, entre otros. Además, se analizan las medidas y configuraciones de seguridad implementadas, explicando sus funciones y cómo contribuyen a la securitización de la plataforma.

4.1. Desarrollo de la plataforma Web

Esta sección detalla el proceso de creación y configuración de los diversos componentes que constituyen la plataforma. Esta sección abarca el desarrollo del *back-end*, la implementación de la API, el diseño del *front-end* y las interfaces de usuario, así como la configuración del servidor *proxy*. Se especifica la estructura de los archivos del proyecto, la conexión a la base de datos, y los métodos de autenticación implementados para la API. Además, se describe el desarrollo de las plantillas HTML, la integración de estilos y scripts, y otros aspectos cruciales que garantizan el correcto funcionamiento y la usabilidad de la plataforma.

4.1.1. Back-End

Se comienza con el desarrollo del Back-End según lo diseñado en la sección 3.5.

Conexión a la Base de Datos

Para la conexión a la base de datos de MySQL se utilizó "mysqlclient", una librería que implementa una API de acceso a la base de datos gestionada por MySQL. Para ello, se estableció la configuración de conexión dentro de las configuraciones del proyecto de Django, como se muestra en el código 4.1.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': os.environ.get("DB_NAME"),
        'USER': os.environ.get("DB_USER"),
        'PASSWORD': os.environ.get("DB_PASSWORD"),
        'HOST': os.environ.get("DB_HOST", "db"),
        'PORT': os.environ.get("DB_PORT", "3306"),
    }
}
```

Listing 4.1: Conexión a la base de datos

Variables de Entorno

Los valores de configuración necesarios para la implementación de la aplicación se almacenaron en variables de entorno establecidas dentro de un archivo `.env`, cómo se puede ver en el código 4.2. De este modo se logra evitar el uso de credenciales en formato duro dentro del código, manteniendo una configuración limpia y personalizable a lo largo del desarrollo que promueve la portabilidad y flexibilidad de la aplicación, y evitando filtraciones de información.

```
# Environment variables for the Django project

# Database
DB_NAME=your_db_name
DB_ROOT_PASSWORD=your_db_root_password
DB_USER=your_db_user
DB_PASSWORD=your_db_user_password

# If you modify the following variables, you must modify the docker-
    compose.yml file
DB_HOST="db"
DB_PORT="3306"

# Django settings
SECRET_KEY=your_secret_key
DEBUG="False"
ALLOWED_HOSTS=localhost 127.0.0.1 [::1]
```

Listing 4.2: Variables de entorno

Estructura de las Aplicaciones

El directorio raíz del proyecto contiene tanto el proyecto Django como los archivos de configuración necesarios para su despliegue, incluyendo los archivos `.env` para la configuración de variables de entorno, los Dockerfiles para la creación de imágenes Docker, y el archivo `docker-compose.yml` para orquestar los contenedores. Además, alberga los archivos del repositorio que gestionan el control de versiones del proyecto.

El desarrollo se organizó en directorios separados para cada una de las aplicaciones explicadas en la sección 3.5. Esta estructura de directorios incluye una carpeta "`static`" para los archivos estáticos de la aplicación y una carpeta "`templates`" para las plantillas de HTML. Los archivos de implementación de Django se encuentran en el directorio raíz, como se ilustra en la figura 4.1.

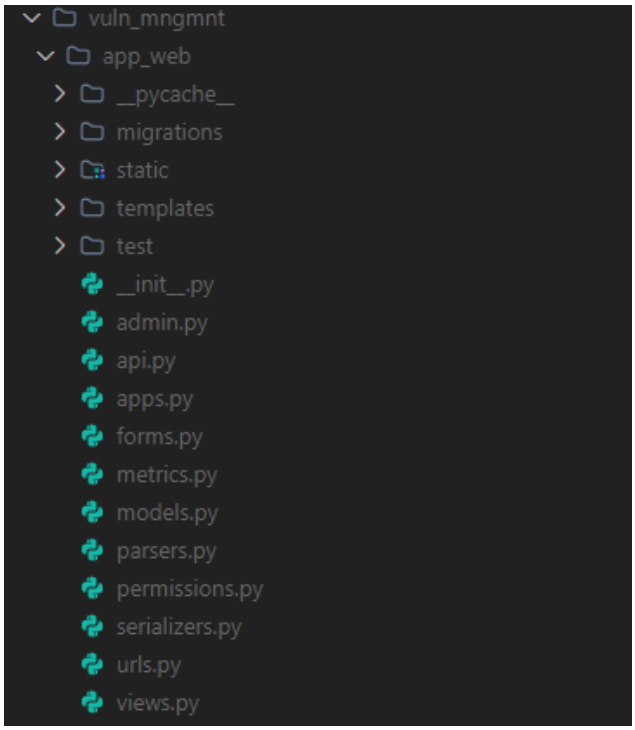

Figura 4.1: Estructura archivos para aplicaciones

Dentro de los archivos proporcionados por Django se encuentran `models.py`, `views.py` y `urls.py`. En primer lugar, en `models.py` se implementan los modelos de datos descritos en la sección 3.6 y detallados en el anexo A.3.

En segundo lugar, se encuentra el archivo `views.py` en el cual se describen y define la lógica de cada *endpoint* de la aplicación. Los *endpoints* fueron creados utilizando funciones de vistas de Django en lugar de clases, lo que permite una implementación más directa y sencilla. Estas vistas fueron diseñadas y desarrolladas pensando en habilitar las operaciones *CRUD (Create, Read, Update, Delete)* sobre los objetos de los modelos de la aplicación. Es por ello que cada entidad posee vistas específicas para crear nuevos objetos, actualizar datos del objeto, visualizar objetos y eliminarlos. Por ejemplo, la vista `new_application_view` permite agregar nuevas aplicaciones, mientras que `update_application_view` se encarga de actualizar la información de una aplicación existente.

Además de las operaciones *CRUD*, se implementaron vistas para manejar otras funcionalidades de la plataforma. Entre ellas `inicio_view`, encargada de renderizar la página de inicio de la aplicación, y `dashboard_view` que permite acceder al panel de control de una aplicación específica para poder visualizar las métricas y gráficos relacionados. Algunas de estas vistas utilizan *AJAX* para poder actualizar partes de la página sin la necesidad de tener que recargarla completamente.

Cada una de estas funciones, o vistas, tienen como objetivo manejar las solicitudes HTTP de forma eficiente y responder con las plantillas HTML correspondientes, utilizando la renderización de Django. En el código A.1 del anexo A.4.1 se pueden visualizar las vistas definidas para la aplicación.

Por último, se tiene el archivo `urls.py` encargado de establecer las rutas de la aplicación para los *endpoints*, realizando un mapeo de las vistas a las rutas que se definen. Para elegir los nombres de las rutas se optó por un diseño simple y autocontenido, que solo con leerlo sea posible entender que es lo que se realiza en dicha vista, anteponiendo el nombre de la aplicación en el directorio principal. El código 4.3 muestra un ejemplo de como se desarrollaron dichas rutas.

```
/app/ # Inicio de la aplicacion
/app/aplicaciones/<int:id>/dashboard # Dashboard de una aplicacion
/users/registrarse # Registro de usuarios
/users/login # Inicio de sesion de usuarios
/app/api/subir-escaneo # Ruta para subir escaneos mediante API
```

Listing 4.3: Patrones de las rutas de los endpoints

Además de estos tres módulos principales proporcionados por Django, se desarrollaron varios módulos adicionales específicos para las necesidades de la aplicación: `parsers.py`, `permisions.py`, `metrics.py` y `serializers.py`. Los detalles de estas implementaciones se pueden encontrar en el anexo A.4.2. Sin embargo, es importante mencionar cuál es el objetivo de cada uno de estos archivos.

En `parsers.py`, se implementaron los *parseadores* de reportes de escaneo utilizados por la API. En este archivo se declaran las funciones encargadas de interpretar y extraer los datos necesarios de los reportes generados por las herramientas de escaneo, transformándolos en un diccionario para luego ser almacenados y utilizados por la aplicación.

El archivo `permissions.py` contiene los decoradores que verifican los permisos dentro de las vistas. Un ejemplo es el decorador "user_is_application_owner", el cual verifica que el usuario que realiza una solicitud sobre un objeto es el propietario de la aplicación u objeto en cuestión. Esto asegura que solo el dueño de una aplicación pueda modificar o eliminar los datos asociados a esta.

En `metrics.py`, se implementaron las funciones encargadas de calcular las métricas que se muestran en el *dashboard*. Entre ellas, el puntaje y nota de seguridad asociado a la aplicación.

Por último, `serializers.py` define los serializadores para los modelos `Scan` y `Vulnerability`. Estos tienen como función convertir los objetos de estos modelos a formato JSON y viceversa, facilitando el guardado de los datos procesados por los *parseadores*.

Manejo de usuarios

El manejo de usuarios se realizó generando una extensión de la clase "AbstractUser" de Django con el fin de únicamente de agregar un nuevo campo para poder almacenar el correo electrónico de los usuarios, como se muestra en el código 4.4.

```python
from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    """Model to store user data."""
    user_id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)

    def __str__(self) -> str:
        """Function to return the string representation of the user.

        Returns:
            str: The string representation of the user. It contains the username.
        """
        return self.username
```

Listing 4.4: Implementación del modelo User

La clase "AbstractUser" de Django provee todos lo necesario para poder registrar y autenticar usuarios en la aplicación, por lo que únicamente basto implementar las vistas que manejaran dichos casos. Estas últimas se encuentran detalladas en el anexo A.4.1.

4.1.2. API

El desarrollo de la API se realizó en un módulo de Python dedicado exclusivamente a esta funcionalidad, `api.py`, manteniendo la modularidad y organización del código. Se implementó una clase `FileUploadAPI` que hereda de la clase `APIView` de Django REST Framework (DRF). Las vistas en DRF funcionan de la misma forma que las vistas en Django, por lo que bastó con implementar métodos específicos, conocidos como "*handlers*", para las solicitudes de tipo `GET` y `POST`.

En la clase `FileUploadAPI`, se configuró la necesidad de estar autenticado para establecer la comunicación, asegurando que únicamente los usuarios autenticados puedan interactuar con la API. Esta autenticación se implementó utilizando *tokens* JWT (JSON Web Tokens), como se detalló en la sección 3.7.1.

Además, se añadieron validaciones y controles adicionales dentro de los métodos `GET` y `POST` para asegurar que las solicitudes cumplan con los requisitos esperados. Esto incluyó la validación de los archivos subidos, el manejo adecuado de los errores y la respuesta con códigos HTTP apropiados en caso de fallos. La modularidad y claridad en el diseño del módulo `api.py` facilitan la ampliación y mantenimiento de la API en el futuro.

Para manejar excepciones en la API, se crearon excepciones personalizadas utilizando la clase `APIException` de Django REST Framework. Esta clase proporciona una manera eficiente de crear nuevas excepciones para las APIs, permitiendo modificar los mensajes de error, el código de estado HTTP y un código identificador para la excepción. Se creó una nueva clase `CustomAPIException`, que hereda de `APIException`, y se modificó la función `__init__` para recibir códigos de estado y mensajes de error personalizables como parámetros. Los casos abordados incluyen situaciones en las que no se provee un archivo en la solicitud `POST` a la

API, cuando el reporte de escaneo entregado ya ha sido importado previamente, cuando el reporte no contiene ninguna vulnerabilidad nueva, cuando ocurre un error al intentar parsear el reporte recibido, cuando hay un error al intentar serializar los datos y cuando ocurre un error al intentar guardar la información del escaneo o de alguna vulnerabilidad. En el anexo A.7 se pueden observar las excepciones creadas con mayor detalle.

El método `GET` permite obtener los datos de los escaneos (sin incluir las vulnerabilidades) importados a la plataforma para una aplicación específica. Para ello, el método recibe el *ID* de la aplicación como parámetro en la solicitud. Primero, busca la aplicación en la base de datos; si no se encuentra, redirige a la vista de error 404. Luego, verifica que el usuario autenticado mediante el *token* JWT sea el propietario de la aplicación consultada; si no es así, se devuelve una excepción "`PermissionDenied`" con el código de estado 403. Si la aplicación y usuario son válidos, el método obtiene todos los escaneos asociados a la aplicación, los convierte en un objeto JSON utilizando el serializador del modelo `Scan` y devuelve el resultado en la respuesta. En caso de que ocurra algún error al intentar serializar los datos de los escaneos, se devuelve una excepción con el código 500, informando sobre el problema de serialización.

Para el método `POST` se requiere que la solicitud incluya el *ID* de la aplicación, el nombre de la herramienta utilizada y los *bytes* del archivo JSON codificados en un formato "`multipart/form-data`" en la solicitud. Se crearon *parsers* para las dos herramientas aceptadas por la aplicación, "Wapiti" y "OWASP ZAP", con el fin de generar un diccionario estandarizado que contenga toda la información necesaria para importar los datos a la plataforma. Además, se desarrollaron funciones auxiliares para manejar los casos excepcionales descritos anteriormente, las cuales se detallan en el anexo A.7.1.

Utilizando estas herramientas, el método `POST` busca la aplicación referida por el usuario mediante el *ID* proporcionado; si no la encuentra, redirige a una vista con el código de estado 404. Una vez localizada la aplicación, se asegura de que el usuario realizando la solicitud sea el propietario de la aplicación; de no serlo, retorna una respuesta con código 403 indicando el permiso denegado para realizar la acción. Tras verificar la identidad del usuario, el método parsea el reporte de escaneo en formato JSON proporcionado y revisa si ya existe un reporte similar, comprobando que no haya un escaneo de la misma herramienta y el mismo *DateTime*. Si ya existe, se levanta una excepción con el código de estado 400.

Luego, el método itera sobre las vulnerabilidades reportadas en el escaneo, revisando que estas no hayan sido importadas previamente en los datos históricos de la aplicación, verificando el nombre de la vulnerabilidad y la *URI* afectada. Si la vulnerabilidad ya existe, pero fue mitigada, se actualiza su estado a reactivada y se enlaza al nuevo escaneo importado. Si la vulnerabilidad ya existe y sigue activa, se pasa a la siguiente vulnerabilidad en la iteración. Al finalizar la iteración, si no se agregó o actualizó ninguna vulnerabilidad, se levanta una excepción con código de estado 400. Si al menos una vulnerabilidad se actualizó o creó, entonces se efectúa la transacción en la base de datos y se retorna una respuesta con código 201, indicando el éxito del importe.

Todo este proceso se realiza utilizando una transacción para evitar que los datos queden corrompidos en la base de datos, asegurando que todas las vulnerabilidades del escaneo se importen correctamente o que ninguna se importe en caso de error.

Autenticación

Como se mencionó en la sección 3.5.3, la autenticación se lleva a cabo utilizando JWT Tokens. Para ello, se empleó la librería "Simple JWT" para Django REST Framework. Esta librería proporciona todas las herramientas necesarias para implementar el uso de *tokens* JWT en la API. Solamente requiere establecer el uso de la librería en la configuración del proyecto, definir la duración de los *tokens* y establecer los *endpoints* para la obtención y actualización de estos. Se crearon los *endpoints* `/api/token/` y `/api/token/refresh`, los cuales permiten a los usuarios obtener un nuevo *token* de acceso y refrescar un *token* existente, respectivamente. Esto asegura que el proceso de autenticación sea seguro y eficiente, permitiendo a los usuarios mantener sesiones activas sin necesidad de volver a autenticarse constantemente.

Para la obtención del *token*, se requiere hacer una POST al *endpoint* correspondiente, entregando el nombre de usuario y la contraseña. Debido a la sensibilidad de esta información, es fundamental utilizar una conexión segura mediante HTTPS para proteger las credenciales del usuario en la comunicación. Del mismo modo, para actualizar un *token* de acceso, se debe hacer una solicitud POST al endpoint correspondiente, pero esta vez entregando únicamente el *token* de actualización.

4.1.3. Front-End

Según lo explicado en la sección 3.7 el *front-end* se creó utilizando únicamente plantillas HTML provistas por Django, en conjunto con librerías de CSS y JavaScript. Por ello, el desarrollo consistió en la creación de las plantillas HTML, la configuración de las librerías utilizadas para lograr una integración fluida y el levantamiento del servidor *proxy*.

Plantillas HTML

Se decidió crear componentes reutilizables para todas las vistas, a modo de agilizar el desarrollo y mantener una consistencia visual. Entre los componentes creados se encuentran una barra de navegación, implementada utilizando Bootstrap, que permite a los usuarios moverse fácilmente entre distintas secciones de la aplicación, y una tabla para mostrar datos, generada utilizando la librería de JavaScript DataTables, que facilita la visualización, filtrado y gestión de los datos. También se desarrolló una plantilla base para la aplicación, que sirve como esqueleto para las demás plantillas creadas, asegurando que los elementos en común, como los scripts y hojas de estilo importadas, se carguen de manera eficiente y uniforme en todas las vistas.

La barra de navegación consiste en un bloque de código proporcionado por Bootstrap, listo para usar, solamente requiere que la librería sea importada. Esta barra de navegación fue personalizada acorde a las necesidades de la aplicación. El menú cuenta con cuatro secciones principales: la página de inicio, un menú desplegable de aplicaciones (que incluye enlaces a la vista de aplicaciones y a la vista para agregar nuevas aplicaciones), y enlaces para iniciar sesión, cerrar sesión y registrar un usuario. Además, se implementó una verificación para adaptar el contenido del menú según el estado de autenticación del usuario. Si el usuario no

está autenticado, solo se muestran las opciones de registro e inicio de sesión. En caso de estar autenticado, se muestran los enlaces a la página de inicio, el menú de aplicaciones y la opción de cierre de sesión.

Para la plantilla de tablas, que se utiliza en las vistas de aplicaciones, escaneos y vulnerabilidades, y que extiende de la plantilla base, se creó un esqueleto de tabla utilizando las etiquetas `<table>`, `<thead>` y `<tbody>`. La tabla se genera con el tag `<table>` al que se le agregan los atributos `id` y `class`, asignándoles los valores de las variables "table_id" y "table_class", respectivamente, para así generar la referencia y estilo de la tabla de manera dinámica. En la cabecera de la tabla, se creó un bloque "table_header" en el cual se deben especificar los atributos o columnas de la tabla. Del mismo modo, en el cuerpo de la tabla, se creó el bloque "table_body" para los datos. Además, se extendió el bloque de scripts de la plantilla base para incluir un nuevo script según lo especificado por la biblioteca DataTables. En este script se genera una configuración por defecto para la tabla, entregando la referencia al objeto `<table>` del DOM mediante el valor de su atributo *ID*.

La plantilla base comienza con la declaración del tipo de documento, seguido de las configuraciones esenciales dentro de la cabecera del documento, como la codificación del documento, los *meta tags*, *scripts*, librerías y hojas de estilo. Se utilizaron bloques ("blocks") de los Templates de Django para generar dinámicamente los títulos, contenido y para poder agregar *scripts* y/u hojas de estilos adicionales a lo largo de las plantillas. Los bloques permiten la personalización y extensión de la plantilla base sin tener que duplicar código.

El cuerpo del documento comienza con la inclusión de la barra de navegación, la cual se habilita o deshabilita mediante una variable de contexto llamada "disable_nav", proporcionada al renderizar las plantillas en las vistas de Django. Luego, se coloca un contenedor mediante una etiqueta `<div>` para los mensajes de alerta que se muestran dinámicamente mediante bloques. Estos mensajes retroalimentan al usuario sobre a las acciones realizadas en la aplicación, o sobre como proceder en ciertos casos.

Posterior a ello, se define un bloque "content" el cual se utiliza para insertar el contenido específico de cada vista en la estructura general de la plantilla base. Esto permite que el desarrollo de las vistas se concentre únicamente en presentar los datos específicos que cada una tiene como objetivo mostrar, mientras se mantiene una apariencia y comportamiento consistente.

Al final de la plantilla, se realizan las importaciones de las librerías y scripts utilizados que impactan en el renderizado de la vista. Esto se hace por razones de optimización y por funcionamiento de los DOM (Document Object Model). Si una librería o *script* realiza acciones sobre el documento, o genera contenido en este, puede haber errores de referencia si la importación se hace antes de que el contenido HTML haya sido completamente cargado. Colocarlos en esta altura asegura que el documento esté completamente disponible al momento de que el navegador ejecute las etiquetas `<script>`. También mejora la experiencia del usuario, ya que el contenido visual se renderiza con mayor rapidez y no se bloquea por la carga de las librerías.

Con esto, las demás plantillas únicamente deben extender la plantilla base y modificar los bloques que requieran para presentar sus vistas, agilizando el desarrollo y manteniendo

una experiencia uniforme para el usuario a lo largo de la aplicación.

4.1.4. Interfaces de la aplicación

El diseño de las interfaces se realizó siguiendo lo propuesto en los *mockups* expuestos en el anexo A.2.

Página de inicio

Al ingresar a la plataforma web, el usuario se encuentra con una vista de bienvenida que proporciona una breve descripción del objetivo de la aplicación. Si el usuario no está autenticado, la página muestra los botones para iniciar sesión o registrarse en la plataforma. En caso de que el usuario ya esté autenticado, en lugar de los botones de inicio de sesión o registro, se presenta un botón que redirecciona al usuario a la interfaz de sus aplicaciones.

Aplicaciones del usuario

Una vez el usuario se ha autenticado con su cuenta, es redireccionado a la vista de sus aplicaciones. Esta vista tiene como objetivo mostrar al usuario todas las aplicaciones que ha agregado a la plataforma para realizar su seguimiento. Cada aplicación se presenta en una tabla que incluye detalles como *ID*, nombre, *URL*, fecha de creación, entre otros. Además, se proporcionan botones de acción junto a cada aplicación que permiten al usuario acceder al panel de control para ver más detalles, actualizar los datos de la aplicación, o eliminarla de la plataforma.

Nueva aplicación

Si el usuario aún no ha agregado aplicaciones a la plataforma, puede dirigirse al menú desplegable de "Aplicaciones" ubicado en la barra de navegación y hacer clic en "Trackear Nueva Aplicación" para ser redireccionado a la página de nuevas aplicaciones. En esta página se presenta un formulario que debe ser completado con los detalles de la aplicación a la que se quiere hacer seguimiento. Una vez enviado el formulario, el usuario es redireccionado a la página de aplicaciones, donde podrá visualizar un mensaje que indica el éxito de la operación junto a la nueva aplicación en la tabla.

Panel de control

Si el usuario selecciona el botón "Ver Detalles" en la tabla de aplicaciones, este será redireccionado al panel de control de la aplicación, donde podrá visualizar diferentes métricas y gráficos sobre el estado de seguridad de la aplicación y sus vulnerabilidades. En esta página, los gráficos son interactivos, permitiendo al usuario filtrar datos y obtener información más

detallada acerca de estos. En la esquina superior derecha, se presentan tres botones: uno para ir a la página de escaneos, otro para ir a la página de vulnerabilidades y un tercero para ir a la página de importación de escaneos en la aplicación sin el uso de la API.

Escaneos de la aplicación

La página de escaneos presenta al usuario un listado de los escaneos que han sido realizados e importados a la plataforma, mostrándole los detalles de estos, como con qué herramienta se realizó y en que fecha. Además, se proporciona la opción de eliminar el escaneo mediante un botón. En la esquina superior derecha, se encuentra un botón para volver al panel de control de la aplicación.

Vulnerabilidades de la aplicación

La página de vulnerabilidades muestra al usuario todas las vulnerabilidades que han sido reportadas en los escaneos de la aplicación. Se presentan en una tabla, mostrando detalles como el nombre de la vulnerabilidad, el escaneo asociado (en el cual se reportó), el estado actual, en que fecha se reportó, su severidad, las recomendaciones para su solución y/o referencias, la *URI* afectada y un botón de acción para eliminar la vulnerabilidad de la base de datos. Debajo del estado actual de la vulnerabilidad, hay un enlace que permite al usuario marcar la vulnerabilidad como mitigada al ser cliqueado, actualizando su estado. En la esquina superior derecha se encuentra un botón para volver al panel de control de la aplicación.

Eliminación de datos

Como se mencionó anteriormente, todos los datos, ya sea aplicación, escaneo o vulnerabilidad, poseen la opción de ser eliminados de la base de datos mediante un botón. Si el usuario hace clic sobre este botón, automáticamente aparecerá un *modal* advirtiendo al usuario de la acción que está por realizar y pidiendo su confirmación para llevarla a cabo. Si el usuario confirma la acción, los datos son eliminados de forma permanente de la base de datos.

4.1.5. Servidor Proxy

Según lo explicado en la sección 3.7.2, se levanta un servidor *proxy* utilizando NGINX para recibir las solicitudes y servir el contenido estático de la aplicación. Para ello, se crea un archivo de configuración con extensión `.conf`. Esta configuración consiste en la apertura de los puertos 80 y 443 para la comunicación HTTP/HTTPS entre el servidor y el cliente, la configuración del *proxy* para redirigir las solicitudes al *back-end* de la aplicación, el establecimiento de los certificados SSL para la comunicación segura y la configuración para servir los archivos estáticos. El anexo A.5 presenta la configuración utilizada para crear el servidor *proxy*.

Los certificados SSL necesarios para la configuración del servidor deben ser proporcionados por el usuario. Estos certificados pueden ser de dos tipos: "self-signed.º certificados autorizados por una entidad certificadora (CA) a través de un dominio registrado.

4.2. Despliegue de la aplicación

Para desplegar la aplicación se utilizó Docker junto a la herramienta Docker Compose a modo de *contenerizar* el proyecto. Se creó un archivo Dockerfile para cada componente de la aplicación mediante el uso de imágenes provenientes de Docker Hub, un repositorio público de imágenes Docker. En cada Dockerfile se incluyeron las instrucciones necesarias para la instalación y configuración de los componentes, ensamblando así la imagen final para cada uno, como se detalla en el anexo A.6. Estas imágenes se emplearon posteriormente en el despliegue de los servicios de la aplicación.

Una vez creadas las imágenes de los componentes de la aplicación, se desarrolló un archivo "`docker-compose.yml`" para orquestar el levantamiento de los contenedores Docker y configurar sus parámetros mediante el uso de Docker Compose. En este archivo se declaran los servicios a levantar, las imágenes utilizadas por cada uno, los volúmenes persistentes de datos, las redes de comunicación entre los servicios y otras configuraciones necesarias. El detalle de esta configuración se puede revisar en el anexo A.6, código A.11. Aunque no se profundizará en los parámetros específicos, es importante destacar que solo el contenedor *front-end* se expone a internet a través de la red "vuln_track_external" y la exposición de los puertos `443` y `80` para la conexión de los usuarios mediante sus navegadores. El contenedor de *back-end* se encuentra en ambas redes, actuando como nexo entre los servicios. Además, se comparte un volumen persistente de datos entre *back-end* y *front-end* para almacenar y servir los archivos estáticos de la aplicación. El servicio de la base de datos cuenta con su propio volumen persistente para almacenar los datos de la aplicación de manera segura.

Esta configuración de despliegue permite también al usuario personalizarla según sus necesidades o situación particular. Por ejemplo, se puede cambiar el uso de imágenes de Docker Hub por imágenes propias alojadas en algún repositorio de imágenes privado como Rancher. Del mismo modo, se puede optar por desplegar la aplicación dentro de una red privada, haciéndola accesible solo para usuarios dentro de esa red y evitando su exposición a Internet. Esta flexibilidad facilita la adaptación de la infraestructura a diferentes entornos y requisitos de seguridad.

Con los archivos preparados, el siguiente paso es utilizar los comandos de Docker Compose para generar las imágenes y crear los contenedores que ejecutaran los servicios de la aplicación. Docker Compose automatiza este proceso, ensamblando los componentes y levantando la aplicación. El código 4.5 muestra el comando a utilizar para iniciar este proceso.

```
1    docker compose up --build
```

Listing 4.5: Comando para levantar la aplicación con Docker Compose

Este comando construirá las imágenes y levantará los contenedores según la configuración especificada en el archivo `docker-compose.yml`. En caso de que ocurran errores en el des-

pliegue o que se desee actualizar la configuración, Docker Compose lo facilita, permitiendo reiniciar los servicios o reconstruyendo las imágenes según sea necesario.

Una vez el comando se haya ejecutado sin problemas, se podrá acceder a la aplicación mediante un navegador web en la dirección `https://127.0.0.1/`.

4.3. Seguridad

En aspectos de seguridad, la aplicación se desarrolló utilizando diversas opciones ofrecidas por Django en su documentación para garantizar una protección robusta contra amenazas comunes. Algunas de las medidas implementadas incluyen:

- Secure Proxy SSL Header: Asegura que Django reconozca cuando el cliente está accediendo a través de HTTPS, previniendo su exposición en conexiones no seguras, chequeando un *header* específico incorporado por el servidor *proxy* al redirigir la solicitud. Esto debido a que es el servidor web del *front-end* quien recibe las solicitudes y realiza las consultas al *back-end* mediante HTTP, de este modo el servidor *back-end* sabe que en realidad es una conexión segura chequeando el *header* que esta configuración genera.
- Session Cookie Secure: Garantiza que las *cookies* de sesión solo se envíen a través de HTTPS, previniendo la exposición de estas en conexiones no seguras.
- CSRF Cookie Secure: Similar al caso anterior, establece que la *cookie* CSRF solo se transmita mediante HTTPS, protegiendo la plataforma frente ataques de falsificación de solicitudes entre sitios, o CSRF por sus siglas en inglés.
- Secure SSL Redirect: Para redirigir de manera automática todas las solicitudes HTTP a HTTPS, asegurando que todas las comunicaciones entre el usuario y el servidor se realicen de forma segura.
- HTTP Strict Transport Security (HSTS): Esta configuración permite indicar a los navegadores que solo deben interactuar con la plataforma a través de HTTPS, protegiendo frente a ataques de degradación de protocolo y secuestro de cookies. Sin embargo, al utilizar un certificado SSL con algún error como ser firmado por una autoridad no reconocida, ser autofirmado o caducado, entre otros, los navegadores respetarán HSTS y no permitirán al usuario realizar la conexión. Por ello, la configuración por defecto es desactivar esta opción, dándole la opción al usuario de activarla mediante variables de entorno.

Otra medida de seguridad implementada es el uso de la biblioteca "django-admin-honeypot" para generar un *honeypot* en la *URL* de administración de Django. Dado que todos los proyectos de Django poseen un *endpoint* para la administración de su aplicación, y por defecto se encuentra en la ruta `/admin` del servidor, este es un objetivo común para intentos de acceso no autorizados. Para mitigar este riesgo, se cambia la ruta por defecto a una secreta, designada por el usuario mediante una variable de entorno "ADMIN_PATH" en el archivo

`.env`, y se implementa un *honeypot*, que consiste en una vista de administración falsa. Esta vista detecta intentos de inicio de sesión y notifica a los administradores designados de la plataforma mediante un correo electrónico o mediante un *listener* personalizado. Solo se requiere que el usuario indique los administradores de la plataforma y configure su servidor de correo electrónico en las configuraciones del proyecto. En el anexo A.8 se puede observar el detalle de la configuración.

Por último, se estableció el uso de Content Security Policy (o CSP, por sus siglas) la cual "es una capa de seguridad adicional que ayuda a prevenir y mitigar algunos tipos de ataque, incluyendo Cross Site Scripting (XSS) y ataques de inyección de datos."(Mozilla Contributors, 2024)[17]. La implementación de CSP requiere que el servidor devuelva la cabecera "Content-Security-Policy" en las respuestas a las solicitudes del cliente, restringiendo las fuentes de los *scripts*, estilos e imágenes permitidas. Para configurar esta política de seguridad de contenido, se utilizó la librería "django-csp", que facilita la adición de la cabecera al permitir especificar las fuentes en la configuración del proyecto. En el anexo A.8 se detalla la configuración utilizada.

Capítulo 5

Evaluación de la solución

Este capítulo presenta las acciones realizadas para validar el correcto funcionamiento de la aplicación. Se detalla el uso de herramientas para la automatización de pruebas unitarias y la detección de secretos, asegurando la calidad del código y evitando la filtración de datos sensibles. Además, se evalúa el diseño de la solución, analizando la arquitectura y los beneficios asociados, y se discute la prueba de usuarios realizada.

5.1. GitHub Workflows

Para garantizar el correcto funcionamiento del desarrollo se utilizó una herramienta de GitHub llamada "Workflow". Estos corresponden a procesos automatizados configurables que ejecutan uno o más trabajos y cada trabajo tiene una serie de uno o más pasos encargados de ejecutar un *script* (GitHub, Inc., 2021).[10] Estos flujos se configuran mediante archivos de tipo YAML en los cuales se declara el nombre del flujo, sobre qué ramas de desarrollo ejecutarse y en qué ocasiones (si al subir cambios a una rama o combinar ramas), que servicios levanta y los trabajos junto a sus pasos.

5.1.1. Integración continua

Para asegurar que cada vez que la aplicación sufriera cambios en su código fuente, todo funcionara correctamente y no se rompieran funcionalidades, se diseñaron pruebas unitarias utilizando los módulos `Test` proporcionados por Django y Django REST Framework. Estas pruebas verifican el funcionamiento de las funcionalidades creadas para la plataforma y su ejecución se automatizó de la ayuda de un flujo de trabajo de GitHub.

En este flujo, se levanta la plataforma utilizando el comando `runserver` de Django junto a un servicio de base de datos MySQL para ejecutar las pruebas unitarias diseñadas para la aplicación cuando se realizan cambios sobre las ramas `"main"`, `"dev"` y `"test"`, o cuando se combinan cambios en las ramas `"main"` y `"dev"`. Esto permite probar y verificar las diferentes funcionalidades que la aplicación ofrece. El archivo YAML con la configuración se

puede revisar en el anexo A.9, código A.17.

5.1.2. Detección de secretos

También se utilizó la herramienta "Gitleaks" [1] para la detección de secretos, contraseñas o información sensible incrustada en el código fuente de la aplicación dentro de repositorios Git. La implementación de esta herramienta se automatizó mediante la creación de un nuevo flujo de trabajo de GitHub. En este flujo, se configuró la ejecución de la herramienta al realizar y combinar cambios sobre las ramas principales de desarrollo, `"main"` y `"dev"`. El detalle de la configuración se encuentra en el anexo A.9, código A.18.

5.2. Evaluación del diseño

Para evaluar el diseño de la plataforma se analizó la arquitectura monolítica adoptada, destacando sus beneficios para el mantenimiento y escalabilidad del proyecto. Además, se realizó una prueba de usuarios llevada a cabo con profesionales del área de ciberseguridad, la cual permitió recopilar retroalimentación valiosa para el mejoramiento del sistema y la validación de este.

5.2.1. Arquitectura monolito

La arquitectura monolítica del proyecto y la modularización de los componentes del sistema son beneficiosos para la mantención y el desarrollo futuro del código. Al concentrar el código fuente en un único repositorio y estructura, se simplifica la gestión y seguimiento de los cambios, lo que facilita la detección y solución de errores.

La modularización de los componentes adoptada durante el desarrollo permite que los distintos componentes de la aplicación, ya sea *back-end*, *front-end* o la base de datos, se gestionen de manera independiente pero cohesiva a la vez. Cada módulo puede actualizarse o modificarse sin dañar el funcionamiento del sistema. Esto garantiza que las mejoras o correcciones de un componente no causen problemas en otros, lo que aumenta la estabilidad del sistema.

Sin embargo, es importante tener en cuenta las limitaciones que conlleva esta elección. En particular, la adopción de nuevas tecnologías se vuelve costosa tanto en tiempo como en horas-hombre, y la velocidad y complejidad del desarrollo se vuelven inversamente proporcionales a medida que la aplicación crece.

Por último, aunque la arquitectura monolítica tiene ciertas limitaciones, la simplicidad, la gestión y la estabilidad hacen que sea una opción viable y efectiva para el desarrollo de esta plataforma.

5.2.2. Prueba de usuarios

Se llevó a cabo una prueba de usuario con un equipo de desarrollo compuesto por seis personas especializadas en ciberseguridad, específicamente en roles de "DevSecOps", para validar la funcionalidad y la utilidad de la plataforma desarrollada.

El proyecto se presentó en una reunión al inicio del proceso de prueba. Durante esta reunión, se dio una explicación detallada de los objetivos del trabajo, se habló sobre la arquitectura del sistema, su despliegue y las características principales de la plataforma. Se demostró cómo la plataforma permite el seguimiento de vulnerabilidades y cómo el tablero visualiza los datos.

Después de la introducción a la plataforma, el equipo pudo interactuar con el sistema. Para poner a prueba las funcionalidades del sistema, cargaron reportes de sus desarrollos y analizaron cómo la plataforma mostraba los datos acerca de las vulnerabilidades. Además, evaluaron como el tablero de mando generaba las visualizaciones y métricas a partir de los datos, verificando la utilidad de la información que proporcionada.

Esta prueba de usuario generó una valiosa retroalimentación acerca del desarrollo de la plataforma. Se sugirieron mejoras estéticas para hacer que la interfaz fuese más atractiva y fácil de usar. Se mencionaron posibles cambios en las métricas mostradas en el tablero de control, así como la adición de nuevas métricas para que la visualización del estado de seguridad fuese más precisa y agregase mayor valor.

El equipo mostró además un gran interés por el proyecto, enfatizando en su potencial y utilidad en el campo de la ciberseguridad. Este *feedback* no soló validó la funcionalidad de la plataforma, sino que también ofreció oportunidades de mejora significativas. La retroalimentación fue esencial para mejorar el sistema y asegurar que cumple con las expectativas de los posibles usuarios finales.

En conclusión, la prueba de usuario en un ambiente real fue un éxito; brindando tanto validación de la funcionalidad de la plataforma como sugerencias útiles para su mejora continua.

Capítulo 6

Conclusión

En el siguiente capítulo se presentan las conclusiones del trabajo llevado a cabo, discutiendo si se cumplieron los objetivos planteados. Además, se proponen futuras líneas de trabajo para seguir el desarrollo de la plataforma.

6.1. Discusión final

Recapitulando, el objetivo principal de esta memoria fue crear un sistema para el seguimiento y análisis de vulnerabilidades en aplicaciones web, utilizando un tablero de control que considera el apetito de riesgo de la aplicación para calcular un nivel de seguridad. El sistema debería poder recibir reportes generados por herramientas de escaneo de vulnerabilidades y ofrecer una representación visual y cuantitativa de cómo ha cambiado la seguridad de las aplicaciones a lo largo del tiempo. Para lograrlo, se establecieron diez objetivos específicos, de los cuales se discutirá su cumplimiento.

De acuerdo con lo establecido en el objetivo específico número uno, se investigaron y estudiaron algunas de las herramientas más populares de escaneo de vulnerabilidades para aplicaciones web, con lo que se determinó el uso de dos de ellas: OWASP ZAP y Wapiti. Estas herramientas proporcionan la información acerca del estado de seguridad de las aplicaciones y son la fuente de información para la plataforma.

Se crearon las métricas y gráficos que se usarían como parte del diseño del tablero de mando de la aplicación para que el usuario pueda visualizar la cantidad de vulnerabilidades por tipo de severidad, el estado general de mitigación, el tiempo que tomaría llegar a cero vulnerabilidades activas dada una tasa de mitigaciones por día, la nota de seguridad que se le asignaría a la aplicación dadas las vulnerabilidades activas, etc. Estas visualizaciones muestran cómo el estado de seguridad de las aplicaciones cambia en el tiempo, y, por lo tanto, cumplen con el objetivo número dos.

Para cumplir con los objetivos tres y cuatro, se desarrolló un algoritmo *parser* encargado de analizar y extraer la información producida por los reportes de seguridad generados por las herramientas de escaneo de vulnerabilidades elegidas. Este algoritmo estandariza la infor-

mación, facilitando su acceso y procesamiento por parte de la plataforma. Adicionalmente, se diseñó una base de datos para la plataforma, gestionada con *MySQL*, que permite guardar y gestionar las vulnerabilidades detectadas en los escaneos de las aplicaciones, asegurando su disponibilidad para futuros análisis y comparaciones.

Se crearon las interfaces de la plataforma web, incluyendo la lógica para la interacción del usuario, la creación del estilo visual de la plataforma y la integración de los *scripts* correspondientes. Paralelamente, se desarrolló la *API* a través de la cual los usuarios pueden importar los reportes generados por las herramientas de escaneo permitidas mediante solicitudes HTTP, cumpliendo así con los objetivos cinco y seis.

Una vez desarrollados los componentes de la aplicación, se realizó la integración y despliegue de estos mediante la utilización de Docker, alcanzando el objetivo siete. Con la plataforma desplegada, se procedió a realizar pruebas utilizando escaneos de aplicaciones web vulnerables para verificar el correcto funcionamiento de los componentes creados. Cuando se alcanzó el nivel de madurez deseado para la plataforma, fue evaluada por un equipo de desarrollo del área de ciberseguridad, que consistía en seis personas con estudios afines al área de la computación, de una empresa del rubro *retail*, los cuales proporcionaron retroalimentación acerca del trabajo realizado. Esto ayudó a cumplir los objetivos ocho y diez respectivamente.

En paralelo con el desarrollo de los componentes, se generó la documentación necesaria sobre la arquitectura del sistema, sus componentes, las funcionalidades desarrolladas y cómo utilizarlo, manteniendo actualizada tanto la información técnica como la práctica. Esto permite la mantención, mejora y desarrollo continuo del trabajo realizado, cumpliendo así con el objetivo nueve.

En resumen, los objetivos específicos del proyecto se cumplieron a cabalidad y, gracias a la retroalimentación recibida, se logró validar el funcionamiento y el valor de la solución creada. Por ello, se alcanza el objetivo general del proyecto.

6.2. Trabajo Futuro

Con esta primera versión del proyecto, se logra cumplir con los objetivos propuestos para esta memoria de título. Sin embargo, esto corresponde únicamente al comienzo del desarrollo de una nueva solución para llevar a cabo el seguimiento de vulnerabilidades. Este avance abre las puertas a diversas mejoras y nuevas funcionalidades gracias a la escalabilidad del proyecto.

Entre las posibles mejoras al trabajo desarrollado, se propone la ampliación del sistema de usuarios de la plataforma para permitir la creación de equipos u organizaciones de las cuales múltiples usuarios puedan ser parte. De este modo, una aplicación podría ser administrada por múltiples usuarios mediante la creación de una organización, facilitando el uso de la plataforma en contextos empresariales o de equipos de desarrollo. También, se sugiere ampliar el uso de la plataforma para el seguimiento de vulnerabilidades en cualquier tipo de desarrollo de software, no limitándolo únicamente a aplicaciones web.

Por otro lado, existe la idea de la creación de un *"metaparser"* capaz de recibir reportes de cualquier herramienta de escaneo de vulnerabilidades sin la necesidad de tener que ser habilitada previamente en la plataforma mediante la creación de un *parser* específico. Esto podría realizarse mediante el desarrollo de un modelo de *machine learning* capaz de recibir un reporte proveniente de cualquier herramienta y estandarizar la información que este proporciona para ser utilizada en la plataforma. Este enfoque abriría nuevas oportunidades para investigaciones y memorias de título en el campo de la inteligencia artificial.

Bibliografía

[1] Gitleaks LLC. Gitleaks. https://gitleaks.io/, 2022. Último acceso: 11 de julio de 2024.

[2] Amazon Web Services. ¿Qué es la ciberseguridad? https://aws.amazon.com/es/what-is/cybersecurity/, 2022. Último acceso: 19 de Septiembre de 2023.

[3] Brinqa. Brinqa Attack Surface Intelligence Platform. https://www.brinqa.com/, 2023. Último acceso: 14 de Julio de 2024.

[4] Cloudflare, Inc. Almacenamiento en la memoria caché de contenido estático y dinámico |cloudflare. https://www.cloudflare.com/es-es/learning/cdn/caching-static-and-dynamic-content/, 2021. Último acceso: 14 de Junio de 2024.

[5] Cloudflare, Inc. En qué consisten los servidores proxy inversos |Cloudflare. https://www.cloudflare.com/es-es/learning/cdn/glossary/reverse-proxy/, 2021. Último acceso: 14 de Junio de 2024.

[6] DefectDojo, Inc. Core Data Classes |Documentation. https://documentation.defectdojo.com/usage/models/, 2023. Último acceso: 24 de Noviembre de 2023.

[7] DefectDojo, Inc. DefectDojo |CI/CD and DevSecOps Automation. https://www.defectdojo.org/, 2023. Último acceso: 17 de Noviembre de 2023.

[8] DefectDojo, Inc. Documentation. https://documentation.defectdojo.com/, 2023. Último acceso: 24 de Noviembre de 2023.

[9] Forrester Research. The forrester wave: Vulnerability risk management, q3 2023. https://reprints2.forrester.com/#/assets/2/2531/RES178523/report, 2023. Último acceso: 27 de Noviembre de 2023.

[10] GitHub, Inc. Entender las github actions documentación de github. https://docs.github.com/es/actions/learn-github-actions/understanding-github-actions#workflows, 2021. Último acceso: 11 de julio de 2024.

[11] Greenbone AG. Greenbone community documentation: Background. https://greenbone.github.io/docs/latest/background.html#history-of-the-openvas-project, 2023. Último acceso: 26 de Noviembre de 2023.

[12] International Organization for Standardization. Iso 31073:2022(en) risk management - vocabulary. https://www.iso.org/obp/ui/#iso:std:iso:31073:ed-1:v1:en, 2010. Último acceso: 18 de Noviembre de 2023.

[13] M. Jones, J. Bradley, and N. Sakimura. Rfc 7519: Json web token (jwt). https://datatracker.ietf.org/doc/html/rfc7519, 2015. Último acceso: 18 de junio de 2024.

[14] Maria Hernandez. Banking industry sees 1318 % increase in ransomware attacks in 2021. https://www.securitymagazine.com/articles/96128-banking-industry-sees-1318-increase-in-ransomware-attacks-in-2021, 2021. Último acceso: 19 de Septiembre de 2023.

[15] Jacob Thornton Mark Otto. Bootstrap: The world's most popular framework for building responsive, mobile-first sites. https://getbootstrap.com/, 2012. Último acceso: 14 de Junio de 2024.

[16] Microsoft. ¿Qué es la ciberseguridad? https://support.microsoft.com/es-es/topic/-qu%C3%A9-es-la-ciberseguridad-8b6efd59-41ff-4743-87c8-0850a352a390, 2021. Último acceso: 18 de Noviembre de 2023.

[17] Mozilla Developer Network. Content security policy (csp). https://developer.mozilla.org/es/docs/Web/HTTP/CSP, 2024. Último acceso: 1 de julio de 2024.

[18] Nicolas Surribas. Wapiti : a free and open-source web-application vulnerability scanner in python. https://wapiti-scanner.github.io/, 2018. Último acceso: 11 de julio de 2024.

[19] Okta, Inc. Json web token introduction. https://jwt.io/introduction/, 2015.

[20] Stack Overflow. Stack overflow developer survey 2023. https://survey.stackoverflow.co/2023/, 2023. Último acceso: 14 de junio de 2024.

[21] OWASP Foundation. OWASP DefectDojo. https://owasp.org/www-project-defectdojo/, 2016. Último acceso: 19 de Septiembre de 2023.

[22] Oxford University Press. cybersecurity, n. In *OED Online*. Oxford University Press, Septiembre 2023.

[23] Tenable®, Inc. Nessus Vulnerability Scanner: Network Security Solution. https://www.tenable.com/products/nessus, 2014. Último acceso: 17 de Noviembre de 2023.

[24] Tenable®, Inc. Tenable Security Center. https://www.tenable.com/products/tenable-sc, 2014. Último acceso: 26 de Noviembre de 2023.

[25] Tenable®, Inc. Tenable Vulnerability Management. https://www.tenable.com/products/tenable-io, 2017. Último acceso: 26 de Noviembre de 2023.

[26] the ZAP Dev Team. Zed Attack Proxy. https://www.zaproxy.org/, 2011. Último acceso: 26 de Noviembre de 2023.

[27] TIOBE Software BV. Tiobe index for may 2024. https://www.tiobe.com/tiobe-index/, 2024.

[28] Uizard Technologies. Uizard: AI-powered Design Tool for Rapid Prototyping. https://uizard.io/, 2021. Último acceso: 14 de Junio de 2024.

[29] Verizon. Data breach investigations report 2023. https://www.verizon.com/business/resources/Tb70/reports/2023-data-breach-investigations-report-dbir.pdf, 2023. Último acceso: 19 de Septiembre de 2023.

Anexos

Anexo A

Detalles de la solución

En el presente anexo se detallan y ahonda en algunas de las secciones del presente documento, a modo de dejar en claro la implementación de la plataforma en su totalidad.

A.1. Gráficos

Esta sección muestra los gráficos explicados en la sección 3.4.1. En la figura A.1 se puede visualizar el gráfico de barras de severidades a lo largo del tiempo, en donde cada fecha representa un reporte de escaneo de seguridad importado, y las barras crecen según la cantidad de vulnerabilidades reportadas por severidad. La figura A.2 muestra el gráfico de torta, en el que se observan las proporciones de vulnerabilidades activas y mitigadas. La figura A.3 presenta la tendencia de incremento de la cantidad de vulnerabilidades por severidad a lo largo del tiempo, con las fechas de importación de los reportes de seguridad representando el eje temporal. Además, la figura A.4 ilustra las proporciones de cada tipo de severidad dentro del total de vulnerabilidades activas. Finalmente, la figura A.5 muestra el estado de mitigación de cada reporte mediante barras apiladas que indican la cantidad de vulnerabilidades mitigadas y activas para cada fecha de importación de reportes, mientras que la figura A.6 presenta las frecuencias de cada tipo de vulnerabilidad presente en la aplicación.

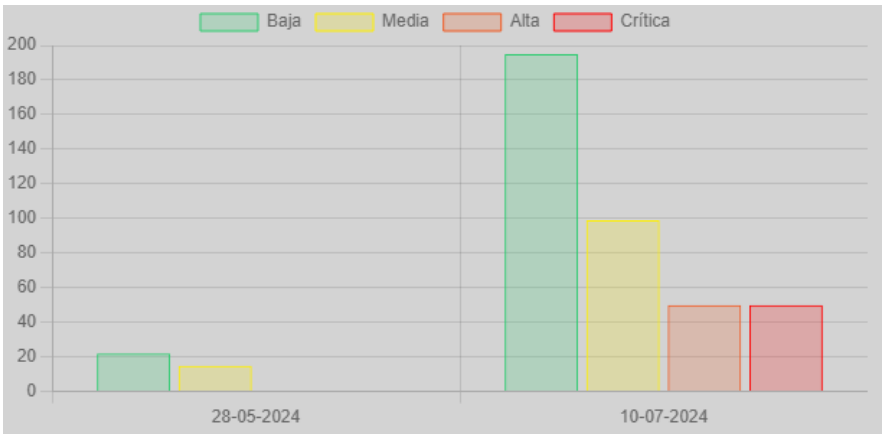

Figura A.1: Gráfico de barras, severidades en el tiempo

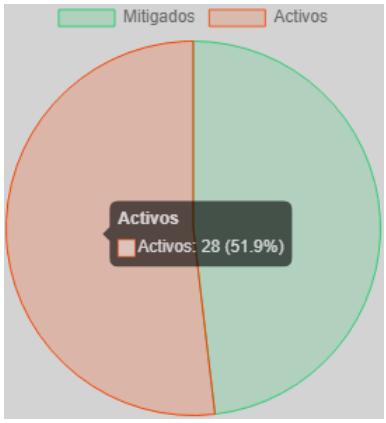

Figura A.2: Gráfico de torta, vulnerabilidades mitigadas versus activas

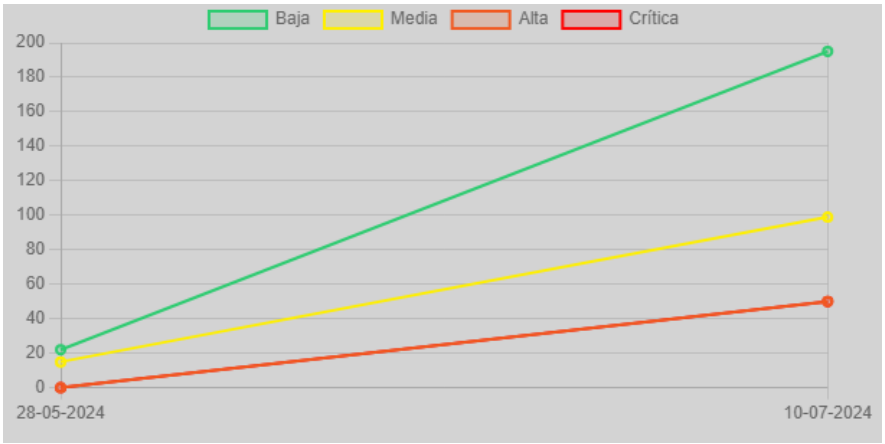

Figura A.3: Gráfico de tendencias, severidades en el tiempo

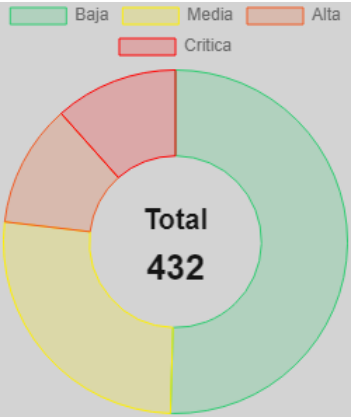

Figura A.4: Gráfico de torta, proporción de severidades

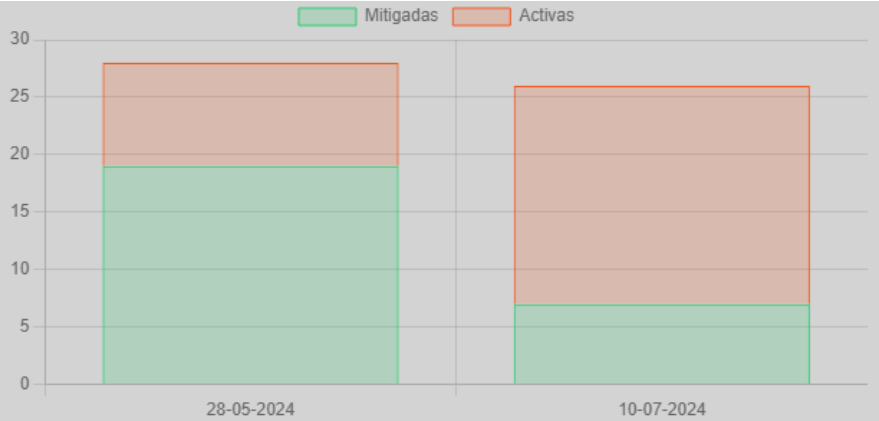

Figura A.5: Gráfico de barras apiladas, vulnerabilidades mitigadas versus activas

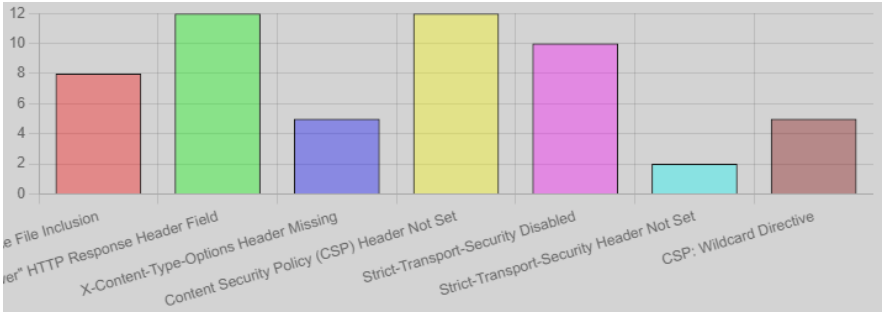

Figura A.6: Gráfico de barras, frecuencia de vulnerabilidades

A.2. Mockups

A continuación se presentan los *mockups* diseñados para la interfaz de la plataforma web. La imagen A.7 muestra la página de inicio de la aplicación, en donde se da una breve descripción sobre la plataforma y su objetivo, y da la opción al usuario de iniciar sesión o registrarse para comenzar a hacer uso de la plataforma. Luego, en las imágenes A.9 y A.8 se presentan las vistas de registro e inicio de sesión para los usuarios. Posterior a esto, en la imagen A.10 podemos ver la vista de aplicaciones, en esta cada usuario podrá ver las aplicaciones que ha ingresado a la plataforma junto a los detalles de esta y botones para poder editar los detalles, visualizar el panel de control y eliminar la aplicación del sistema. Esta vista también se utiliza para mostrar los escaneos asociados a la aplicación y las vulnerabilidades presentes en la aplicación. Luego, la imagen A.11 exhibe la vista para el panel de control, en donde se muestran los gráficos y métricas para cada aplicación junto a dos botones para acceder a los escaneos y vulnerabilidades asociadas a dichas aplicaciones.

Figura A.7: Página de inicio

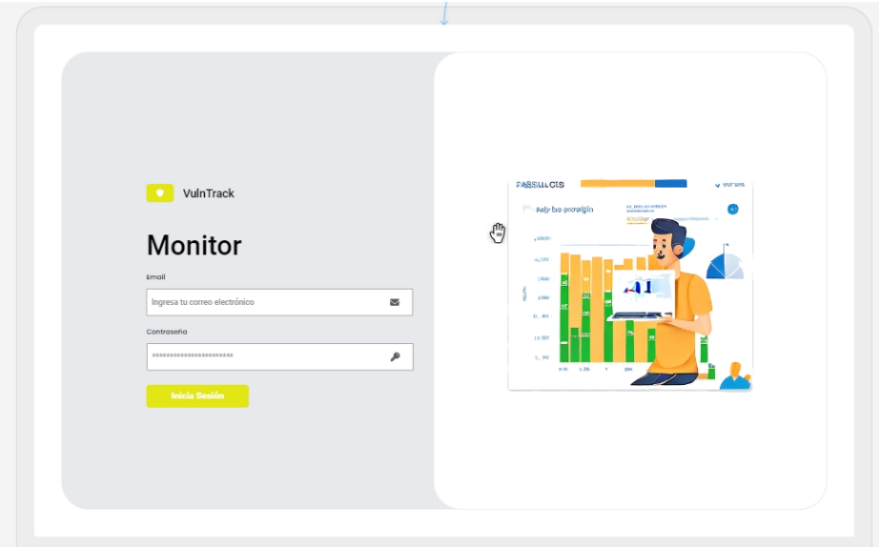

Figura A.8: Página de inicio de sesión

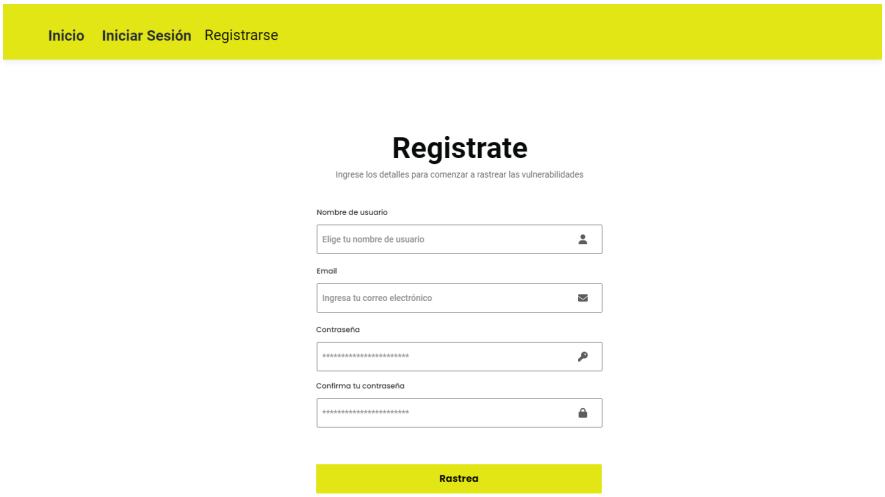

Figura A.9: Página de registro de usuarios

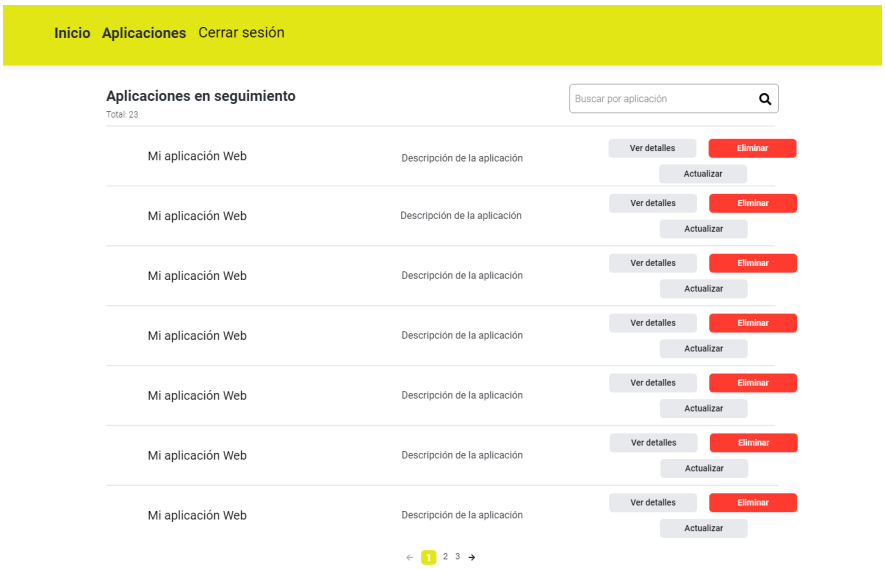

Figura A.10: Página de aplicaciones del usuario

Figura A.11: Página del dashboard de aplicaciones

A.3. Modelo de datos

A continuación se detalla los atributos de cada entidad del modelo de datos propuesto en la sección 3.6.

Usuarios (User)

La tabla de usuarios almacenará la información esencial sobre los individuos que utilizarán la aplicación. Esta tabla contará con los siguientes campos:

- user_id: La llave primaria de la tabla, identificando de manera única a cada usuario.
- username: Nombre de usuario que el individuo decidirá. Necesario para el inicio de sesión.
- email: La dirección de correo electrónico del usuario que permitirá establecer un canal de comunicación.
- password: La contraseña que el usuario utilizará para iniciar sesión y acceder a la aplicación.

Aplicaciones (Application)

La tabla de aplicaciones registrará información acerca de la aplicación que el usuario realizará seguimiento. Los campos que incluirá son:

- application_id: Llave primaría de la tabla.
- application_name: Nombre de la aplicación.
- application_description: Una breve descripción acerca de la aplicación.
- application_url: La URL donde está ubicada la aplicación web.
- application_risk_apetite: El apetito de riesgo establecido por el usuario para el cálculo de la nota de seguridad.
- application_created_at: Fecha de creación de la aplicación en la plataforma.
- application_updated_at: Fecha del último momento en que los datos de la aplicación fueron actualizados.
- owner_user: Llave foránea que relaciona la aplicación con el usuario propietario.

Escaneos (Scan)

La tabla de escaneos almacenará los metadatos de cada escaneo realizado. Los campos son:

- scanned_at: Fecha en la que el escaneo a la aplicación fue realizado.

- scantool_name: Nombre de la herramienta utilizada para realizar el escaneo.
- scanned_applitacion: Llave foránea para relacionar el escaneo con la aplicación en la que se realizó.

Vulnerabilidades (Vulnerability)

La tabla de vulnerabilidades contendrá datos detallados sobre cada vulnerabilidad reportada. Los campos incluirán:

- vulnerability_name: Nombre de la vulnerabilidad.
- vulnerability_desc: Descripción de la vulnerabilidad reportada.
- vulnerability_severity: Severidad de la vulnerabilidad.
- vulnerability_status: Estado de la vulnerabilidad (activa, mitigada o reactivada).
- vulnerability_note: Nota adicional con detalles sobre la vulnerabilidad, incluyendo cómo solucionarla o referencias adicionales.
- vulnerability_uri: URI afectada por la vulnerabilidad.
- vulnerability_mitigation_date: Fecha en que la vulnerabilidad fue mitigada por última vez.
- linked_scan: Llave foránea para relacionar la vulnerabilidad encontrada con el escaneo en donde fue reportada.

Esta estructura detallada permite una clara organización y gestión de los datos, facilitando el seguimiento y resolución de vulnerabilidades para los usuarios. Esta definición asegura una base de datos robusta y eficientes, capaz de manejar grandes volúmenes de información de manera efectiva, evitando problemas como la duplicación de información.

A.4. Back-End

Esta sección del anexo busca mostrar y precisar los desarrollos llevados a cabo en el *back-end* de la aplicación.

A.4.1. Vistas

A continuación, se detallan las funciones definidas para las vistas del *back-end* de la aplicación. El código A.1 muestra las declaraciones de las funciones de Django para cada una de las vistas de la aplicación basadas en las operaciones *CRUD*. Sin embargo, no se profundiza en la lógica que cada una de estas define debido a la extensión del documento. El código A.2 incluye las declaraciones para las vistas de inicio, registro y cierre de sesión en la aplicación *"Users"* de la plataforma.

```
# App_web APP
def inicio_view(request: HttpRequest) -> render:

def new_application_view(request: HttpRequest) -> render:

def applications_view(request: HttpRequest) -> render:

def update_application_view(request: HttpRequest, app_id: int) -> render:

def delete_application_view(request: HttpRequest, app_id: int) -> render:

def dashboard_view(request: HttpRequest, app_id: int) -> render:

def new_scan_view(request: HttpRequest, app_id: int) -> render:

def scans_view(request: HttpRequest, app_id: int) -> render:

def delete_scan_view(request: HttpRequest, app_id: int, scan_id: int) ->
    render:

def vulnerabilities_view(request: HttpRequest, app_id: int) -> render:

def update_vulnerability(request: HttpRequest, app_id: int, vuln_id: int)
    -> render:

def delete_vulnerability_view(request: HttpRequest, app_id: int, vuln_id:
    int) -> render:

def get_mitigation_rate_ajax(request: HttpRequest) -> JsonResponse:

def dashboard_view(request: HttpRequest, app_id: int) -> render:
```

Listing A.1: Funciones para las vistas de la aplicación *App_web*

```
# Users APP

def login_user(request: HttpRequest) -> render:
    """View for the login page.

    This view handles the login form and authenticates the user.

    It handles both GET and POST requests:
    - GET: Renders the login page.
    - POST: Authenticates the user and logs them in.
    """
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return redirect('app_web:applications')
        else:
            messages.error(request, 'Nombre de usuario o contrasena
    incorrectos. Por favor, intentelo de nuevo.')
            return redirect('users:login')
```

```
22      else:
23          return render(request, 'auth/login.html')
24
25  def logout_user(request: HttpRequest) -> redirect:
26      """View for the logout page.
27
28      This view logs out the user and redirects them to the home page.
29      """
30      logout(request)
31      messages.success(request, 'Has cerrado sesion correctamente.')
32      return redirect('app_web:home')
33
34  def register_user(request: HttpRequest) -> render:
35      """View for the registration page.
36
37      This view handles the registration form and creates a new user.
38
39      It handles both GET and POST requests:
40      - GET: Renders the registration page.
41      - POST: Creates a new user and redirects to the login page.
42      """
43      if request.user.is_authenticated:
44          return redirect('app_web:applications')
45      elif request.method == 'POST':
46          form = RegisterForm(request.POST)
47          if form.is_valid():
48              form.save()
49              messages.success(request, 'Tu cuenta ha sido creada con exito.
        Ahora puedes iniciar sesion.')
50              return redirect('users:login')
51      else:
52          form = RegisterForm()
53      return render(request, 'auth/register.html', {'form': form})
```

Listing A.2: Funciones para las vistas de la aplicación *Users*

A.4.2. Módulos

A continuación se presentan las implementaciones llevadas a cabo en los módulos `parsers.py`, `permisions.py`, `metrics.py` y `serializers.py`, según los objetivos que cada uno de ellos cumple, explicados en la sección 4.1.1 del documento. El código A.3 muestra las funciones utilizadas para el *parseo* de reportes de escaneo. El código A.4 ilustra la declaración de un decorador de Python, encargado de verificar que quien realiza la solicitud HTTP en las vistas sea el mismo usuario que figura como propietario de la aplicación. Por otro lado, los códigos A.5 y A.6, exhiben las funciones utilizadas para calcular las métricas mostradas en el *dashboard* de seguridad de las aplicaciones y las clases creadas para la serialización de objetos de las entidades *Scan* y *Vulnerability* de la base de datos, respectivamente.

```
1
2   def strip_tags(html: str) -> str:
3
4   def sanitize_data(data: str) -> str:
5
```

```
def sanitize_links(data: str) -> str:

def parse_zap(readed_data: dict) -> dict:
    def parse_alert(alert: dict) -> dict:
    def process_instance(alert_data: dict, instance: dict) -> dict:
    def initialize_report(readed_data: dict) -> dict:

def parse_wapiti(readed_data: dict) -> dict:
    def parse_alert(vuln: str, readed_data: dict) -> dict:
    def parse_vulnerability(vuln_data: dict, alert_data: dict) -> dict:

def parse_report(tool: str, json_file: IO[bytes]) -> dict:
```

Listing A.3: Funciones declaradas para el parseo de reportes

```
def user_is_application_owner(view_func: callable) -> callable:
    """Decorator to check if the current user is the owner of the application.

    Args:
        view_func (callable): The view function to be decorated.

    Returns:
        callable: The decorated view function.
    """
    def _wrapped_view(request: HttpRequest, *args, **kwargs) -> callable:
        """Function to check if the current user is the owner of the application.

        Args:
            request (HttpRequest): The request object from Django.

        Returns:
            callable: The view function if the user is the owner of the application, a 403 Forbidden response otherwise.
        """
        app_id = kwargs.get('app_id')

        if not app_id:
            app_id = request.GET.get('application_id', None) or request.POST.get('application_id', None)

        # If the 'app_id' is still not found, return a 403 Forbidden response
        if not app_id:
            return render(request, '403.html', status=403)

        application = get_object_or_404(Application, pk=app_id)

        if request.user != application.owner_user:
            return render(request, '403.html', status=403)

        # If the user is the owner of the application, call the view function
        return view_func(request, *args, **kwargs)
    return _wrapped_view
```

Listing A.4: Funciones para el chequeo de permisos en las vistas

```python
def get_application_score(low_count: int, medium_count: int, high_count: int, critical_count: int) -> float:

def get_application_calification(score: int, k: int, risk_apetite: Decimal) -> Decimal:

def get_days_in_period(start_date: date, end_date: date) -> int:

def get_mitigation_rate(vulnerabilities: QuerySet[Vulnerability], start_date: date, end_date: date) -> Decimal:

def get_time_to_zero_vulns(mitigation_rate: Decimal, vulnerabilities: QuerySet[Vulnerability], starting_date: date):

def get_severity_mode(vuln_severity_list: list) -> str:

def get_max_severity(vuln_severity_list: list) -> str:
```

Listing A.5: Funciones para el cálculo de metricas

```python
class VulnerabilitySerializer(serializers.ModelSerializer):
    """Serializer to convert the Vulnerability model to JSON and vice versa.

    Args:
        serializers (rest_framework.serializers.ModelSerializer): The ModelSerializer class from Django REST Framework.
    """
    class Meta:
        """Meta class to define the model and fields to serialize.
        """
        model = Vulnerability
        fields = '__all__'

class ScanSerializer(serializers.ModelSerializer):
    """Serializer to convert the Scan model to JSON and vice versa.

    Args:
        serializers (rest_framework.serializers.ModelSerializer): The ModelSerializer class from Django REST Framework.
    """
    class Meta:
        """Meta class to define the model and fields to serialize.
        """
        model = Scan
        fields = '__all__'
```

Listing A.6: Clases definidas para la serialización de objetos

A.5. Servidor proxy

Esta sección detalla el desarrollo del servidor *proxy* de la plataforma mediante la utilización de NGINX. En el código A.7 se puede visualizar la configuración utilizada para levantar

el servidor. En esta se puede observar la apertura de los puertos `80` y `443` para la comunicación HTTP/HTTPS con el cliente. La sección `location /static/` especifica la ruta de los archivos estáticos, que se encuentran en el volumen de datos persistentes `/app/staticfiles/`, y agrega la cabecera `X-Content-Type-Options` con el valor `nosniff` para mejorar la seguridad al evitar que los navegadores intenten adivinar el tipo de contenido.

La sección `location /` se encarga de redirigir todas las solicitudes al *back-end* de la aplicación, configurado para escuchar en el puerto `8000`, mediante la directiva `proxy_pass`. Además, se establecen varias cabeceras (`Host`, `X-Real-IP`, `X-Forwarded-For`, y `X-Forwarded-Proto`) para asegurar que el servidor *back-end* recibe la información necesaria sobre la solicitud original.

Para habilitar la comunicación segura HTTPS, se especifican las rutas de los certificados SSL, `ssl_certificate` y `ssl_certificate_key`, utilizados para cifrar las comunicaciones entre el cliente y el servidor. Finalmente, la directiva `server_tokens off` se utiliza para ocultar la versión de NGINX en las respuestas HTTP, mejorando así la seguridad al reducir la información expuesta sobre el servidor.

```
server {
    listen 80;
    listen 443 ssl;

    location /static/ {
        alias /app/staticfiles/;
        add_header X-Content-Type-Options "nosniff";
    }

    location / {
        proxy_pass http://backend:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    ssl_certificate /etc/nginx/certs/ssl.crt;
    ssl_certificate_key /etc/nginx/certs/ssl.key;
    server_tokens off;
}
```

Listing A.7: Configuración del servidor NGINX

A.6. Despliegue de la aplicación

Este anexo presenta los archivos Dockerfile y Docker Compose utilizados para configurar el despliegue y ensamblaje de la aplicación mediante la contenerización del proyecto. En primer lugar, el código A.8 muestra el archivo Dockerfile para crear la imagen del servicio del componente Back-End. Este utiliza una imagen de la distribución Alpine de Linux, instala un intérprete de Python3 junto a las dependencias del proyecto, y copia los archivos de la aplicación dentro del contenedor. En segundo lugar, el código A.9 contiene el archivo Docker-

file utilizado para levantar el servicio de la base de datos de la aplicación, que simplemente utiliza una imagen de MySQL disponible en Docker Hub. En tercer lugar, el código A.10 genera la imagen necesaria para el Front-End de la aplicación, utilizando una imagen de NGINX e importando los certificados SSL/TSL para la comunicación HTTPS. Por último, el código A.11 muestra el archivo `docker-compose.yaml` que define y configura los servicios de los componentes de la aplicación, permitiendo su integración y despliegue conjunto.

```
# Use an alpine image
FROM alpine:3.20

# Update the package list and upgrade installed packages
RUN apk update && apk upgrade

# Install Python 3, pip, and other necessary build dependencies
RUN apk add --no-cache python3 py3-pip python3-dev pkgconfig build-base
    libffi-dev openssl-dev

# Install MySQL development libraries
RUN apk add --no-cache mariadb-dev

# Set the environment variable
ENV PYTHONUNBUFFERED=1

# Set the working directory
WORKDIR /app

# Copy the parent directory contents into the container at /app
COPY . /app

# Create the log directory
RUN mkdir -p /app/logs

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir --break-system-packages -r requirements.txt
```

Listing A.8: Archivo Dockerfile para el Back-End de la aplicación

```
FROM mysql:8.0.37
```

Listing A.9: Archivo Dockerfile para la base de datos de la aplicación

```
FROM nginx:latest

# Copy the Nginx configuration file
COPY ./nginx/nginx.conf /etc/nginx/conf.d/default.conf

# Copy the certificate and key files
COPY docker/certs /etc/nginx/certs
```

Listing A.10: Archivo Dockerfile para el Front-End de la aplicación

```
services:
  db:
    build:
      context: .
      dockerfile: Dockerfile-mysql
    container_name: vuln_track_mysql
```

```
    restart: always
    volumes:
      - data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: ${DB_ROOT_PASSWORD}
      MYSQL_DATABASE: ${DB_NAME}
      MYSQL_USER: ${DB_USER}
      MYSQL_PASSWORD: ${DB_PASSWORD}
    networks:
      - vuln_track_internal
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost", "-p${
    DB_ROOT_PASSWORD}"]
      interval: 30s
      timeout: 10s
      retries: 10

  backend:
    build:
      context: .
      dockerfile: Dockerfile-backend
    container_name: vuln_track_backend
    command: sh -c "python3 vuln_mngmnt/manage.py migrate --noinput &&
    python3 vuln_mngmnt/manage.py collectstatic --noinput && python3
    vuln_mngmnt/manage.py runserver 0.0.0.0:8000 && gunicorn --workers 3 --
    bind 0.0.0.0:8000 --timeout 120 --access-logfile /app/logs/
    gunicorn_access.log --error-logfile /app/logs/gunicorn_error.log
    vuln_mngmnt.wsgi:application"
    restart: always
    volumes:
      - static_volume:/app/vuln_mngmnt/staticfiles
    env_file:
      - .env
    depends_on:
      db:
        condition: service_healthy
    networks:
      - vuln_track_internal
      - vuln_track_external

  nginx:
    build:
      context: .
      dockerfile: Dockerfile-nginx
    container_name: vuln_track_nginx
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - static_volume:/app/staticfiles
    depends_on:
      - backend
    networks:
      - vuln_track_external

networks:
```

```
  vuln_track_internal:
  vuln_track_external:

volumes:
  data:
  static_volume:
```

Listing A.11: Archivo docker-compose para la configuración de los servicios de la aplicación

A.7. API

En esta sección se puede observar la implementación de la clase "`CustomAPIException`" junto a las excepciones personalizadas creadas para la API de la aplicación. Estas excepciones permiten manejar errores específicos de manera más efectiva, proporcionando mensajes de error claros y códigos de estado HTTP adecuados.

```
class CustomAPIException(APIException):
    """Custom base class for API exceptions.

    This exception is used as a base class for custom exceptions in the API.
    It inherits from the APIException class from Django REST framework.

    Attributes:
        status_code (int): The status code of the exception.
        default_detail (str): The default detail message of the exception.
        default_code (str): The default code of the exception.
    """
    status_code = 400
    default_detail = 'Error en la API.'
    default_code = 'api_exception'

    def __init__(self, detail: str =None, code: int =None):
            """Initializes the CustomAPIException with the detail and code attributes.

            Args:
                detail (str, optional): The detail message of the exception. Defaults to the default_detail attribute.
                code (int, optional): The code of the exception. Defaults to the default_code attribute.
            """
            if detail is not None:
                self.detail = detail
            else:
                self.detail = self.default_detail

            if code is not None:
                self.code = code
            else:
                self.code = self.default_code

class NoFileProvided(CustomAPIException):
```

```
    default_detail = 'No se proporciono ningun archivo.'
    default_code = 'no_file_provided'

class ScanAlreadyExists(CustomAPIException):
    default_detail = 'Ya existe un escaneo con los mismos datos.'
    default_code = 'scan_already_exists'

class NoNewVulnerabilityFound(CustomAPIException):
    default_detail = 'No se encontraron vulnerabilidades nuevas en el reporte.'
    default_code = 'no_new_vulnerability_found'

class ErrorParsingFile(CustomAPIException):
    default_detail = 'Error al parsear el archivo.'
    default_code = 'error_parsing_file'

class ErrorSavingScan(CustomAPIException):
    default_detail = 'Error al guardar el escaneo.'
    default_code = 'error_saving_scan'

class ErrorSavingVulnerability(CustomAPIException):
    default_detail = 'Error al guardar la vulnerabilidad.'
    default_code = 'error_saving_vulnerability'

class ErrorSerializingScan(CustomAPIException):
    default_detail = 'Error al serializar el escaneo.'
    default_code = 'error_serializing_scan'
```

Listing A.12: Excepciones creadas para la API

A.7.1. Funciones auxiliares

Se presentan las definiciones en el código A.13 de las funciones auxiliares utilizadas por la API *RESTful* de la plataforma para recibir los reportes de seguridad, *parsearlos* y guardar la información en la base de datos. No se ahonda en la lógica de cada función por temas de extensión del documento.

```
def ensure_owner(self, request: Request, application: Application) -> None:
    """Ensures that the user is the owner of the application.

    Args:
        request (rest_framework.request.Request): The request object.
        application (Application): The application object.

    Raises:
        rest_framework.exceptions.PermissionDenied: PermissionDenied exception to raise when theuser is not the owner of the application.
    """

def check_scan_exists(self, scan_serialized_dict: dict) -> bool:
    """Check if a scan with the same data already exists in the database.

    Args:
```

```
        scan_serialized_dict (dict): The serialized scan data.

    Returns:
        bool: True if a scan with the same data already exists in the database, False otherwise.
    """

def standardize_date(self, date_str: str) -> datetime:
    """Standarize the date string to ISO 8601 format.

    Args:
        date_str (str): The date string to standarize.

    Raises:
        ValueError: ValueError to raise when the date string cannot be parsed.

    Returns:
        datetime: The date string in ISO 8601 format.
    """

def serialize_scan_data(self, application_id: int, tool: str, report_dict: dict) -> dict:
    """Serializes the scan data.

    Args:
        application_id (int): The ID of the application.
        tool (str): The tool that generated the JSON file.
        report_dict (dict): The parsed report dictionary.

    Returns:
        dict: The serialized scan data.
    """

def check_vuln_exists(self, vuln_serialized_dict: dict, application_id: int) -> bool:
    """Check if a vulnerability with the same data already exists in the database.

    Args:
        vuln_serialized_dict (dict): The serialized vulnerability data.
        application_id (int): The ID of the application.

    Returns:
        bool: True if a vulnerability with the same data already exists in the database, False otherwise.
    """

def serialize_vuln_data(self, scan_id: int, alert: dict) -> dict:
    """Serializes the vulnerability data.

    Args:
        scan_id (int): The ID of the scan.
        alert (dict): The alert dictionary.

    Returns:
```

```
        dict: The serialized vulnerability data.
    """

def get_file_from_request(self, request: Request) -> IO[bytes]:
    """Gets the file from the request and returns it.

    Args:
        request (rest_framework.request.Request): The request object.

    Raises:
        self.NoFileProvided: APIException to raise when no file is
    provided in the request. Status code 400.

    Returns:
        file: The file object.
    """

def parse_and_serialize_scan(self, tool: str, file: IO[bytes],
    application_id: int) -> tuple[dict, dict]:
    """Parses the report and serializes the scan data.

    Args:
        tool (str): The tool that generated the JSON file.
        file (file): The JSON file generated by the tool.
        application_id (int): The ID of the application.

    Returns:
        dict, dict: The parsed report dictionary and the serialized scan
    dictionary.
      """

def handle_existing_scan(self, scan_serialized_dict: dict) -> None:
    """Handles the case when a scan with the same data already exists in
    the database.

    Args:
        scan_serialized_dict (dict): The serialized scan data.

    Raises:
        self.ScanAlreadyExists: APIException to raise when a scan with the
     same data already exists in the database. Status code 400.
    """

def save_scan(self, scan_serialized_dict: dict) -> Scan:
    """Saves the scan data to the database.

    Args:
        scan_serialized_dict (dict): The serialized scan data.

    Returns:
        Scan: The saved Scan object.
    """

def handle_vulnerability(self, scan: Scan, scan_dict: dict, application_id
    : int) -> bool:
    """Handles the vulnerability data.
```

```
116
117     Args:
118         scan (Scan): The scan object.
119         scan_dict (dict): The scan data.
120         application_id (int): The ID of the application.
121
122     Returns:
123         bool: True if at least one new vulnerability was added, False
    otherwise.
124     """
125
126 def manage_existing_vulnerability(self, scan: Scan, vuln_serialized_dict:
    dict, application_id: int) -> bool:
127     """Manages the case when a vulnerability already exists in the
    database.
128
129     Args:
130         scan (Scan): The scan object.
131         vuln_serialized_dict (dict): The serialized vulnerability data.
132         application_id (int): The ID of the application.
133
134     Returns:
135         bool: True if the vulnerability was updated, False otherwise.
136     """
137
138 def save_new_vulnerability(self, vuln_serialized_dict: dict) -> bool:
139     """Saves a new vulnerability to the database.
140
141     Args:
142         vuln_serialized_dict (dict): The serialized vulnerability data.
143
144     Returns:
145         bool: True if the vulnerability was saved, False otherwise.
146     """
```

Listing A.13: Funciones auxiliares API

A.8. Seguridad

En esta sección se abordan detalles acerca de las medidas de seguridad implementadas sobre la plataforma. En A.14 se muestra la configuración para levantar el honeypot en la ruta `admin/` del servidor, junto a la creación de los admins de la plataforma y la configuración del servidor de correo electrónico para poder notificar a estos. Por otro lado, el código A.15 muestra la configuración para establecer la cabecer "`Content-Security-Policy`" mediante el uso de la librería "django-csp". Finalmente, el código A.16 muestra las configuraciones establecidas para las opciones de seguridad ofrecidas de manera nativa por Django.

```
1 # settings.py
2 # Admins
3 ADMINS = [("Admin", "admin_email@example.com")]
4
5 # Email settings
```

```
EMAIL_BACKEND = "django.core.mail.backends.smtp.EmailBackend"
EMAIL_HOST = os.environ.get("EMAIL_HOST", "")
EMAIL_PORT = os.environ.get("EMAIL_PORT", "")
EMAIL_HOST_USER = os.environ.get("EMAIL_HOST_USER", "")
EMAIL_HOST_PASSWORD = os.environ.get("EMAIL_HOST_PASSWORD", "")

# urls.py
urlpatterns = [
    ...
    path('admin/', include('admin_honeypot.urls', namespace='
    admin_honeypot')),
    path(os.environ.get("ADMIN_PATH"), admin.site.urls),
    ...
]
```

Listing A.14: Configuración del honeypot

```
# Content Security Policy settings
CSP_DEFAULT_SRC = ("'self'", )
CSP_STYLE_SRC = ("'self'", "https://cdn.jsdelivr.net", "https://cdnjs.
    cloudflare.com", "https://cdn.datatables.net")
CSP_SCRIPT_SRC = ("'self'", "https://ajax.googleapis.com", "https://cdn.
    jsdelivr.net", "https://cdnjs.cloudflare.com", "https://cdn.datatables.
    net")
CSP_IMG_SRC = ("'self'", "data:")
CSP_FONT_SRC = ("'self'", "https://cdnjs.cloudflare.com/")
CSP_CONNECT_SRC = ("'self'", "https://cdn.datatables.net")
CSP_FORM_ACTION = ("'self'", )
CSP_FRAME_ANCESTORS = ("'none'", )
```

Listing A.15: Configuración CSP

```
# Security settings
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')
SESSION_COOKIE_SECURE = True
CSRF_COOKIE_SECURE = True
SECURE_SSL_REDIRECT = True
SECURE_BROWSER_XSS_FILTER = True
SECURE_CONTENT_TYPE_NOSNIFF = True
SECURE_HSTS_SECONDS = os.environ.get("HSTS_SECONDS", 0)
SECURE_HSTS_INCLUDE_SUBDOMAINS = os.environ.get("HSTS_INCLUDE_SUBDOMAINS",
    "False")
SECURE_HSTS_PRELOAD = os.environ.get("HSTS_PRELOAD", "False")
```

Listing A.16: Opciones de seguridad nativas de Django

A.9. GitHub Workflows

A continuación, se detallan las configuraciones especificadas en los archivos *YAML* para establecer los *workflows* de integración continua y detección de secretros utilizados durante el desarrollo de la plataforma. Estas configuraciones aseguran que el código se pruebe y se valide automáticamente con cada cambio, además de verificar la presencia de información sensible, manteniendo así la seguridad y la integridad del desarrollo.

```yaml
name: Django CI

on:
  push:
    branches: [ "main", "dev", "test" ]
  pull_request:
    branches: [ "main", "dev" ]

jobs:
  test:
    runs-on: ubuntu-latest

    services:
      mysql:
        image: mysql:8.0
        env:
          MYSQL_DATABASE: "test_db"
          MYSQL_USER: "test_user"
          MYSQL_PASSWORD: ${{ secrets.DB_TEST_PASSWORD }}
          MYSQL_ROOT_PASSWORD: ${{ secrets.DB_TEST_PASSWORD }}
        ports:
          - '3306:3306'
        options: >-
          --health-cmd="mysqladmin ping -h localhost"
          --health-interval=10s
          --health-timeout=5s
          --health-retries=3

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: 3.11

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Wait for MySQL
      run: |
        until mysqladmin ping -h "127.0.0.1" --silent; do
          echo 'waiting for mysql...'
          sleep 5
        done

    - name: Run Django tests
      env:
        SECRET_KEY: ${{ secrets.SECRET_KEY }}
        DB_NAME: "test_db"
        DB_USER: "root"
        DB_PASSWORD: ${{ secrets.DB_TEST_PASSWORD }}
        DB_HOST: '127.0.0.1'
        DEBUG: 'False'
```

```
      run: |
        cd vuln_mngmnt
        python manage.py test app_web.test.test_api
        python manage.py test app_web.test.test_models
        python manage.py test app_web.test.test_views
```

Listing A.17: Configuración del workflow Django test

```
name: Gitleaks Scan

on:
  push:
    branches: [ "main", "dev" ]
  pull_request:
    branches: [ "main", "dev" ]

jobs:
  gitleaks:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Install gitleaks
      run: |
        wget https://github.com/gitleaks/gitleaks/releases/download/v8
    .18.3/gitleaks_8.18.3_linux_x64.tar.gz
        tar -xvf gitleaks_8.18.3_linux_x64.tar.gz
        sudo mv gitleaks /usr/local/bin/

    - name: Run gitleaks scan
      run: gitleaks detect --verbose
```

Listing A.18: Configuración del workflow Gitleaks