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

SISTEMA DE PRUEBAS DE PENETRACIÓN AUTOMATIZADAS PARA APLICACIONES WEB

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

GUILLERMO ENRIQUE PILLEUX GALLARDO

PROFESOR GUÍA:
ALEJANDRO HEVIA ANGULO

MIEMBROS DE LA COMISIÓN:
TOMÁS BARROS ARANCIBIA
EDUARDO RIVEROS ROCA
ÉRIC TANTER

SANTIAGO DE CHILE
2021

RESUMEN

El trabajo presentado en esta memoria implementa una **herramienta modular** que **automatiza** y **centraliza** la ejecución simultánea de **múltiples herramientas de *pentesting***, estandarizando los resultados de las herramientas para que puedan ser utilizados por usuarios con conocimientos básicos de ciberseguridad. La solución desarrollada permite agregar herramientas de *pentesting* adicionales con la implementación de los módulos necesarios correspondientes que los integren a la solución.

Para corroborar el correcto funcionamiento en ambientes programados con distintos lenguajes, la herramienta se probó en ambientes con vulnerabilidades documentadas como OWASP Mutillidae, Railsgoat, Google Gruyere, los cuales son ambientes programados en PHP, Ruby on Rails y Django (Python), respectivamente. En estos tres ambientes se detectó un 46 % de las vulnerabilidades documentadas, lo que exhibe la capacidad de detección de la herramienta. A partir de los resultados, se observó que los *scanners* descubrieron distintos tipos de vulnerabilidades y, de las detectadas del mismo tipo, descubrieron diferentes vulnerabilidades, lo que indica que cada *scanner* posee una capacidad particular de detección de vulnerabilidades.

La herramienta desarrollada se incorporará al pipeline de integración continua de la empresa Fintual AGF, una Administradora General de Fondos Mutuos regulada por la Comisión para el Mercado Financiero (CMF). Como la plataforma web está en constante evolución, aumenta la probabilidad de introducir código vulnerable, por lo que la herramienta desarrollada será clave para sistematizar y estandarizar la búsqueda de vulnerabilidades críticas antes de que el código desarrollado pase a producción.

Se ejecutó la herramienta desarrollada en la plataforma web de Fintual, encontrándose 3 vulnerabilidades de riesgo alto y 2 vulnerabilidades de riesgo medio. Se diseñó un *Dashboard* para visualizar los resultados, tanto históricos como el detalle del último análisis, de manera que los usuarios tecnológicos y no tecnológicos sean capaces de entender el estado de seguridad de la aplicación y puedan resolver rápidamente las vulnerabilidades detectadas.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Introducción

La motivación para desarrollar este trabajo de memoria se fundamenta en el problema recurrente de presencia de vulnerabilidades críticas en los sistemas tecnológicos en distintos contextos y ambientes industriales. Para ello se han desarrollado múltiples herramientas capaces de detectar vulnerabilidades, las que deben ser ejecutadas por especialistas y, debido a esto, se deben invertir recursos económicos tanto en el aspecto de capital humano, para la contratación de dichos especialistas, como en el de desarrollo y adaptación de dichas herramientas para el caso particular de su organización. Por esta razón, este trabajo de memoria desarrolla una plataforma que simplifica el uso de diversas herramientas de detección de vulnerabilidades, automatizando y centralizando la ejecución de las pruebas de penetración de aplicaciones o *Penetration Testing*.

Es necesario resaltar que la principal solución para eliminar vulnerabilidades en los sistemas tecnológicos es que éstos sean diseñados y desarrollados sin vulnerabilidades desde su concepción, siendo el *Pentesting* un proceso complementario que sirve al control de calidad.

El *Penetration Testing*, también conocido como *Pentesting* o *hackeo ético*, es la práctica de probar y auditar los sistemas computacionales o aplicaciones web para encontrar sus vulnerabilidades de seguridad, las que podrían ser utilizadas por un atacante externo para fines maliciosos. Un *pentester* es quien demuestra y documenta la existencia de vulnerabilidades en los sistemas, buscando hacerlo antes que un tercero malicioso las pueda detectar y explotar, y entrega recomendaciones de mitigaciones y adaptaciones a las políticas de seguridad de los sistemas, aportándole argumentos al Oficial Jefe de Seguridad (CISO, por sus siglas en inglés) para justificar y asegurar presupuestos adecuados para el área de ciberseguridad.

Es de suma importancia que aquellas organizaciones que posean activos tecnológicos realicen este tipo de pruebas en forma periódica, debido a que el área de la ciberseguridad está constantemente evolucionando, por lo que se generan y encuentran nuevas vulnerabilidades en los sistemas. Si no se realizan estas pruebas, las organizaciones se podrían ver expuestas a ataques cibernéticos, colocando en riesgo tanto la confidencialidad, integridad o disponibilidad de sus sistemas.

Un caso específico son las organizaciones que utilizan metodologías de desarrollo ágil como el desarrollo continuo, en el cual sus activos tecnológicos están periódicamente sufriendo pequeñas modificaciones, pues ello amplía la exposición a una mayor cantidad de vulnerabilidades por la inevitable y casual introducción de código vulnerable a los sistemas si este no se audita de forma diligente.

La realización de pruebas en forma periódica presenta como problema el alto costo que puede llegar a tener adquirir un servicio como éste, debido a que requiere de *pentesters*, los cuales son escasos, por lo que su servicios son altamente cotizados en el rubro. Además, es importante considerar un presupuesto permanente debido a que constantemente se encuentran nuevas vulnerabilidades en los sistemas, por lo que existe la necesidad de verificar que éstos estén actualizados. Además, la metodología de desarrollo ágil, debido a que una de sus características es la constante evolución de las funcionalidades del sistema, genera como contrapartida el incremento de códigos que puedan llegar a generar vulnerabilidades.

Asimismo, los *pentesters* poseen una gran variedad de herramientas de distintas funcionalidades para llevar a cabo estas pruebas de seguridad. A pesar que algunas de estas herramientas sean automáticas, aún así el analista requiere configurarlas para la ejecución específica, por lo que su labor puede ser bastante repetitiva. Además, la ejecución de las herramientas no es instantánea, sino que son intensivas en tiempo, lo que puede traer dificultades en el ámbito operacional de una organización, debido a que estas pruebas deben realizarse sobre el núcleo del sistema o red, lo cual puede llevar a disrupciones en las actividades de producción, por lo que, es útil sistematizar su uso y maximizar su automatización.

Además, las numerosas herramientas que utilizan los analistas de seguridad no están centralizadas y se requiere configurarlas manual e individualmente, lo cual aumenta la labor a realizar y la complejidad de la misma. Si bien existen distintas herramientas para realizar *pentesting*, es complejo encontrar una herramienta que las englobe, estandarice, automatice su ejecución, y unifique los resultados de éstas.

La adopción del desarrollo constante en el aspecto tecnológico a lo largo del tiempo ha generado que los sistemas sean cada vez más complejos, por lo que, existe una necesidad imperante de lograr su simplificación y se ha identificado que la automatización y la democratización de la experiencia es el camino que se debe seguir para lograr el fin. La automatización y democratización de la experiencia son una valla fundamental que se espera superar.

En este trabajo de memoria se abordará la creación y ejecución de una plataforma que centraliza, automatiza y estandariza el uso de las herramientas de *pentesting* dentro de un ambiente de desarrollo continuo, ejecutándolas en conjunto con la pila de pruebas de desarrollo de una organización. Esta plataforma tendrá una arquitectura modular que posea la capacidad de agregar nuevas herramientas de *pentesting* en base a los requerimientos del usuario. El objetivo será desarrollar e implementar una herramienta que brinde un punto de partida al analista de seguridad que facilite y simplifique la realización de un *pentest*.

Este trabajo de memoria se aplicó en la empresa fintech Fintual AGF, la cual implementa metodologías ágiles de desarrollo y no posee herramientas centralizadas o estandarizadas en relación a las pruebas de vulnerabilidades de su plataforma web.

La herramienta desarrollada se vuelve relevante para Fintual AGF porque le proporciona una base de análisis de vulnerabilidades a su plataforma de manera automatizada y aporta a la integración completa del ciclo de desarrollo, ayudándoles a analizar y corregir las vulnerabilidades que se presentan.

Se conlleva una metodología experimental, donde se planteó una solución y se pretende

validarla mediante experimentos denominados laboratorios de aplicaciones web vulnerables. Las etapas de la metodología de trabajo seguida en esta memoria son las siguientes:

1. Análisis de la situación actual. Estudio y evaluación del rubro de la ciberseguridad y las herramientas utilizadas actualmente.
2. Selección de herramientas de análisis de vulnerabilidades.
3. Selección de aplicaciones web vulnerables como ambientes de verificación.
4. Configuración del ambiente de trabajo y de las herramientas seleccionadas.
5. Estudio de uso, validación de funcionamiento y automatización de las herramientas seleccionadas.
6. Estudio de la arquitectura y programación de la solución propuesta.
7. Validación de la solución propuesta.
8. Integración de la herramienta desarrollada en el ambiente de desarrollo de Fintual.

Capítulo 1

Antecedentes Generales

En este capítulo se realiza una introducción al *Pentesting*, se continua con una revisión de las herramientas más utilizadas en la actualidad, luego se describen los objetivos de este trabajo de memoria, junto con la solución propuesta, los resultados esperados, los alcances y finalmente, se describe el contexto en el que se aplica el trabajo de memoria.

1.1. Introducción al Pentesting

El arte del *Pentesting* es la práctica de probar y auditar sistemas computacionales o aplicaciones web para encontrar sus vulnerabilidades de seguridad, las que de no ser detectadas, podrían ser utilizadas por un atacante externo para fines maliciosos. Esto se puede llevar a cabo mediante distintos métodos donde cada cual podría tener un resultado distinto, dependiendo del enfoque o punto de vista que se le puede llegar a dar. Estos métodos pueden contemplar las siguientes 7 fases [1]:

1. **Pre-Engagement Actions o Acciones de Pre-compromiso.** Es la fase en la que se define el alcance del análisis, como la definición de los activos que se pondrán a prueba, bajo qué métodos se deben probar y qué activos no se deben manipular. En esta fase también se definen las expectativas del usuario, las implicancias legales y los objetivos.
2. **Reconocimiento**. Es la fase de búsqueda de información sobre el objetivo y los activos tecnológicos definidos en el alcance, lo cual profundiza en las tecnologías utilizadas e intenta comprender la lógica de negocio implementada en la plataforma de manera de detectar brechas de seguridad en estos ámbitos.

   Algunos métodos comúnmente utilizados en esta fase son la utilización de motores de búsqueda sobre los empleados, los sistemas y las tecnologías del cliente, la búsqueda de dominios y subdominios, al igual que métodos físicos como *dumpster diving* para la búsqueda de documentos de utilidad en los basureros o *tailgating* para recopilar información sobre los accesos físicos de las facilidades, por mencionar algunos ejemplos.
3. **Modelamiento de Amenazas e Identificación de Vulnerabilidades**. La siguiente fase se puede realizar una vez que se posee suficiente información del objetivo, con la cual se debe realizar un modelo de las amenazas que el cliente podría enfrentar de manera realista e identificar las vulnerabilidades que permitan realizar estos ataques.

Para esto, se pueden utilizar escáners automáticos para detectar los puertos disponibles de los sistemas y las vulnerabilidades de la red con lo que se busca determinar si los sistemas están funcionando como también la detección de software preventivo que podría estar presente como un *firewall*, un antivirus o un detector de intrusos.

Para realizar esta fase, es necesario colocarse en el lugar de un atacante y pensar en las distintas posibilidades en que se podrían utilizar los recursos del cliente para incurrir en un ataque y documentar los posibles vectores de ataques.

4. **Explotación**. En esta fase se lleva a cabo el plan de acción documentado en el modelamiento de amenazas con el objetivo de penetrar los sistemas y obtener acceso a ellos. En la medida que el analista obtiene acceso a los sistemas, este va documentando constantemente sus hallazgos, además de ir recopilando mayor información sobre los mismos sistemas que le permitan realizar ataques de mayor complejidad.
5. **Persistencia del acceso y movimientos laterales**. Es la etapa en la que el *pentester* introduce a los sistemas agentes que permiten la persistencia del acceso obtenido, es decir, el analista no debe volver a realizar todos los pasos necesarios para obtener acceso, sino que el agente introducido le permite tener acceso directo a los sistemas vulnerados aún así si los sistemas fueran reiniciados.

   Al igual que en la fase anterior, el analista siempre está en la búsqueda de mayor información de los sistemas y en esta etapa tiene el objetivo de realizar movimientos laterales para encontrar otros servicios que estén alojados en el sistema y que inicialmente no tenía visibilidad. De modo de ejemplificar, es común que un servidor no aloja una única aplicación web, sino que varias al mismo tiempo, por lo que al realizar el análisis de una de ellas, es posible obtener acceso la información de las demás y poder realizar acciones sobre ellas.
6. **Recolección y destrucción de artefactos**. La fase posterior a explotación de las vulnerabilidades y persistencia del acceso asegura que los sistemas vuelvan al estado inicial previo a ser manipulados por el analista, proceso en el cual se eliminan los agentes introducidos, se revierten las contraseñas de credenciales cambiadas, se eliminan usuarios creados y se realiza la recolección de la información obtenida.
7. **Reporte**. La fase final de un *pentest* resulta ser la generación de un reporte que incorpore los hallazgos del análisis, junto con posibles formas de remedición para cada vulnerabilidad detectada. El reporte estará dirigido a los ejecutivos de negocios al igual que al equipo técnico por lo que debe estar confeccionado de tal forma que contenga los detalles técnicos como también explicaciones de las implicancias de los hallazgos para un público no técnico.

Como existen distintos activos que se relacionan con los sistemas tecnológicos, también existen distintos tipos de *pentesting* cuyo foco son los diferentes elementos que involucran a éstos. A continuación se listan los diferentes tipos de *pentesting* y se explica de manera general en que consiste cada uno.

1. **Pentest de los Servicios de Redes o Pentest de Infraestructura**. El objetivo principal es detectar las vulnerabilidades de mayor exposición de la infraestructura de la red, como los servidores, *firewalls*, *switches*, rutificadores, impresoras, computadoras de los puestos de trabajo, etc. Este tipo de *pentest* es de los más comunes debido al gran impacto que puede incurrir un ataque a estos activos, el cual influye directamente

en la continuidad del negocio.

2. **Pentest de Aplicaciones web y/o móvil**. El objetivo es descubrir las vulnerabilidades de las aplicaciones para el cual se utilizan métodos de mayor especificidad debido a que es necesario poner a prueba todas las rutas con las que interactúan los usuarios de manera cotidiana. Las razones fundamentales por la que es necesario realizar este tipo de pruebas es para asegurar las componentes de la aplicación como las bases de datos y los servidores mismos.
3. **Client Side Pentest**. Consiste en realizar análisis de vulnerabilidades de los servicios con los que interactúan los empleados de manera directa y constante, como son los *browsers*, los servicios de correo, Adobe Photoshop o Microsoft Office Suite, por mencionar algunos. La relevancia de poner a prueba estos servicios es tal que si los empleados reciben documentos maliciosos y los abren o ejecutan, no se infecte el sistema personal o aún más crítico, la red completa, lo que podría intervenir con la continuidad del negocio.
4. **Wireless Pentest o Pentest de Redes Inalámbricas**. Consiste en examinar todas las conexiones y los dispositivos conectados a la red inalámbrica (WiFi). Es de suma importancia este tipo de *pentest* debido a que las redes inalámbricas son comunicaciones invisibles entre dispositivos, por lo que un acceso no autorizado que implique una brecha de información no podría ser detectada.
5. **Pentest de Ingeniería Social**. Es donde el analista se hace pasar por alguien de confianza, engañando al usuario para que, sin el usuario estar al tanto del objetivo del analista, le entregue información sensible, como nombres de usuarios, contraseñas u otros datos que podrían ser utilizados para campañas de ingeniería social posteriores o para obtener acceso no autorizado de los sistemas. Este tipo de *pentest* tiene gran importancia debido a que las personas son uno de los vínculos más débiles de una organización y resultan ser uno de los factores común en los ciberataques.
6. **El Pentest Físico**. Busca simular una amenaza real donde el analista tiene el objetivo de comprometer las entradas físicas de un edificio u oficinas, los puestos de trabajo e incluso a los mismos empleados, para así identificar las debilidades en los puntos de control físicos. La relevancia de este tipo de *pentest* es alta debido al impacto que puede llegar a tener, si por ejemplo un agente malicioso tuviese acceso no autorizado a los servidores físicos de una organización.

Cada tipo de *pentest* tiene sus propios métodos para conllevarlos, como también el grado de relevancia dependiendo de los activos tecnológicos que posea cada organización.

Este trabajo se focaliza en el *pentesting* de aplicaciones web.

1.2. Herramientas de Pentesting

Existen numerosas herramientas que son utilizadas para llevar a cabo las labores que involucran los distintos tipos de *pentests* descritos con anterioridad, por lo que en esta subsección se mencionarán y describirán a grandes rasgos algunas de las herramientas utilizadas para realizar dichas labores.

Para comenzar, es importante mencionar el sistema operativo Kali Linux, el cual le brinda

al usuario un arsenal compuesto de herramientas comúnmente utilizadas por los *pentesters*. Este conjunto resulta conveniente para el *pentester* debido a la practicalidad que brinda al realizar pruebas de penetración, por el hecho de tener una gran variedad de herramientas centralizadas en un mismo sistema lo que ahorra los tiempos de instalación de cada una de las herramientas.

Dentro de las herramientas más comunes están los *scanners*, los cuales realizan un análisis de las vulnerabilidades que podría presentar una aplicación. En general, los *scanners* pueden ser el punto de partida para la exploración de vulnerabilidades debido a que detecta la vulnerabilidad y explicita por qué podría ser un problema para el sistema. Sin embargo, los *scanners* son limitados cuando se trata de la detección de vulnerabilidades más sofisticadas.

A continuación se listan las herramientas comúnmente utilizadas para realizar análisis de vulnerabilidades en los servicios de redes de una aplicación:

1. **Zmap** [2]. *Scanner* de redes utilizado para recopilar información inicial de la red objetivo.
2. **NMAP** [3]. *Scanner* de redes para identificar qué tecnologías utiliza su objetivo y qué puertos utilizan estas tecnologías para su comunicación.
3. **Metasploit** [4]. Se centra principalmente en vulnerabilidades de infraestructura pero también tiene módulos para explotar aplicaciones web.
4. **Armitage** [5]. Interfaz gráfica que automatiza las funcionalidades de Metasploit.

Las herramientas más populares usadas en el *Pentesting* de tipo *Wireless* son Wireshark y Aircrack-ng. El primero es un rastreador de red que permite generar trazabilidad de los paquetes generados y tiene la capacidad de interceptar el tráfico en la red, pudiendo, potencialmente, tener lectura de información sensible. La segunda es una herramienta de penetración de redes WiFi que se destaca por tener la capacidad de inyectar paquetes a la red. Esto último podría permitir enviar mensajes construidos de forma especial y engañar a los dispositivos en la red.

Este trabajo de memoria se centra en la detección de vulnerabilidades en aplicaciones web. Por ello, se listan las herramientas utilizadas para estos análisis, las cuales se subdividen en las categorías según su accesibilidad como las herramientas de código abierto (*open source*) o las herramientas comerciales que requieren comprar una licencia para su uso.

Herramientas *open source* de análisis de vulnerabilidades de aplicaciones web:

1. **Zed Attack Proxy (ZAP)** [6]. *Scanner* que realiza análisis de código dinámico de aplicaciones web,
2. **Wapiti** [7]. *Scanner* que realiza análisis de código dinámico de aplicaciones web,
3. **W3AF** [8]. *Scanner* que realiza análisis de código dinámico de aplicaciones web,
4. **Nikto** [9]. *Scanner* de servidores web,
5. **Astra** [10]. *Scanner* de API REST,
6. **Dawnscanner** [11]. *Scanner* de vulnerabilidades específico para aplicaciones programadas en Ruby,

7. **WS-Attacker** [12]. Framework modular para realizar pruebas de vulnerabilidades particularmente de API SOAP,
8. **Codeburner** [13]. *Framework* que contempla múltiples herramientas de análisis de código estático, por ejemplo Brakeman [14] o Dawnscanner.

Las herramientas comerciales de análisis de vulnerabilidades en aplicaciones web son las siguientes:

1. **Burp Suite Pro** [15]. Herramienta *proxy* y *scanner* web que permite visualizar el comportamiento de una aplicación web mientras se interactúa con ésta. Esta aplicación posee una versión gratuita, la cual posee menor grado de automatización.
2. **Tenable.io Web App Scanning** [16]. Es uno de los productos de Tenable que busca automatizar el *pentest* de aplicaciones web. Este producto permite visualizar los activos tecnológicos de una red junto con la identificación de sus vulnerabilidades. Además, tiene la capacidad de priorizar las vulnerabilidades de mayor impacto y brinda una propuesta de solución para resolverlas.
3. **Nexpose** [17]. *Scanner* de vulnerabilidades tanto de infraestructura como de aplicaciones web.
4. **Pentoma** [18]. Herramienta que automatiza el pentest de aplicaciones web y utiliza inteligencia artificial para validar los falsos positivos.

1.3. Objetivos

1.3.1. Objetivo General

El objetivo principal de este trabajo de memoria es diseñar, desarrollar e implementar una herramienta que simplifique y facilite la realización de pruebas de penetración de aplicaciones web que le permitan a un usuario del área de tecnología realizar análisis de la seguridad de una aplicación web.

El objetivo de la herramienta propuesta no sustituye al analista de seguridad.

1.3.2. Objetivos Específicos

1. Desarrollar una herramienta que detecte vulnerabilidades en aplicaciones web a partir de patrones de ataques preconfigurados. Se validará realizando laboratorios de aplicaciones web con y sin vulnerabilidades, donde se medirá la precisión de detección de las vulnerabilidades, es decir, cuántas vulnerabilidades detecta la herramienta de todas las presentes.
2. Desarrollar una herramienta con arquitectura modular que permite su extensión y modificación en el tiempo. La herramienta incluirá al menos tres tecnologías de *pentesting*.
3. Desarrollar e implementar un módulo que le permita al usuario configurar los tipos de ataques a partir de opciones preestablecidas. Se evaluará si la herramienta le permite al usuario seleccionar tipos de ataques y ejecutarlos.
4. Probar la herramienta en la aplicación web de una empresa. Generar un reporte analítico de los resultados de los análisis de vulnerabilidades de dicha empresa.

Originalmente se propuso como objetivo específico el desarrollar e implementar un módulo que extraiga los reportes de las vulnerabilidades de bases de vulnerabilidades comunitarias en repositorios públicos e incorpore los *exploits* de dichas vulnerabilidades en la herramienta, para así definir nuevos vectores de ataque para probar la aplicación y medir el porcentaje de vulnerabilidades que contiene la solución y el tiempo entre que se publica la vulnerabilidad en las bases comunitarias y que ésta es incorporada a la herramienta. Sin embargo, no fue necesario utilizar bases de vulnerabilidades comunitarias porque las herramientas seleccionadas ya contienen los módulos de vulnerabilidades, por lo que este objetivo específico no se abordó.

1.4. Solución Propuesta

Se propone el desarrollo de un *framework* modular que consolida herramientas de *pentesting* y funciona como una plataforma multidisciplinaria que le proporciona al usuario la posibilidad de realizar análisis más intensos y efectivos, ofreciendo una gama de opciones para el análisis de vulnerabilidades, además la realización de los análisis pueda hacerse en forma paralela y automatizada.

La plataforma propuesta tiene como principal función automatizar la ejecución de múltiples herramientas de *pentesting*, las cuales realizarán los análisis de vulnerabilidades para que la plataforma consolide los resultados, lo cual, elimina la necesidad de ejecutar cada una de ellas de forma individual.

Además, uno de los rasgos particulares que se desprenden del sistema modular, es que la plataforma permite la adhesión a un tercero desarrollador quien puede agregar una nueva herramienta, generando una evolución de la plataforma en base al funcionamiento y la necesidad de los mismos usuarios. Debido a esto, una de las posibilidades que se le brinda al usuario es la utilización y agregación de las herramientas de *pentesting*.

Cada una de las herramientas está programada de forma particular, por lo que poseen distintas capacidades de detección de vulnerabilidades y un formato de resultados propio para cada una, lo que conlleva que la obtención de resultados no tenga un formato estándar. Esta plataforma modular soluciona este problema centralizando la utilización de herramientas, permitiéndole al usuario realizar un análisis más amplio aplicando múltiples herramientas mediante solo una plataforma.

A medida que se integren diferentes tipos de herramientas de análisis de vulnerabilidades, la plataforma obtendrá la capacidad de detectar vulnerabilidades del rubro de dicha herramienta. La plataforma viene en su diseño original con herramientas para detectar vulnerabilidades en aplicaciones web y está adaptada para operar en un ambiente de desarrollo continuo. Sin embargo, esta plataforma también puede incluir otras herramientas para detectar otros tipos de vulnerabilidades.

1.5. Resultados Esperados

Uno de los resultados esperados de este trabajo de memoria es un aporte a la democratización de servicios tecnológicos en el área de la ciberseguridad. Se espera desarrollar una plataforma que le permita a un usuario del área de tecnología la capacidad de realizar análisis de vulnerabilidades en aplicaciones web. La herramienta propuesta no sustituye al analista de seguridad.

Como la arquitectura de la plataforma es modular, se espera que sea capaz de integrar múltiples herramientas de *pentesting* mediante la implementación de módulos respectivos de cada herramienta. Asimismo, se espera que uno de los módulos de la solución permita al usuario configurar las herramientas para realizar un análisis de vulnerabilidades específicas y, además, para analizar secciones particulares de la aplicación en cuestión.

1.6. Alcances

El desarrollo de este trabajo de memoria se centra en la automatización, centralización, estandarización, simplificación y estratificación del análisis de vulnerabilidades de un sistema. Como la plataforma estará inserta en un proceso evolutivo al que se le podrán integrar herramientas adicionales, los primeros *scanners* sólo estarán dirigidos a encontrar las vulnerabilidades que las herramientas ya rastrean y no está dirigida a crear herramientas para la detección de nuevas vulnerabilidades. La plataforma permitirá que el usuario incorpore nuevas herramientas de análisis de vulnerabilidades a su catálogo para incrementar su capacidad de detección, dado el supuesto que las herramientas añadidas son capaces de detectar vulnerabilidades distintas.

Como el análisis de vulnerabilidades es una operación delicada y requiere de conocimiento, interpretación y experiencia, siempre requerirá de un analista que las verifique y valide, por lo que su alcance es el de una herramienta de consultoría cuyo objetivo es el de entregar mayor y mejor información por lo que la validación de falsos positivos no es el enfoque de la plataforma.

Se requiere que el analista que utilice esta herramienta posea, por lo menos, conocimientos básicos de seguridad para entender e interpretar los resultados. En particular, esto se aplica cuando la herramienta establece que ha encontrado una vulnerabilidad, y como ello no significa que dicha vulnerabilidad está presente en el sistema, es el usuario el que debe aplicar sus conocimientos para confirmar o descartar la amenaza.

Asimismo, la plataforma es una herramienta más del área de ciberseguridad y busca ser un aporte al análisis de uno de los controles establecidos en la lista de *Critical Secutity Controls* (CSC) establecido por el *Center for Internet Security* (CIS) [19].

1.7. Caso Práctico: Contexto Fintual

Esta memoria se desarrolló en la empresa Fintual AGF, una Administradora General de Fondos Mutuos (AGF) regulada por la Comisión para el Mercado Financiero (CMF) de Chile. Fintual es una startup tecnológica con tres años de trayectoria que ofrece un servicio de

inversión similar al de los fondos mutuos, cuyo principal activo es su plataforma tecnológica que está en permanente evolución. Fintual ganó el premio a la mejor startup Chilena el 2018 y fue la primera empresa Chilena en ganar el programa de incubadoras estadounidense Y-Combinator (YC) [20].

Este trabajo de memoria se comenzó a desarrollar en Fintual en marzo de 2020. En esta fecha la empresa estaba compuesta de 35 personas y administraban USD$80 millones. A fines de febrero de 2021 ya la integran más de 80 personas (85 % crecimiento) y administran un capital de más de USD$400 millones (500 % crecimiento), con una cartera de más 45.000 clientes [21].

En Fintual, para el desarrollo de su plataforma tecnológica, se basan en la metodología de desarrollo de software ágil *Shape Up* con ciclos de desarrollo de 6 semanas donde los desarrolladores generan propuestas (pitch) y, luego de un período de 2 semanas (cool down), el comité de desarrollo lo evalúa y decide los proyectos que se van a desarrollar en el siguiente ciclo de 6 semanas. Todo el desarrollo realizado en la empresa pasa por un *pipeline* de integración continua donde se ejecutan pilas de pruebas sobre el código que se ha agregado antes que se introduzca en el ambiente de producción. De esta manera se adhieren a un estándar mínimo de calidad respecto al código que generan previo a su incorporación al ambiente de producción.

Fintual está comprometida a tener los más altos estándares de ciberseguridad y posee, entre otros, un programa remunerado de *bug bounty* que promueve la recepción de reportes de vulnerabilidades sugeridos por hackers éticos independientes que ponen a prueba el funcionamiento apropiado de la plataforma tecnológica.

La herramienta desarrollada se incorporará al *pipeline* de integración continua de Fintual y le permitirá sistematizar y estandarizar la búsqueda de vulnerabilidades críticas antes de que el código desarrollado pase a producción. Adicionalmente, la herramienta le permitirá visualizar las vulnerabilidades de su sistema de tal manera que sean capaces de entender el estado de seguridad de la aplicación y puedan resolver rápidamente las vulnerabilidades detectadas.

Capítulo 2

Diseño de la Solución

2.1. Análisis de la situación actual

Para llevar a cabo este trabajo de memoria se realizó un análisis de la situación actual del rubro de la ciberseguridad. Se estudiaron las herramientas existentes y los servicios o productos tecnológicos relacionados al rubro. El análisis se extendió a las herramientas utilizadas para llevar a cabo las labores de la seguridad en la actualidad, se evaluaron las funciones de cada herramienta y la factibilidad de instalarlas en un contenedor y automatizar su ejecución. Este análisis se encuentra en los *Antecedentes Generales* del documento.

2.2. Selección de aplicaciones web vulnerables como ambientes de verificación

Para determinar las aplicaciones web en las cuales se ejecutaría la plataforma objeto de este trabajo de memoria, se seleccionaron aplicaciones web con ambientes vulnerables para la verificación continua de la plataforma que permitieran la demostración empírica del correcto funcionamiento de la plataforma utilizando vulnerabilidades ya catalogadas. Esta demostración exitosa solo comprueba que la plataforma responde adecuadamente a las vulnerabilidades comunes y no es posible concluir con ello que no existan otras vulnerabilidades de mayor sutileza.

Para las pruebas con vulnerabilidades conocidas se seleccionaron tres aplicaciones que contaban con ambientes vulnerables, las que estaban desarrolladas en lenguajes de programación distintos entre sí, pero siendo éstos de los más populares en el desarrollo de dichas aplicaciones. Es necesario comprobar que los ambientes seleccionados no se vean sesgados por un lenguaje en común, lo que muestra que la plataforma no está focalizada sólo en un tipo de lenguaje.

Los ambientes vulnerables seleccionados junto con su lenguaje de programación fueron los siguientes:

1. **OWASP Mutillidae II** - PHP

2. **OWASP Railsgoat** - Ruby on Rails
3. **Google Gruyere** - Django

2.3. Selección de herramientas de análisis de vulnerabilidades

De todas las herramientas existentes en las diversas áreas del pentesting, en este trabajo se escogieron las herramientas enfocadas en el análisis de vulnerabilidades de aplicaciones web.

Los criterios de selección de las herramientas a incorporar en la plataforma de este trabajo de memoria consideraron la aplicabilidad de la herramienta seleccionada, la efectividad de la herramienta para detectar vulnerabilidades, y la confiabilidad de los resultados que arroja la herramienta. Además, las herramientas debían cumplir con los siguientes criterios adicionales de integración: portabilidad en un contenedor; la capacidad de ejecutar la herramienta a través de la línea de comando o que posea una API de interacción; y que posea la capacidad de especificar un ataque particular sobre el objetivo.

Para la elección de de las herramientas se revisaron y evaluaron las herramientas de análisis de vulnerabilidades de aplicaciones web listadas en la *sección 1.2* y se determinó utilizar:

1. **Zed Attack Proxy (ZAP)**, por ser una herramienta popular en la comunidad de ciberseguridad, que posee un equipo activo que mantiene el proyecto actualizado y tiene un contenedor desarrollado donde se puede ejecutar el *scanner*.
2. **Wapiti**, porque cumplió con los criterios de aplicabilidad al tener la capacidad de instalarlo en un contenedor y ejecutarlo mediante la línea de comando.
3. **W3AF**, porque cumplió con los criterios de aplicabilidad al tener la capacidad de instalarlo en un contenedor y ejecutarlo mediante la línea de comando.

2.4. Configuración del ambiente de trabajo y de las herramientas seleccionadas

En primer lugar, se buscaron contenedores que incluyeran las herramientas seleccionadas anteriormente.

Se debe tener en cuenta que dichos contenedores se encontraron en Dockerhub en el caso de Wapiti, en la página oficial de ZAP y, por último, en el caso de W3AF se debió desarrollar una versión propia porque las versiones públicas no eran compatibles con la plataforma (se replicaron las condiciones del contenedor ubicado en Dockerhub y se instalaron las librerías con las versiones adecuadas para compatibilizar con la plataforma).

La etapa siguiente se centró en la programación de la ejecución de las herramientas dentro de su contenedor respectivo, lo cual requirió el uso del comando de Docker que levanta el contenedor de la herramienta y ejecuta el *scanner* acorde a la configuración del usuario. Fue necesario programar un módulo particular que generara este comando de ejecución de manera

automática para cada una de las tres herramientas seleccionadas.

En el caso de ZAP, se programó un *script* que utiliza su API para configurar y ejecutar el *scanner*. En el caso de W3AF, se programó un *script* que genera la configuración con los parámetros del usuario, que luego se le entrega al *scanner* y este lo utiliza para su ejecución. Finalmente, en el caso de Wapiti, al realizar un análisis autentificado, tiene la particularidad de que su funcionamiento es a partir de las *cookies* de la aplicación, por lo que posee un módulo que realiza peticiones para autenticarse y almacena las *cookies* en un archivo. Para esto fue necesario automatizar la ejecución del módulo de la obtención de la *cookie* y luego ejecutar el *scanner* para realizar el análisis autentificado.

Se debe considerar que los contenedores utilizados en este trabajo deben tener acceso a la red externa para hacer peticiones a Internet y/o a una aplicación alojada en la máquina local.

2.5. Estudio de uso, validación de funcionamiento y automatización de las herramientas seleccionadas

Luego de la configuración del ambiente de trabajo se revisó el uso de cada herramienta, lo que incluyó estudiar los detalles de las configuraciones y validar la correcta ejecución de las herramientas a través de la observación de los resultados al momento de escanear los ambientes vulnerables, generando un análisis en aspectos de monitoreo del correcto uso y desarrollo de las herramientas incluidas en la plataforma.

A continuación se automatizó la ejecución de las herramientas seleccionadas mediante un único comando de línea que contiene los parámetros de ejecución.

2.6. Estudio de arquitectura y programación de la solución propuesta

Se estudió la arquitectura y se clasificaron los módulos necesarios que compondrían la solución para continuar con la programación de la arquitectura desarrollada. Los módulos se establecieron en base a los pasos requeridos para ejecutar los *scanners* de vulnerabilidades y basándose en el flujo de información entre éstos y el usuario.

En primer lugar se determinó cómo automatizar las herramientas seleccionadas y para ello se programó la generación del comando de ejecución con los parámetros para el análisis de un ambiente en específico. Por lo tanto, se programó el módulo **Argument Parser** que está encargado de generar el comando para ejecutar la herramienta en función de los parámetros que recibe. Una vez que se logró que la ejecución del *scanner* se realice de manera automática, se comenzaron a obtener resultados de los escaneos y se debió estandarizar el formato de los resultados, para lo cual se creó el módulo **Results Parser**. Éste recibe el archivo de resultados de cada herramienta y genera el objeto en el formato JSON definido.

Adicionalmente, como la realización de los escaneos se ejecutan de manera individual, se requiere programar la ejecución consolidada de los escaneos y los resultados obtenidos sean

estandarizados. Para ello se creó un módulo que gestiona todos estos procesos: el módulo **Scanner Master**. Este módulo está encargado de entregarle los parámetros al **Argument Parser** para generar el comando de ejecución, llevar a cabo la ejecución con este mismo comando y, al finalizar el análisis, le indica el archivo de resultados de la herramienta al módulo **Results Parser** para estandarizarlos.

Una vez que se obtienen los resultados en el formato estándar, como éstos aún permanecen en archivos separados generados por cada **Results Parser**, se creó el módulo **Results Merger** que se encarga de consolidar los archivos estandarizados de cada herramienta en un solo archivo.

Con los resultados consolidados y estandarizados, éstos están listos para que el usuario pueda realizar la clasificación de falsos positivos, por lo que se requiere tener un módulo que imprima los resultados en la línea de comando, donde en cada fila se visualiza un índice con la especificación de la alerta encontrada, y quede a la espera de la respuesta del usuario: el módulo **Analyzer**. Este módulo espera que el usuario responda con un listado de índices separados por espacio de las vulnerabilidades que el analista evaluó y consideró como falsos positivos, con lo que dichas alertas quedan identificadas como falsos positivos en la base de datos.

Con los resultados ya clasificados por el usuario, se debe generar un reporte con las alertas encontradas por las herramientas de la plataforma, para lo cual se programó el módulo **Reporter**, el cual exporta la base de datos a un archivo separado por comas (CSV) y se utiliza como archivo de entrada para el *Dashboard*.

El *Dashboard* permite visualizar, tanto a los usuarios de tecnología como a los gerentes, el estado de seguridad de su aplicación web en función de la última ejecución, al igual que las ejecuciones históricas. Al tener una visualización clara de cuáles son las vulnerabilidades actuales de la organización, se puede evaluar cuales son las que mayor riesgo presentan a la aplicación y es posible tomar una decisión estratégica para mitigar las vulnerabilidades detectadas. De esta misma manera, éste permite ver el detalle de la URL junto con el parámetro afectado y así se puede dar una resolución expedita ante las detecciones.

Como todos los módulos descritos anteriormente funcionan de manera independiente y tienen una administración centralizada, se requiere de un módulo que administre los módulos **Scanner Masters** de las herramientas de la plataforma, el **Results Merger**, el **Analyzer** y el **Reporter**, para lo cual se programó el módulo **APT2**. Este módulo se encarga de la ejecución coordinada de los distintos módulos necesarios para automatizar el análisis de vulnerabilidades.

El modo de interacción entre el usuario y la plataforma desarrollada se definió para que la forma más simple de interactuar con la herramienta fuera a través de un archivo de configuración, en el cual el usuario debe especificar los campos necesarios para realizar un análisis de vulnerabilidades. Para ello se programó el módulo **User Config Parser**, el cual tiene la capacidad de leer el archivo de configuración y autocompletarlo en el caso de que el usuario haya omitido alguno de los campos.

Como se realizarán constantemente pruebas sobre los ambientes vulnerables, se hizo ne-

cesario crear el módulo **Env** encargado de configurarlos de manera automática. Asimismo, como también se determinó que se debe tener un ambiente individual para la realización del análisis de vulnerabilidades de cada herramienta, se creó el módulo **Env Manager** que administra los ambientes, el cual tiene la capacidad de levantar la cantidad de ambientes que sean necesarios para llevar a cabo el análisis.

Como el objetivo final de la plataforma desarrollada es automatizar los procesos para la realización de análisis de vulnerabilidades, se requirió crear el módulo **APT2 Manager** el cual está encargado de orquestar la interacción entre el usuario, a través del **User Config Parser**, y los módulos necesarios para llevar a cabo el análisis, el cual le entregará los parámetros recibidos a través del usuario, al módulo **Env Manager** y al módulo **APT2**. Este último se encargará de la ejecución de los *scanners*.

Se puede encontrar un diagrama de los módulos explicados en esta sección en el *Capítulo 3* de este trabajo.

2.7. Validación de la solución propuesta

La investigación que se realizó en este trabajo de memoria fue de tipo experimental, verificándose en base a la comprobación experimental en ambientes vulnerables. Como no es suficiente la validación a través de los ambientes vulnerables mencionados con anterioridad, ya que estos pueden no ser representativos de la realidad de las aplicaciones web en producción, es necesario realizar la validación de la solución mediante su ejecución en una aplicación web en producción. Por esta razón se probó la herramienta producto de este trabajo de memoria en la aplicación web de Fintual, la cual se desarrolla en un ambiente de integración continua y se integra la solución en dicho ambiente. En el *Capítulo Resultados* se ahondará en los detalles de la ejecución de la herramienta.

2.7.1. Métricas

Cada herramienta entrega una clasificación de riesgo para cada vulnerabilidad dentro de su capacidad de detección y la clasificación común entre las herramientas resultó ser la evaluación del riesgo como riesgo alto, medio, bajo o informacional.

Los resultados de dos de los tres *scanners* incorporan una clasificación de riesgo con respecto a la alerta encontrada, y su clasificación es la establecida en este trabajo. Para el *scanner* que no incorpora esta clasificación en sus resultados (Wapiti), se debió crear un diccionario que manualmente mapeó las vulnerabilidades con el riesgo, manteniendo la consistencia con los criterios utilizados por los otros *scanners* y los criterios de la *Vulnerability Rating Taxonomy* (VRT) establecidos por Bugcrowd [22].

2.7.2. Verificación de detección básica

El Control de sanidad, más conocido en el rubro como *Sanity check* se realiza específicamente con ambientes vulnerables donde se conoce la existencia de las vulnerabilidades presentes en el sistema. Se debe tener presente que el hallar vulnerabilidades en este tipo de sistemas vulnerables no necesariamente implica que la solución sirva en un ambiente real.

Si no las logra encontrar, por otra parte, implica que la herramienta propuesta debe ser corregida.

2.7.3. Fintual: validación con una aplicación web en producción

Como se mencionó con anterioridad, se puso a prueba la herramienta en un ambiente real. Se probó la aplicación web de Fintual en dos ambientes distintos. El primero fue local, donde se levantó la aplicación de Fintual en la aplicación en la máquina local, y en la segunda se utilizaron las *review apps* de Heroku [23] la cual levanta una máquina en los servidores de este proveedor.

2.8. Integración de la herramienta desarrollada en el ambiente de desarrollo de Fintual

La aplicación web está desarrollada en el framework Ruby on Rails la cual está alojada en los servidores de Heroku y tienen el código fuente en un repositorio en Github, el cual está conectado con CircleCI, el proveedor de servicios de integración continua, en el cual ejecutan una pila de pruebas para cada solicitud de revisión de código de los desarrolladores (*pull request*), lo que asegura que su código posea un estándar mínimo de validación antes de lanzar una nueva versión a producción.

Adicionalmente, utilizan las denominadas *review apps* de Heroku, las cuales son aplicaciones símiles a la de producción, construidas en base a las versiones del código de una rama de desarrollo específica, con una base de datos del ambiente de desarrollo, y tienen el propósito de poder visualizar y materializar los cambios que se han realizado en el código en un ambiente idéntico en el que se aloja la aplicación de producción.

Para probar las distintas versiones del código a medida que van surgiendo, utilizan el ambiente de desarrollo e implementan pruebas en RSpec [24], lo que permite poner a prueba diferentes contextos y estados que la aplicación puede poseer para algún momento determinado. Para ello existe una prueba RSpec para cada una de las funcionalidades que ofrece la aplicación. Es por este motivo que se deben implementar pruebas en RSpec para el análisis de vulnerabilidades debido a los distintos contextos y estados que existen dentro de la aplicación de Fintual. Es importante destacar este último punto, pues si existe un contexto que no se puso a prueba, no se estaría verificando la aplicación en su totalidad y el análisis podría pasar por alto alguna vulnerabilidad.

Dentro de las opciones que otorga el ambiente de integración continua están las ejecuciones de las pruebas en un ambiente denominado como *machine executor* y *Docker executor*, los cuales se diferencian por la tecnología que utilizan como medio para ejecutar las pruebas. Es decir, en el primero se utilizan máquinas virtualizadas y en el segundo imágenes de Docker. El ejecutor de Docker, por motivos de seguridad, no permite la comunicación entre los distintos contenedores, característica crucial debido a la necesaria interacción entre los contenedores para realizar el análisis y obtener los resultados. Es por esto que se debe utilizar el ejecutor de máquinas virtuales, lo que exige utilizar Docker para el montaje de la aplicación web de Fintual. Esto último se complementa con la plataforma desarrollada en este trabajo, dado

que Docker es una de sus tecnologías base.

2.9. Integración Módulo Adicional

Uno de los principales focos de este trabajo de memoria es la construcción de una herramienta con una arquitectura modular y, de esta manera, que permita agregar otras herramientas de *pentesting* de manera simple y directa, sin la necesidad de realizar modificaciones estructurales al código de la plataforma desarrollada.

Al realizar un diseño modular, se tienen clases genéricas con las funcionalidades básicas que debe realizar cada módulo. Como los módulos genéricos ya están desarrollados, agregar una herramienta adicional, implica crear las clases de Python que heredan de los módulos genéricos y por lo tanto las funcionalidades básicas ya están desarrolladas, con lo que es necesario ver los métodos que son particulares para cada herramienta.

A continuación, se detallan los métodos de los módulos que se deben implementar para lograr la integración de una herramienta de *pentesting* adicional en la plataforma.

1. **Scanner**. En primera instancia se requiere tener la herramienta a integrar en un contenedor de Docker, requerimiento básico para compatibilizar con la plataforma desarrollada. Dado el contexto actual y la migración de las tecnologías utilizadas al uso de contenedores, este paso puede ya estar desarrollado en las herramientas más conocidas.
2. En el módulo **Scanner Master** se debe implementar un único método para casos particulares de las herramientas. En los casos de ZAP y Wapiti, se agregó una línea de código adicional en cada caso para tratar con estas particularidades, como establecer el puerto en que se ejecutará el servicio de ZAP o si se debe ejecutar el módulo aparte de la Cookie si se realizará un análisis autentificado para Wapiti. En el caso de W3AF, no se realizaron modificaciones a este método, por lo que la cantidad de desarrollo a realizar dependerá de las particularidades de cada herramienta.
3. El método para generar el comando de ejecución en el módulo **Argument Parser** es uno de los pasos que implican mayor desarrollo, justamente debido a que cada herramienta se ejecuta de manera particular, por lo que se requiere entender el funcionamiento de la herramienta e identificar qué parámetros se deben administrar para su correcta ejecución.
4. Para el módulo **Results Parser** es necesario implementar dos métodos. El primero es el que genera el objeto de estructura estandarizada a partir de la estructura de los resultados obtenidos de la herramienta nueva y, el segundo, es el que filtra los campos de interés.
5. Finalmente, se debe incluir el **Scanner Master** de la herramienta a integrar en la variable de tipo lista en el módulo **APT2 Manager**, la cual almacena los **Scanner Masters** de los *scanners* implementados en la plataforma.

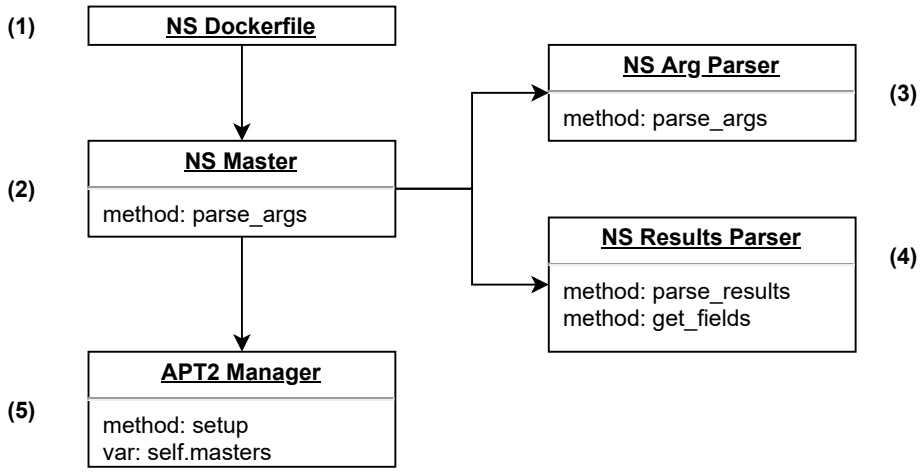

Figura 2.1: Esquema de integración de una herramienta de análisis de vulnerabilidades adicional a la plataforma.

Capítulo 3

Implementación de la Solución

En este capítulo se explica el funcionamiento de la solución propuesta, se detallan los módulos que componen la plataforma y se describe el flujo de información de la herramienta desarrollada con los distintos agentes involucrados.

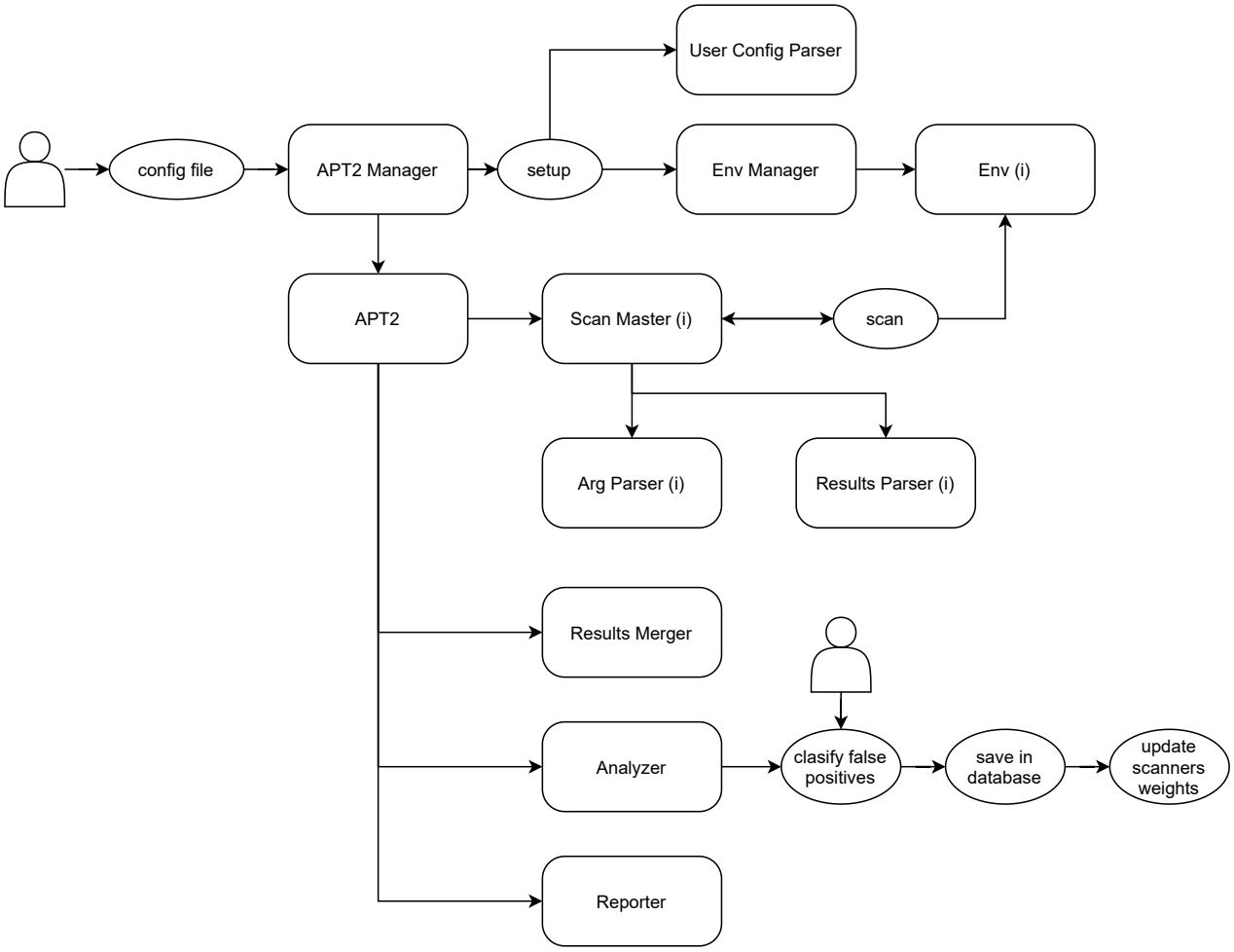

Figura 3.1: Diagrama modular de la solución desarrollada.

3.1. Módulo User Config Parser: Configuración del usuario

La forma de interacción del usuario con la plataforma es a través de un archivo de configuración que debe completar el usuario. En este archivo se pueden especificar diversas variables como las de ejecutar la plataforma sobre un ambiente vulnerable o si se desea realizar el análisis mediante un usuario autenticado. Para el segundo caso, se requiere completar la información mínima para realizar dicha autenticación, como por ejemplo, la ruta de la aplicación en que se realiza el ingreso, junto con las credenciales de usuario y contraseña. En algunos casos, es necesario especificar los nombres o *tags* HTML de los campos de entrada del formulario de ingreso.

En este mismo archivo también se especifican las variables para ejecutar los *scanners*, como la URL a la cual se realizará el análisis, el puerto en que se encuentra la aplicación web, los módulos de las vulnerabilidades a ejecutar y la ruta compartida entre los contenedores de las herramientas con la máquina del usuario donde se almacenarán los resultados obtenidos.

Este módulo se encarga de colocar las variables por defecto en el caso que el usuario no las especifique. Por ejemplo, si se omite el campo de *target*, viene por defecto el valor *localhost*, para el puerto, el valor por defecto es 8080 y, para los módulos de las vulnerabilidades a ejecutar, se ejecutan todas las posibles dentro de las herramientas de la plataforma.

3.2. Módulo Env: Ambientes

Se creó una clase base en Python, la cual posee los atributos necesarios para que un ambiente pueda existir, como la URL, el puerto, los nombres de las imágenes de Docker. Además incluye los métodos para generar el contenedor y preparar las configuraciones propias del ambiente, junto con la creación de usuarios y el método para ejecutar o levantar el ambiente a través del comando generado a partir de sus configuraciones establecidas.

Para cada ambiente de prueba utilizado en este trabajo se creó una clase que hereda de la clase base del ambiente. Esto facilita la integración de nuevos ambientes debido a que solamente se deben implementar los métodos para configurarlo adecuadamente y no hay necesidad de implementar el levantamiento particular de este ya que la clase base ya posee el método para ejecutarlo.

3.3. Módulo Env Manager: Administrador de Ambientes

El módulo **Env Manager** es el administrador de ambientes. La necesidad de este módulo surge cuando se utiliza más de un *scanner* para el análisis de vulnerabilidades, ya que es importante utilizar distintos ambientes de prueba y como las herramientas de pentesting generalmente son intensivas en recursos, por lo que un único *scanner* puede dejar sin capacidad de respuesta al ambiente en análisis. Por esta razón es que se debe utilizar un ambiente para cada herramienta, la cual es la funcionalidad del **Env Manager**. Este módulo recibe el módulo **Env**, el puerto inicial en el que quedarán configurados los ambientes, la cantidad de ambientes a levantar, el cual equivale a la cantidad de *scanners* a utilizar y, finalmente, las

variables de configuración para el ambiente, propiamente tal.

Este módulo posee los métodos para ejecutar los ambientes en paralelo, asignando el puerto inicial al primer ambiente, sumando uno al puerto para cada uno de los siguientes ambientes. También tiene el método para configurar los ambientes, el cual invoca el método de configuración de cada ambiente particular, en los cuales se tiene la lógica para preparar el ambiente y crear los usuarios de ser necesario.

Cabe destacar que para la integración de nuevos ambientes no se debe implementar ningún método en este módulo ya que toda la lógica está descrita en el administrador de ambientes genérico.

3.4. Módulo APT2 Manager: Administrador de APT2

El **APT2 Manager (APT2M)** es el administrador del módulo **APT2**, el cual se encarga de la interacción inicial del usuario la que consta de la lectura del archivo de configuración y configura las variables de entorno necesarias para ejecutar los *scanners*. Además, el **APT2M** es el encargado de configurar los ambientes en los casos que corresponda, entre otras variables.

El **APT2M** posee una variable tipo lista en la cual se almacenan los **Scanner Masters** que se utilizarán para el siguiente escaneo. Además, contiene una variable de tipo diccionario con los administradores de los ambientes a poner en prueba. A partir del archivo de configuración, se determina cuál ambiente se utilizará en el análisis, con lo que el módulo **APT2M** le indica al **Env Manager** que inicie el ambiente y quede listo para el análisis. La preparación del ambiente se detallará en la subsección relacionada a dicho módulo.

Luego, se inicia la preparación del ambiente de trabajo, proceso en el cual se carga la bases de datos y, si esta no existe, la crea. Posteriormente, se invoca el módulo **User Config Parser** donde se cargan las variables de entorno entregadas por el usuario y se levantan los ambientes a analizar mediante el **Env Manager**. Finalmente, se invoca el módulo **APT2** con las variables necesarias para realizar el análisis de vulnerabilidades del ambiente determinado por el usuario.

A modo de ejemplo, el usuario puede especificar ejecutar los *scanners* en uno de los ambientes vulnerables. En este ambiente vulnerable es posible autenticarse y acceder a lugares a los cuales es imposible llegar sin esta autenticación. El usuario debe especificar los campos de autenticación en el archivo de configuración y el **APT2M** invoca las funciones para levantar el ambiente, se configuran las variables del objetivo y se invoca el módulo **APT2**.

3.5. Módulo APT2: Automated Penetration Testing Tool

El módulo **APT2** es el encargado de orquestar los procesos para llevar a cabo el análisis de vulnerabilidades del objetivo propuesto. Este está al mando de los módulos: **Scanner Masters**, **Results Merger**, **Analyzer**, y **Reporter**.

El método de escaneo automático llama a los **Scanner Masters** a que configuren el ambiente en que se ejecutarán, configura el **Results Merger** y ejecuta el método *run_scans*.

Este método le indica a cada maestro que ejecute su *scanner* e invoca el método de espera, el cual esperará hasta que concluyan todos los *scanners* de realizar su debido análisis. Luego se invoca el método que estandariza los resultados, el cual le indica a cada maestro que utilice su respectivo **Results Parser** para estandarizar los resultados. Finalmente se invoca el método del módulo **Results Merger** que unifica los resultados de las distintas herramientas.

Al tener los resultados estandarizados y unificados, se ejecuta el módulo del **Analyzer**, el que interactúa con el usuario para realizar el análisis de falsos positivos de los resultados obtenidos.

Posteriormente, se invoca el módulo **Reporter** que coloca los resultados históricos en un *Dashboard* que permite la evaluación de los resultados.

Lo último que hace el módulo **APT2** es ejecutar el método de limpieza de los ambientes, el cual le indica a los **Scanner Masters** que deben limpiar su entorno, lo que incluye eliminar los archivos temporales utilizados para almacenar los resultados, como también los contenedores levantados para realizar el análisis propiamente tal.

3.6. Módulo Scanner Master: Gestor de herramientas de detección de vulnerabilidades

El módulo **Scanner Master** es el encargado de dar las órdenes a los submódulos necesarios para llevar a cabo el análisis. Este interactúa con los módulos **Argument Parser** y **Results Parser** para primero construir el comando de ejecución y, una vez concluido el análisis, se obtienen los resultados y se estructuran en un formato en común entre todos los *scanners*.

La inicialización de este módulo requiere de la ruta del archivo Dockerfile, en el caso de que se requiera construir la imagen, o el nombre de la imagen del contenedor a utilizar para ejecutar el análisis si ésta ya está en la máquina local, por lo que no es necesario reconstruirla. También recibe los módulos **Argument Parser** y **Results Parser** específicos para cada herramienta, los cuales son almacenados en los atributos de este módulo.

Para la configuración del módulo, este recibe tres variables de tipo diccionario, en donde en la primera están los datos relevantes a la ejecución del *scanner*, como la ruta compartida, la URL, el puerto, y los módulos a ejecutar, por mencionar algunos. En el segundo diccionario están las variables del ambiente a analizar, el cual puede contener rutas o sufijos de la aplicación que se desean ignorar en el análisis. Por ejemplo, en el ambiente Mutillidae hay rutas en la aplicación que eliminan la base de datos o reinician la aplicación a su estado inicial, lo que arruinaría el análisis en cuestión. Esto también sirve para definir el alcance del análisis, por ejemplo si por algún motivo no se desea analizar una sección de la aplicación. Finalmente, el tercer diccionario contiene las variables de autenticación en la aplicación. Estos dos últimos diccionarios son utilizados exclusivamente por el módulo **Argument Parser** para construir el comando de ejecución.

Luego se invoca el **Argument Parser**, el cual construye el comando para ejecutar el *scanner* con las variables establecidas. Existe un punto de verificación para continuar el análisis, esto es, para el caso en que el *scanner* no posea los módulos que el usuario indique

ejecutar, el *scanner* no se ejecutará. Para finalizar la configuración del módulo, este construye la imagen de la herramienta si el usuario lo ha indicado.

Cuando el módulo **APT2** llama a ejecutar los *scanners*, posteriormente a la configuración de cada uno, el **Scanner Master** crea un subproceso que ejecuta el comando generado en los pasos anteriores. El proceso se almacena en un atributo del módulo para cuando el módulo **APT2** indique esperar a que las herramientas terminen sus análisis, el maestro tenga el proceso a su disposición para ejecutar la espera.

Al término de la ejecución de la herramienta, el **APT2** indicará que se limpien los ambientes, como se describió en la sección anterior y el módulo **Scanner Master** delega esta función a los submódulos **Argument Parser** y **Results Parser**.

3.6.1. Módulo Argument Parser: Gestor de argumentos de ejecución

Este módulo tiene la función de generar el comando que ejecutará el *scanner* con las configuraciones especificadas en un contenedor de Docker.

Dado que para cada *scanner* hay que indicarle que ejecute un módulo de vulnerabilidad de manera específica a la herramienta, cada **Argument Parser** posee un atributo de tipo diccionario para el cual las llaves son los nombres de los módulos estandarizados y los valores son la traducción de las vulnerabilidades estandarizadas al formato en que el *scanner* las puede ejecutar.

Para el caso en que se le indica a una herramienta ejecutar un módulo que no es capaz de realizar, debido a que no existe su implementación, el **Argument Parser** responde con *None* y el **Scanner Master** detecta este parámetro y concluye el proceso en ese mismo instante.

3.6.2. Scanner: Herramienta de detección de vulnerabilidades

Cada herramienta de análisis de vulnerabilidades debe estar asociado a una imagen en Docker, ya sea que esté en algún repositorio y se pueda descargar desde el mismo o se tenga de manera local. Para este trabajo de memoria se tiene un archivo Dockerfile para cada *scanner*, los cuales construyen la imagen para crear los contenedores respectivos.

Además, se desarrolló un *script* que automatiza la ejecución del *scanner* a partir de los parámetros entregados por el usuario en un inicio y, a continuación, se detalla el funcionamiento de éstos para cada uno de los *scanners*.

1. **Zed Attack Proxy (ZAP)**. El Dockerfile de ZAP descarga la imagen del repositorio de Dockerhub de los mismos creadores de la herramienta, OWASP. Realiza una actualización de las dependencias e instala Netcat [25], una herramienta para manipular los puertos TCP/UDP de un sistema e instala las dependencias para ejecutar la API junto con *Beautifulsoup4* para el manejo de contenido HTML raíz de las peticiones HTTP que se podrían realizar. Finalmente ejecuta el *script* escrito en Bash *entrypoint.sh* el cual levanta el servicio de ZAP, con netcat verifica que se haya inicializado completamente y ejecuta el *script* para realizar el análisis de vulnerabilidades.

   El *script* para realizar el análisis se conecta a través de la API al servicio de ZAP ini-

cializado, el cual configura las variables del entorno de la herramienta, como la URL del objetivo, los módulos a ejecutar, los parámetros de autenticación, entre otros y realiza el análisis propiamente tal. El análisis a realizar consta de los procesos de encontrar todas las URL posibles de la aplicación, ejecutar un análisis pasivo el cual revisa únicamente por configuraciones incorrectas y luego ejecuta el análisis activo el cual utiliza los *payloads* acorde a los módulos especificados a ejecutar. Al finalizar el análisis, se escriben los resultados en un archivo JSON y se almacena en la ruta compartida entre el contenedor y la máquina local.

Para la realización de un análisis autentificado, se proporcionan los nombres de los campos de ingreso del formulario, junto con las credenciales y el *script* se encarga de pasárselas a ZAP a través de su API. Con esto es suficiente para que ZAP realice el análisis autentificado.

2. **W3AF**. Para esta herramienta, si bien existe la imagen del contenedor en Dockerhub, en la práctica, no se logró compatibilizar con la herramienta desarrollada en este trabajo. Es por esto que se generó el contenedor desde la base, utilizando el mismo sistema operativo y librerías para generar el ambiente requerido por la herramienta. Para la ejecución de esta herramienta, el usuario debe ingresar manualmente las configuraciones necesarias para realizar el análisis a través de la línea de comando, como también está la posibilidad de crear un archivo con las configuraciones y el *scanner* se puede ejecutar a partir de este archivo de configuración. Es por esto que se programó un *script* que genera el archivo de configuración a partir de las configuraciones del usuario y se le proporciona al *scanner* como argumento de ejecución, de forma automática. El encargado de utilizar el *script* de configuración es el **Argument Parser** de W3AF.

   Para la realización de un análisis autentificado, se requiere la misma información que en el *scanner* anterior y se especifica en el archivo de configuración de la herramienta que se utilizará el módulo de autenticación y se deben especificar los nombres de los campos del formulario de ingreso y las credenciales.

3. **Wapiti**. El Dockerfile de Wapiti descarga la imagen del repositorio de Dockerhub. Se utilizaron varias imágenes de este repositorio y resultó que la imagen con mejor reputación, según el sitio de descarga, es la que funciona correctamente.

   La ejecución de Wapiti se realiza completamente a través de la línea de comando. El caso excepcional es para un análisis de vulnerabilidades autentificado el cual se requiere la ejecución del módulo *wapiti-getcookie*, el cual realiza las peticiones a la aplicación con las credenciales de ingreso y almacena las *Cookies* en un archivo de salida. Este archivo con las *Cookies* es entregado a Wapiti para la ejecución del análisis autentificado.

3.6.3. Módulo Results Parser: Gestor de resultados

Nuevamente, como las herramientas tienen comportamientos distintos, la estructura de los resultados obtenidos por cada una también es distinta, por lo que existe la necesidad de un módulo que determine la estructura de la información de la herramienta y la modifique a una estructura estándar, de esta manera se pueden centralizar los resultados.

Cada herramienta posee su propio **Results Parser** y se programa de tal forma de buscar en los resultados los campos de interés. Los campos de interés en común establecidos para este trabajo son: el nombre de la vulnerabilidad, su severidad clasificada en las categorías

Alto, Medio, Bajo, Información, la URL en que fue encontrada, el parámetro específico que se utilizó para detectarla y la herramienta que la encontró.

3.7. Módulo Analyzer: Ingreso de falsos positivos

Una vez concluidos los *scanners*, se invoca el módulo **Analyzer** e imprime en la consola las alertas encontradas por los *scanners* y este le pide al usuario indicar manualmente los índices de las alertas que posterior a su evaluación, considera que son falsos positivos. El listado de índices de las alertas se entrega como un listado de números separados por espacios.

De comienzo, el módulo recibe el nombre del archivo unificado y normalizado y un identificador del *scan* que se utiliza para almacenar los resultados en la base de datos.

El método analizar resultados crea la fila en la tabla **Scan** con el identificador si es que no existe. Luego enumera los resultados obtenidos y queda a la espera del usuario a que explicite los números de las alertas que considera falsos positivos separados por espacios, con lo que actualizará el campo falso positivo de las alertas que correspondan y son ingresadas a su tabla respectiva. Finalmente se recalcula el peso de las vulnerabilidades de cada *scanner* con el método actualizar pesos (*update weights*) del módulo.

El método *update weights* utiliza la siguiente ecuación para calcular los pesos de todas las vulnerabilidades (v) detectadas para cada *scanner* (s):

$$W_s v = 1 - \frac{FP(s,v)}{T(s,v)} \tag{3.1}$$

donde FP(s,v) corresponde a los falsos positivos de las vulnerabilidades de tipo v detectadas por el scanner s y T(s,v) corresponde al total de vulnerabilidades del tipo v del *scanner* s. Esta forma de calcular el peso de las vulnerabilidades permite que a medida que se clasifiquen mayor cantidad de falsos positivos en un *scanner* para un tipo de vulnerabilidad, esta tenga menor relevancia al momento de presentar los resultados.

Una variante de esta ecuación para cuantificar los positivos verdaderos y que los *scanners* con mejor detección tengan mayor relevancia dentro de los resultados es la siguiente:

$$W_s v = \left(1 - \frac{FP(s,v)}{T(s,v)}\right) x \left(1 + \frac{TP(s,v)}{T(s,v)}\right) \tag{3.2}$$

donde TP(s,v) corresponde a las vulnerabilidades clasificadas como verdaderos positivos de las vulnerabilidades de tipo v detectadas por el scanner s. A medida que un *scanner* posea mayor cantidad de verdaderos positivos, el peso de la vulnerabilidad para ese *scanner* será mayor, ya que multiplica por un factor mayor a 1.

En este trabajo no se estudió la factibilidad de la función de actualización de los pesos, por lo que se desarrolló este método para que en un trabajo futuro se pueda implementar dicha funcionalidad.

3.8. Módulo Reporter: Visualización de datos

El módulo **Reporter** tiene la función de exportar la base de datos a un archivo separado por comas (CSV) el cual es utilizado en el *Dashboard* y permite presentar los resultados de los análisis realizados históricamente, en el cual, en primera instancia, se pueden visualizar las aplicaciones a las cuales se han realizado análisis de vulnerabilidades. Al entrar a una aplicación, se presentan los diferentes análisis que se han realizado a esa aplicación en particular y al entrar al detalle de uno, se pueden visualizar las vulnerabilidades detectadas por la plataforma, especificando la información descrita en la *subsección* ***Results Parser***.

En los Gráficos 2, 3 y 4 se puede observar el *Dashboard* en el cual se visualizan las cantidades de alertas de riesgo Alto, Medio y Bajo, junto con un gráfico que indica las proporciones de alertas detectadas por cada *scanner* y una tabla con el detalle de las vulnerabilidades detectadas. Al hacer *click* sobre una fila, se despliega el detalle y se pueden visualizar las URL's asociadas a las alertas.

Adicionalmente, en los Gráficos 5, 6, 7, se puede observar una vista histórica del *Dashboard*, en cual despliega el historial de ejecuciones a lo largo del tiempo. Para esta vista también se puede filtrar por riesgo, *scanner* y por un *scan* o ejecución particular. En este ejemplo se muestra la vulnerabilidad *Remote OS Command Injection* y explicita que el *scanner* ZAP fue el que las encontró.

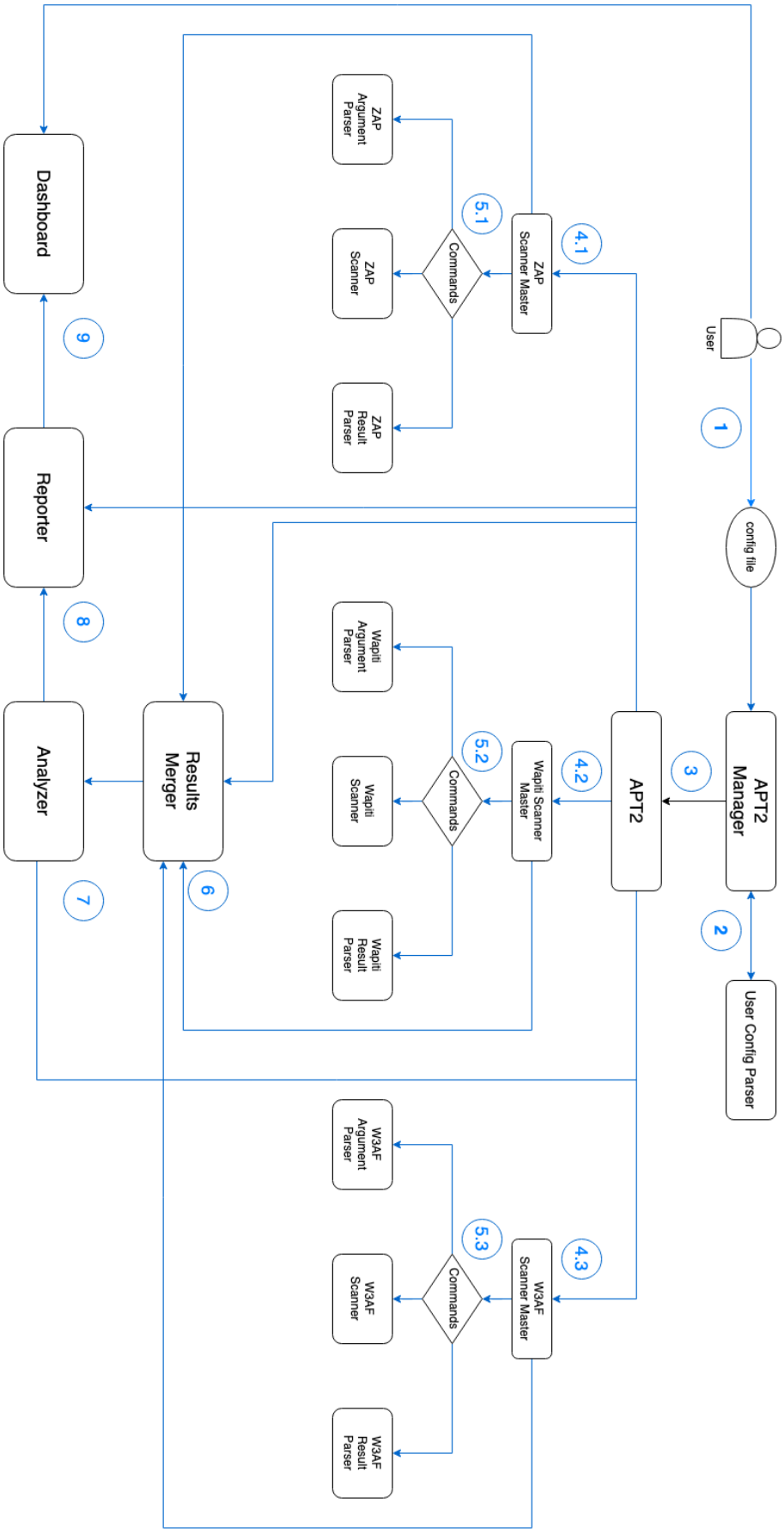

Figura 3.2: Diagrama modular de la solución desarrollada con el detalle de los *scanners* integrados.

Capítulo 4

Resultados

Para el análisis de resultados se consideraron las vulnerabilidades de riesgo alto y medio y no se consideró el análisis de vulnerabilidades de riesgo bajo e informacional.

En algunos casos, de los resultados obtenidos por los *scanners*, se encontraron distintos tipos de la misma vulnerabilidad, por lo que en los resultados de este capítulo se agruparon según su clasificación. Un ejemplo de esto es la vulnerabilidad *Cross Site Scripting* que en las documentaciones se explicitan distintos tipos de ésta, como el *Persistent/Stored Cross Site Scripting* y el *Reflected Cross Site Scripting* y para el método de conteo, ambos tipos se clasificaron como *Cross Site Scripting*.

Para la comparación y evaluación de los resultados de los *scanners* en los distintos ambientes vulnerables, se utilizó la métrica de precisión de detección, la cual se calculó dividiendo la cantidad de vulnerabilidades detectadas por la cantidad de vulnerabilidades presentes de un riesgo específico. Es importante considerar que esta métrica posee relevancia solamente en el contexto de estas pruebas básicas que buscan comprobar la correcta ejecución de la plataforma y no se pueden generalizar al universo de aplicaciones web.

4.1. OWASP Mutillidae II - PHP

En la Tabla 4.1 se desglosa la documentación de vulnerabilidades presentes en la aplicación web Mutillidae [26].

Nombre Vulnerabilidad	Riesgo
Cross Site Scripting	Alto
Insecure File Upload	Alto
OS Command Injection	Alto
Path Traversal	Alto
Privilege Escalation	Alto
Remote File Inclusion	Alto
SQL Injection	Alto
Unvalidated Redirects	Alto
Application Exception	Medio
Buffer Overflow	Medio
Clickjacking	Medio
Denial of Service	Medio
Directory Browsing	Medio
Local File Inclusion	Medio

Tabla 4.1: Vulnerabilidades documentadas en Mutillidae.

En las Tablas 4.2 y 4.3 se pueden observar las detecciones únicas de vulnerabilidades de riesgo alto y medio según el *scanner* que las encontró.

Nombre Vulnerabilidad	ZAP	Wapiti	W3AF
Cross Site Scripting	50	0	3
OS Command Injection	26	0	0
Path Traversal	19	0	0
Remote File Inclusion	1	0	0
SQL Injection	5	12	2
Unvalidated Redirect	3	0	0

Tabla 4.2: Vulnerabilidades de riesgo alto detectadas en el ambiente Mutillidae.

Nombre Vulnerabilidad	ZAP	Wapiti	W3AF
Application Exception	92	0	4
Clickjacking	337	0	0
Cross-Domain Misconfiguration	2	7	7
Directory Browsing	27	0	0
Vulnerable JS Library	2	0	0

Tabla 4.3: Vulnerabilidades de riesgo medio detectadas en el ambiente Mutillidae.

En la Tabla 4.4 se visualiza el conteo de vulnerabilidades según su riesgo y el conteo de las detecciones para cada *scanner*.

Riesgo	Mutillidae	ZAP	Wapiti	W3AF
Alto	8	6	1	2
Medio	6	3	0	1

Tabla 4.4: Conteo de vulnerabilidades según el riesgo y *scanner* en Mutillidae.

En la Tabla 4.5 se puede observar la precisión de detección para cada *scanner* según el riesgo de las vulnerabilidades, además de la precisión global del ambiente, que se calculó como la razón entre el total de tipos vulnerabilidades únicas detectadas y el total de vulnerabilidades documentadas.

Scanner	Precisión Riesgo Alto	Precisión Riesgo Medio	Precisión Global
ZAP	75 %	50 %	64 %
Wapiti	25 %	16 %	21 %
W3AF	12.5 %	0	7 %
Total	75 %	50 %	64 %

Tabla 4.5: Precisión de detección de vulnerabilidades de cada *scanner* para los riesgos Alto, Medio y la precisión global en Mutillidae.

Se puede observar en los resultados de las tablas que el *scanner* Wapiti no fue capaz de detectar la mayoría de las vulnerabilidades documentadas y esto se debe a que la ejecución de éste producía que se reiniciara la aplicación web de Mutillidae, con lo que no lograba completar sus ejecuciones de manera exitosa. Para esto se ejecutaron los módulos de vulnerabilidades por separado y se identificó que el módulo *Code Execution*, el cual busca ejecutar código en el servidor de la aplicación, causaba dicho efecto. Por otra parte, al ejecutar los módulos sin este último, la aplicación todavía presentó este comportamiento, con excepción del módulo de la vulnerabilidad *SQL Injection*. En este trabajo no se pudo determinar la fuente del problema y, como no afecta la funcionalidad general de la plataforma, queda propuesto como una posible mejora para un trabajo futuro.

En la aplicación Mutillidae se detectaron 9 tipos de vulnerabilidades de las 14 documentadas, lo que se traduce en un 64 % de precisión de detección. Es importante destacar que ZAP fue capaz de detectar las 9 vulnerabilidades encontradas por la plataforma, aún así, los otros *scanners* detectaron el mismo tipo de vulnerabilidad en distintas rutas y parámetros, por lo que la composición de los tres *scanners* encontraron una mayor cantidad de vulnerabilidades de cada tipo.

Las dos vulnerabilidades de riesgo alto que no detectó la plataforma fueron *Privilege Escalation* e *Insecure File Upload*. Las vulnerabilidades de riesgo medio que no detectó la plataforma fueron *Buffer Overflow*, *Denial of Service* y *Local File Inclusion*. La plataforma logró detectar las vulnerabilidades *Vulnerable JS Library* y *Cross-Domain Misconfiguration* las cuales no están documentadas en el listado de vulnerabilidades de Mutillidae.

Para el ambiente vulnerable Mutillidae, ZAP fue el *scanner* con la mayor cobertura de vulnerabilidades, con una precisión del 64 %, seguido por Wapiti con un 21 % y finalmente W3AF con un 7 % de precisión. La precisión global de la plataforma en este ambiente fue de un 64 %, idéntica a la del mejor *scanner*.

4.2. Google Gruyere - Django

En la Tabla 4.6 se desglosa la documentación de vulnerabilidades presentes en la aplicación web Gruyere [27].

Nombre Vulnerabilidad	Riesgo
Code Execution	Alto
Cross-Site Request Forgery	Alto
Cross Site Scripting	Alto
Path Traversal	Alto
Privilege Escalation	Alto
SQL Injection	Alto
Application Exception	Medio
Buffer Overflow	Medio
Cookie Manipulation	Medio
Denial of Service	Medio

Tabla 4.6: Vulnerabilidades documentadas en Gruyere.

En las Tablas 4.7 y 4.8 se pueden observar las detecciones únicas de vulnerabilidades de riesgo alto y medio según el *scanner* que las encontró.

Nombre Vulnerabilidad	ZAP	Wapiti	W3AF
Cross Site Scripting	0	7	12
Insecure File Upload	0	0	1
Memcache Injection	0	0	3
SQL Injection	0	3	0

Tabla 4.7: Vulnerabilidades de riesgo alto detectadas en el ambiente Gruyere.

Nombre Vulnerabilidad	ZAP	Wapiti	W3AF
Application Exception	0	0	2
Format String Error	2	0	0
Local File Inclusion	0	0	2
Response Split	0	0	7
X-Frame-Options Header Not Set	43	0	0

Tabla 4.8: Vulnerabilidades de riesgo medio detectadas en el ambiente Gruyere.

En la Tabla 4.9 se visualiza el conteo de vulnerabilidades según el riesgo para cada *scanner*.

Riesgo	Gruyere	ZAP	Wapiti	W3AF
Alto	6	0	2	1
Medio	4	0	0	1

Tabla 4.9: Conteo de vulnerabilidades según el riesgo y *scanner* en Gruyere.

En la Tabla 4.10 se puede observar la precisión de detección para cada *scanner* según el riesgo de las vulnerabilidades y la precisión global de la plataforma sobre este ambiente.

Scanner	Precisión Riesgo Alto	Precisión Riesgo Medio	Precisión Global
ZAP	0 %	0 %	0 %
Wapiti	16 %	25 %	20 %
W3AF	33 %	25 %	20 %
Total	33 %	25 %	30 %

Tabla 4.10: Precisión de detección de vulnerabilidades de cada *scanner* para los riesgos Alto, Medio y la precisión global en Gruyere.

El *scanner* ZAP no detectó ninguna vulnerabilidad de riesgo alto y tampoco detectó vulnerabilidades de riesgo medio documentadas. Sí detectó dos vulnerabilidades de riesgo medio: *X-Frame-Options Header Not Set* y *Format String Error* las cuales no estaban documentadas.

El *scanner* Wapiti detectó dos vulnerabilidades de riesgo alto y no detectó ninguna vulnerabilidad de riesgo medio.

W3AF fue el *scanner* con mayor detección en este ambiente, el cual detectó dos de las vulnerabilidades de riesgo alto y además, detectó dos vulnerabilidades de riesgo alto no documentadas: *Insecure File Upload* y *Memcache Injection*.

Las vulnerabilidades de riesgo alto no detectadas fueron *Code Execution*, *Cross-Site Request Forgery*, *Path Traversal*, *Privilege Escalation*.

Para este laboratorio, se detectaron 3 de las 10 vulnerabilidades documentadas, con lo que se obtuvo una precisión de detección de la plataforma del 30 %, siendo W3AF el *scanner* con mayor detección de vulnerabilidades de riesgo alto y ZAP el *scanner* con la mínima cantidad de detecciones.

4.3. OWASP Railsgoat - Ruby on Rails

En la Tabla 4.11 se desglosa la documentación de vulnerabilidades presentes en la aplicación web Railsgoat [28].

Nombre Vulnerabilidad	Riesgo
OS Command Injection	Alto
Cross-Site Request Forgery	Alto
Cross Site Scripting	Alto
SQL Injection	Alto
Unvalidated Redirects	Alto
Insecure Direct Object Reference	Medio

Tabla 4.11: Vulnerabilidades documentadas en Railsgoat.

En las Tablas 4.12 y 4.13 se pueden observar las detecciones únicas de vulnerabilidades de riesgo alto y medio según el *scanner* que las encontró.

Nombre Vulnerabilidad	ZAP	Wapiti	W3AF
Memcache Injection	0	0	1
OS Command Injection	2	0	0
Path Traversal	1	0	0
Server Side Include	5	0	2
SQL Injection	4	2	0

Tabla 4.12: Vulnerabilidades de riesgo alto detectadas en el ambiente Railsgoat.

Nombre Vulnerabilidad	ZAP	Wapiti	W3AF
Application Exception	0	0	2
Clickjacking	33	0	0
Format String Error	2	0	0
Local File Inclusion	0	0	2
Vulnerable JS Library	2	0	0

Tabla 4.13: Vulnerabilidades de riesgo medio detectadas en el ambiente Railsgoat.

En la Tabla 4.14 se visualiza el conteo de vulnerabilidades según el riesgo para el ambiente vulnerable como para cada *scanner*.

Riesgo	Railsgoat	ZAP	Wapiti	W3AF
Alto	5	2	1	1
Medio	1	0	0	0

Tabla 4.14: Conteo de vulnerabilidades según el riesgo y *scanner* en Railsgoat.

En la Tabla 4.15 se puede observar la precisión de detección para cada *scanner* según el riesgo de las vulnerabilidades junto con la precisión global de la plataforma sobre este ambiente.

Scanner	Precisión Riesgo Alto	Precisión Riesgo Medio	Precisión Global
ZAP	40 %	0 %	33 %
Wapiti	20 %	0 %	16 %
W3AF	20 %	0 %	16 %
Total	40 %	0 %	33 %

Tabla 4.15: Precisión de detección de vulnerabilidades de cada *scanner* para los riesgos Alto, Medio y la precisión global de la plataforma en Railsgoat.

A partir de los resultados expuestos en las tablas, el *scanner* Wapiti detectó una única vulnerabilidad correspondiente a *SQL Injection* y no fue capaz de detectar otra vulnerabilidad, mientras que ZAP detectó dos vulnerabilidades de riesgo alto y W3AF una única vulnerabilidad de riesgo alto documentada. Estos últimos dos *scanners* detectaron 8 vulnerabilidades que no están documentadas, como lo son *Memcache Injection*, *Path Traversal*, *Server Side Include* de riesgo alto y *Application Exception*, *Clickjacking*, *Format String Error* y *Local File Inclusion*, *Vulnerable JS Library* de riesgo medio.

En la aplicación Railsgoat se detectaron 3 tipos de vulnerabilidades de las 10 documentadas, lo que se traduce en un 30 % de precisión de detección, sin considerar que se detectaron 3 vulnerabilidades de riesgo alto y 5 de riesgo medio que no están documentadas.

Las vulnerabilidades de riesgo alto que no detectó la plataforma son *Cross-Site Request Forgery*, *Cross Site Scripting* y *Unvalidated Redirects* y la única vulnerabilidad de riesgo medio no fue detectada por ninguno de los *scanners*.

Para el ambiente vulnerable Railsgoat, el *scanner* ZAP obtuvo la mayor precisión de detección de vulnerabilidades, con un 33 %, seguido por Wapiti y W3AF, ambos con 16 % y los tres *scanners* en conjunto lograron detectar 8 vulnerabilidades adicionales que no estaban en la documentación.

4.4. Fintual - Ruby on Rails

En las Tablas 4.16 y 4.17 se pueden observar las detecciones únicas de vulnerabilidades de riesgo alto y medio según el *scanner* que las encontró.

Nombre Vulnerabilidad	ZAP	Wapiti	W3AF
Remote OS Command Injection	3	0	0

Tabla 4.16: Vulnerabilidades de riesgo alto detectadas en el ambiente Fintual.

Nombre Vulnerabilidad	ZAP	Wapiti	W3AF
Application Error Disclosure	2	0	0
Clickjacking	1	0	0
Cross Domain Misconfiguration	70	0	0
Directory Browsing	2	0	0

Tabla 4.17: Vulnerabilidades de riesgo medio detectadas en el ambiente Fintual.

En la Tabla 4.18 se visualiza el conteo de vulnerabilidades detectadas según el riesgo para cada *scanner*.

Riesgo	ZAP	Wapiti	W3AF
Alto	1	0	0
Medio	4	0	0

Tabla 4.18: Conteo de vulnerabilidades según el riesgo y *scanner* en Fintual.

En la aplicación web de Fintual se detectaron 3 vulnerabilidades de alto riesgo, todas correspondientes a *Remote OS Command Injection*, las que fueron encontradas por el *scanner* ZAP. También se encontraron 4 tipos de vulnerabilidades de riesgo medio: *Application Error Disclosure*, *Clickjacking*, *Cross-Domain Misconfiguration* y *Directory Browsing*.

Se puede observar que ZAP fue el único *scanner* que detectó vulnerabilidades en la aplicación de Fintual, mientras que Wapiti y W3AF no lograron detección.

Capítulo 5

Discusión

5.1. Evaluación de los resultados

En esta subsección se analizarán los resultados obtenidos en este trabajo de memoria con el fin de validar los objetivos establecidos.

Primero se analizarán los resultados detallados en la sección anterior, los cuales evidencian el cumplimiento del primer objetivo de: "*Desarrollar una herramienta que detecte vulnerabilidades en aplicaciones web con patrones de ataques preconfigurados. Se validará realizando laboratorios de aplicaciones web con y sin vulnerabilidades, donde se medirá la precisión de detección de las vulnerabilidades, es decir, cuántas vulnerabilidades detecta la herramienta de todas las presentes*".

Para cada *scanner* integrado a la plataforma, se calculó la precisión de detección de vulnerabilidades presentes en cada aplicación web vulnerable y, además, se detallaron las vulnerabilidades encontradas por la plataforma que no estaban en la documentación de vulnerabilidades presentes en los ambientes vulnerables.

En la Tabla 5.1 se presentan las precisiones de detección de cada *scanner* para cada ambiente vulnerable. Se puede observar que ZAP, con un 36 % de precisión, fue la herramienta con mayor detección dentro de las vulnerabilidades documentadas, seguido por W3AF y Wapiti con un 20 % y 13 %, respectivamente.

Ambiente	ZAP	Wapiti	W3AF	Precisión Global
Mutillidae	64 %	21 %	7 %	64 %
Gruyere	0 %	20 %	20 %	30 %
Railsgoat	33 %	16 %	16 %	33 %

Tabla 5.1: Cantidad de vulnerabilidades detectadas por cada *scanner* en los ambientes vulnerables.

En el ambiente vulnerable Gruyere, se observó una mejora entre la precisión de los *scanners* independientes y la precisión de la plataforma que los utiliza en forma consolidada. Mientras tanto, en los dos ambientes vulnerables restantes, la precisión de la plataforma igualó la

precisión del mejor *scanner*, lo que indica que no hubo una mayor detección de diferentes tipos de vulnerabilidades en estos ambientes, esto es, sin considerar las vulnerabilidades no documentadas detectadas.

En la Figura 8 se puede observar que para las distintas vulnerabilidades detectadas por la plataforma, existe una única categoría en la cual los tres *scanners* lograron encontrar al menos una vulnerabilidad: *SQL Injection*. Para el resto de las categorías, un único *scanner* fue capaz de detectar vulnerabilidades de cada tipo, lo que evidencia que los *scanners* detectan distintas vulnerabilidades, lo que es un indicador de la conveniencia de utilizar múltiples *scanners*.

En la aplicación de Fintual se detectaron 3 *Remote Code Execution* (RCE), los cuales fueron validados manualmente como falsos positivos. Para validarlos, se utilizaron distintos *payloads* de RCE [29] y no se logró explotar la vulnerabilidad. Esto último no implica que la vulnerabilidad no esté presente, ya que puede ser el caso que sea posible explotarla con un *payload* más específico, pero que en la práctica no se logró determinar.

A continuación se listan las fortalezas y debilidades de cada *scanner* en base a los resultados obtenidos:

1. **ZAP**. Posee una gran amplitud de detección, siendo capaz de detectar vulnerabilidades de todo el espectro de riesgo. Las vulnerabilidades que no fueron capaces de detectar fueron *Insecure File Upload*, *Local File Inclusion* y *Response Splitting*. Respecto de las primeras dos vulnerabilidades mencionadas, ambas se enfocan en la subida de archivos lo que podría implicar una debilidad en este ámbito, pero también hay que notar que este *scanner* fue el único capaz de detectar *Remote File Inclusion*, por lo que posee una fortaleza en torno a las vulnerabilidades que involucran archivos.
2. **Wapiti**. Posee fortalezas en las vulnerabilidades clásicas de alto riesgo como *Cross Site Scripting* y *SQL Injection*, pero tiene un espectro reducido de detección, ya que no detectó otros tipos de vulnerabilidades.
3. **W3AF**. Posee fortalezas en las vulnerabilidades clásicas de alto riesgo, al igual que Wapiti, pero con un mayor rango de detección. También muestra fortalezas en vulnerabilidades con enfoque en la subida de archivos y es capaz de detectar vulnerabilidades particulares tanto de inyección como de manipulación de las respuestas HTTP. Las debilidades que presenta W3AF se relacionan con la manipulación de directorios o en la ejecución de código en el servidor como *Path Traversal*, *Directory Browsing* y *Command Injection*.

En base a esto, se puede observar que los *scanners* integrados en la plataforma se complementan entre si, y un *scanner* suple la debilidad de otro. Esta relación se puede observar con mayor claridad entre ZAP y W3AF, donde el primero posee cierta debilidad frente a vulnerabilidades de archivos y el segundo tiene fortalezas justamente en este ámbito. Además, W3AF posee debilidades en vulnerabilidades de manipulación de directorios y ZAP posee fortalezas en la detección de estos tipos de vulnerabilidades.

Esto no significa que entre esas dos herramientas podrán detectar todas las vulnerabilidades, de hecho se observó que no son capaces de hacerlo, por lo que es necesario buscar una herramienta que las complemente. De las debilidades de los *scanners* se puede destacar

que la plataforma carece capacidad de detección de vulnerabilidades con foco en la sesión, ya que no se logró detectar ningún *Cross-Site Request Forgery*. Otro foco puede ser el de escalamiento de privilegios, vulnerabilidad que tampoco fue detectada por ninguno de los *scanners* y, finalmente, carece de capacidad de detección de anomalías del programa, como por ejemplo *Buffer Overflow*.

Con respecto a los resultados obtenidos, se observó que en uno de los tres ambientes la plataforma obtuvo una precisión mayor a la de los *scanners* de manera independiente, sin considerar las vulnerabilidades no documentadas detectadas. Es en este caso en que se observa el beneficio de tener múltiples *scanners* integrados en una misma plataforma. Otro caso donde esta modalidad resulta beneficiosa es en el análisis del número de vulnerabilidades que se detectaron para cada tipo. Como se observó que las herramientas detectan distintas vulnerabilidades dentro del mismo tipo, es decir que las encuentran en distintas rutas y parámetros dentro de la aplicación, se ve aumentada la capacidad de detección de la plataforma a medida que la integren *scanners* que suplan con las debilidades de los demás.

En segundo lugar, se logró desarrollar una herramienta con una arquitectura modular en la cual se integraron tres herramientas de *pentesting* en formato de módulos. Las herramientas de *pentesting* escogidas son específicas para el análisis de vulnerabilidades en aplicaciones web y, además, permite la integración de otros tipos de herramientas de *pentesting* a la plataforma. Esto permite cumplir con el objetivo de: *"Desarrollar una herramienta con arquitectura modular que permite su extensión y modificación en el tiempo. La herramienta incluirá al menos tres tecnologías de pentesting"*.

En tercer lugar, para la obtención de los resultados se utilizó en forma permanente el módulo para definir los ataques preestablecidos a utilizar en cada ejecución. Este módulo se controla mediante el archivo de configuración, el cual le permite al usuario configurar de manera directa el ataque a realizar en el siguiente análisis. Este módulo contempla el archivo de configuración detallado en la subsección *3.1 Módulo User Config Parser*. Con esto se cumplió con el objetivo de: *"Desarrollar e implementar un módulo que le permita al usuario configurar los tipos de ataques a partir de opciones preestablecidas. Se evaluará si la herramienta le permite al usuario seleccionar tipos de ataques y ejecutarlos."*

Para cumplir con el cuarto objetivo, *"Probar la herramienta en la aplicación web de una empresa. Generar un reporte analítico de los resultados de los análisis de vulnerabilidades de dicha empresa"*, la herramienta fue probada en la empresa Fintual y se diseñó un *Dashboard* que le permite al usuario visualizar los resultados obtenidos a partir de la ejecución de la plataforma.

Los resultados obtenidos coinciden con los resultados esperados debido a que se desarrolló una plataforma modular que permite incorporar múltiples herramientas de *pentesting* mediante la implementación de los módulos respectivos para interactuar con la solución. También se logró el desarrollo del módulo que permite al analista especificar los tipos de vulnerabilidades a analizar. Además, se logró evidenciar la capacidad de detección de vulnerabilidades de la plataforma a través de los ambientes con vulnerabilidades documentadas. Finalmente, la plataforma se probó en la aplicación web de Fintual y se generó un reporte con los resultados obtenidos de las ejecuciones de los análisis de vulnerabilidades.

5.2. Trabajo futuro

Debido a que no se obtuvo una precisión del 100 % de detección de las vulnerabilidades en los ambientes vulnerables, se propone como trabajo futuro encontrar e integrar un *scanner* que supla con debilidades de los que ya están incorporados a la plataforma. Esto se hace necesario pues los *scanners* utilizados presentan debilidades en ciertos ámbitos, que si bien entre ellos se compensan en distintas áreas, dejan tipos de vulnerabilidades sin detección.

Además, se requiere obtener mayor validación con distintas aplicaciones reales, por lo que también se propone poner a prueba la plataforma con distintas aplicaciones web de organizaciones establecidas y así, con mayor cantidad de datos, se puede ajustar la arquitectura en base a las necesidades que vayan surgiendo.

Queda como trabajo propuesto la implementación de la actualización de pesos (*update weights)* y la validación de las fórmulas planteadas como mecanismo de priorización de las vulnerabilidades detectadas y clasificadas por el analista.

Finalmente, debido a que se detectó un funcionamiento equívoco del *scanner* Wapiti en el ambiente Mutillidae y no se logró determinar su origen, se propone como trabajo futuro identificar el motivo de este funcionamiento no deseado.

5.3. Open source

Como se ha mencionado en este documento, una de las problemáticas identificadas en el rubro de la detección de vulnerabilidades es la gran variedad de herramientas que son utilizadas cotidianamente y que deben ser ejecutadas de manera manual o que están parcialmente automatizadas.

Debido a que a mayor cantidad de herramientas de análisis de vulnerabilidades, aumenta la capacidad de detección de éstas, es útil que los usuarios de herramientas de pentesting incorporen e integren las herramientas que utilizan de forma cotidiana en la plataforma desarrollada. De esta manera, en vez de utilizar particularmente su herramienta deseada, utilizan la plataforma como intermediario de su herramienta, que a su vez, tendrá mayor capacidades que la herramienta individual original.

Para que esto sea viable, es necesario que la interacción entre el usuario y la herramienta sea fluida y directa debido a que estará compuesta de múltiples herramientas.

Es por esto que se propone que el desarrollo de este trabajo de memoria quede como un proyecto *open source* [1] con el objetivo de generar un aporte en la comunidad de la ciberseguridad, sirviendo de *framework* para centralizar y automatizar las diversas herramientas utilizadas por los usuarios de esta área.

[1]Link del proyecto en Github: https://github.com/gpilleux/automatedwebscan

Conclusión

A partir de la solución propuesta y la plataforma desarrollada en este trabajo de memoria, se cumplió el objetivo general de diseñar, desarrollar e implementar una herramienta que facilite las pruebas de penetración de aplicación, incorporando la capacidad de utilizar múltiples herramientas de *pentesting* a través de una configuración única.

Los objetivos específicos también se lograron con éxito, ya que la plataforma desarrollada es capaz de detectar vulnerabilidades en aplicaciones web con ataques preconfigurados, siendo estos ataques los que vienen por defecto en cada herramienta seleccionada. Este objetivo específico se comprobó a través de pruebas realizadas en los ambientes vulnerables, donde se logró visualizar la capacidad para detectar vulnerabilidades en dichos ambientes.

La solución desarrollada fue diseñada con una arquitectura modular que permite su extensión mediante la implementación de módulos adicionales específicos que se pueden integrar con la plataforma. Como parte de este trabajo, se integraron tres herramientas de análisis de vulnerabilidades de aplicaciones web. Además, el módulo de configuración de la herramienta le permite al usuario especificar ataques específicos preestablecidos para realizar un análisis de vulnerabilidades particulares.

Finalmente, la plataforma producto de este trabajo de memoria fue probada en la aplicación web de la empresa Fintual AGF, donde se realizaron las pruebas de análisis de vulnerabilidades en ambientes controlados, como en el ambiente local y en el ambiente de *staging* en Heroku. Se detectaron vulnerabilidades de riesgo alto que, luego de una inspección manual, se descartaron como falsos positivos. Aún así, es importante que se verifiquen y validen estas detecciones para determinar que ellas efectivamente no están presentes.

La solución desarrollada en este trabajo, junto con los resultados presentados, son un aporte al rubro de la ciberseguridad y, específicamente, al *pentesting*, donde se abordó la problemática no resuelta de la ejecución de múltiples herramientas de *pentesting* en forma simultánea con una única configuración. Se entregó una plataforma que automatiza y simplifica la ejecución de herramientas de *pentesting*, estandarizando y consolidando los resultados de las herramientas que la integran.

Bibliografía

[1] 7 Phases of Pentesting. https://www.redlegg.com/blog/pen-test-steps.

[2] The ZMap Team. https://zmap.io/.

[3] Gordon Lyon. NMAP Network Mapper. https://nmap.org.

[4] H. D. Moore. Vulnerability Scanning with Metasploit. https://www.offensive-security.com/metasploit-unleashed/vulnerability-scanning/.

[5] Raphael Mudge. Armitage - Cyber Attack Management for Metasploit. https://github.com/rsmudge/armitage.

[6] Simon Bennetts. Zap - The world's most widely used web app scanner. Free and open source. https://www.zaproxy.org/.

[7] Nicolas Surribas. Wapiti: The web application vulnerability scanner. https://wapiti-scanner.github.io/.

[8] Andres Riancho. W3AF - Web Application Attack and Audit Framework. https://github.com/andresriancho/w3af.

[9] Chris Sullo and David Lodge. Nikto is an Open Source (GPL) web server scanner. https://cirt.net/nikto2.

[10] David Peterson. Astra - REST API Penetration Testing. https://github.com/flipkart-incubator/Astra.

[11] Paolo Perego. Dawnscanner - The raising security scanner for Ruby web applications. https://github.com/thesp0nge/dawnscanner.

[12] OASIS Group. WS-Attacker is a modular framework for web services Penetration Testing. https://github.com/RUB-NDS/WS-Attacker.

[13] Groupon. Codeburner: One static analysis tool to rule them all. https://github.com/groupon/codeburner.

[14] Justin Collins. Brakeman is a static analysis tool which checks Ruby on Rails applications for security vulnerabilities. https://github.com/presidentbeef/brakeman.

[15] Portswigger. Burp Suite is a set of tools used for penetration testing of web applications. https://portswigger.net/burp.

[16] Tenable. Nessus - one of the most widely deployed security technologies on the planet and the gold standard for vulnerability assessment. https://www.tenable.com/products/nessus.

[17] Rapid7. Nexpose Vulnerability Scanner - your on-prem vulnerability scanner. https://www.rapid7.com/products/nexpose/.

[18] SEWORKS. Pentoma - Automated Penetration Testing for web: No manual forces involved. https://se.works/product/pentoma.

[19] Center for Internet Security. The 18 CIS Controls. https://www.cisecurity.org/controls/cis-controls-list/.

[20] Pedro Pineda. Cómo fuimos la primera startup chilena en quedar en y combinator. https://edu.fintual.cl/c%C3%B3mo-fuimos-la-primera-startup-chilena-en-quedar-en-y-combinator-41363df2e2dc/.

[21] Fintual - Números de capital administrado y clientes. https://fintual.cl/numeros.

[22] Bugcrowd's Vulnerability Rating Taxonomy. https://bugcrowd.com/vulnerability-rating-taxonomy.

[23] Heroku. How Heroku Works. https://devcenter.heroku.com/articles/how-heroku-works.

[24] RSpec - Behaviour Driven Development for Ruby. Making TDD Productive and Fun. https://rspec.info/.

[25] Avian Research. The GNU Netcat. http://netcat.sourceforge.net/.

[26] OWASP. Introduction to the OWASP Mutillidae II Web Pentest Training Environment. https://www.sans.org/reading-room/whitepapers/testing/introduction-owasp-mutillidae-ii-web-pen-test-training-environment-34380.

[27] Google. Google Gruyere Web Application Exploits and Defenses (Part 1). https://google-gruyere.appspot.com/part1.

[28] OWASP. Railsgoat Rails3 Tutorials. https://github.com/OWASP/railsgoat/wiki/Rails-3-Tutorials.

[29] Payloadbox - Command Injection Payload List. https://github.com/payloadbox/command-injection-payload-list.

Anexo A: Flujos de la Solución

A.1. Flujo de la solución desarrollada en el entorno local

El flujo de datos desde el usuario hasta los *scanners* se especifica a continuación. Primero, el usuario completa el archivo de configuración, en el cual entrega los parámetros mínimos del objetivo a analizar, como la URL o la IP, y el módulo **User Config Parser**, controlado por el **APT2 Manager**, lee este archivo y configura las variables de entorno, levanta los ambientes necesarios, y así el administrador invoca el módulo **APT2** con las variables ya definidas, el cual se las entrega a cada **Scanner Master** de las herramientas integradas a la plataforma, y son estas unidades las que delegan la generación del comando de ejecución a los módulos **Argument Parsers** de cada herramienta. Habiendo conformado el comando docker de ejecución, los **Scanner Masters** lo ejecutan y con eso levantan los contenedores en los que se alojan los *scanners* para realizar el análisis indicado.

Cada **Scanner Master** le indica al módulo **APT2** que concluyó la ejecución de su *scanner* y al recibir todas las confirmaciones de término de ejecución, **APT2** llama al módulo **Results Merger** que reúne y unifica los archivos de resultados estandarizados generados por los **Results Parsers** de cada herramienta.

El módulo **Analyzer** le muestra los resultados al usuario a través de la línea de comando para que este realice la clasificación de falsos positivos, el cual debe indicar los índices de las vulnerabilidades que, bajo su criterio y evaluación considera que son falsos positivos.

Finalmente, posterior al proceso de evaluación y clasificación de los resultados obtenidos, se invoca el módulo **Reporter** el cual inyecta los resultados históricos en un archivo que le permite tener una visualización estructurada y detallada de los resultados históricos de la plataforma.

Se puede encontrar un diagrama de los módulos implementados en este trabajo en el *Capítulo 3*.

A.2. Flujo de la herramienta en el ambiente de desarrollo de Fintual

En base al estudio del ambiente de desarrollo de Fintual, se determinó que la forma de integrar la herramienta en CircleCI es mediante la implementación de pruebas en RSpec,

utilizando el ejecutor *machine*.

En primera instancia se realizó la prueba más básica posible que es ejecutar la plataforma sin autenticación con un único módulo de vulnerabilidad, por ejemplo, *SQL Injection*. Se debe crear un *job* dentro del archivo de configuración de CircleCI (config.yml) en el cual se declara la ejecución de las pruebas de vulnerabilidades de RSpec. Luego, cada prueba se encarga de ejecutar la herramienta con un contexto específico, que es el que se desea probar. Para cada prueba de RSpec se asocia un archivo de configuración específico, el cual es el archivo que la herramienta desarrollada leerá para realizar las pruebas relevantes.

En general, las aplicaciones de Continuous Integration (CI) no permiten que las máquinas internas del proveedor interactúen con el medio externo por razones de seguridad y por consecuencia, no se pueden descargar los resultados de los análisis de vulnerabilidades de manera directa. Es por esto que se utilizaron los *artifacts* de CircleCI que permiten almacenar los resultados de las pruebas de la herramienta, los cuales se pueden descargar utilizando la API de CircleCI. Para esto, se programó un *script* que utiliza la API y descarga los artifacts del último análisis realizado. Una vez que se descargan los resultados del análisis de vulnerabilidades, se utiliza el módulo **Analyzer** para realizar la clasificación de falsos positivos para luego agregar los resultados al archivo común de alertas históricas de la aplicación.

Esta integración con el ambiente de desarrollo continuo se postergó en espera a que Fintual complete su desarrollo de la aplicación en Docker, lo que se espera que ocurra durante el segundo semestre de 2021.

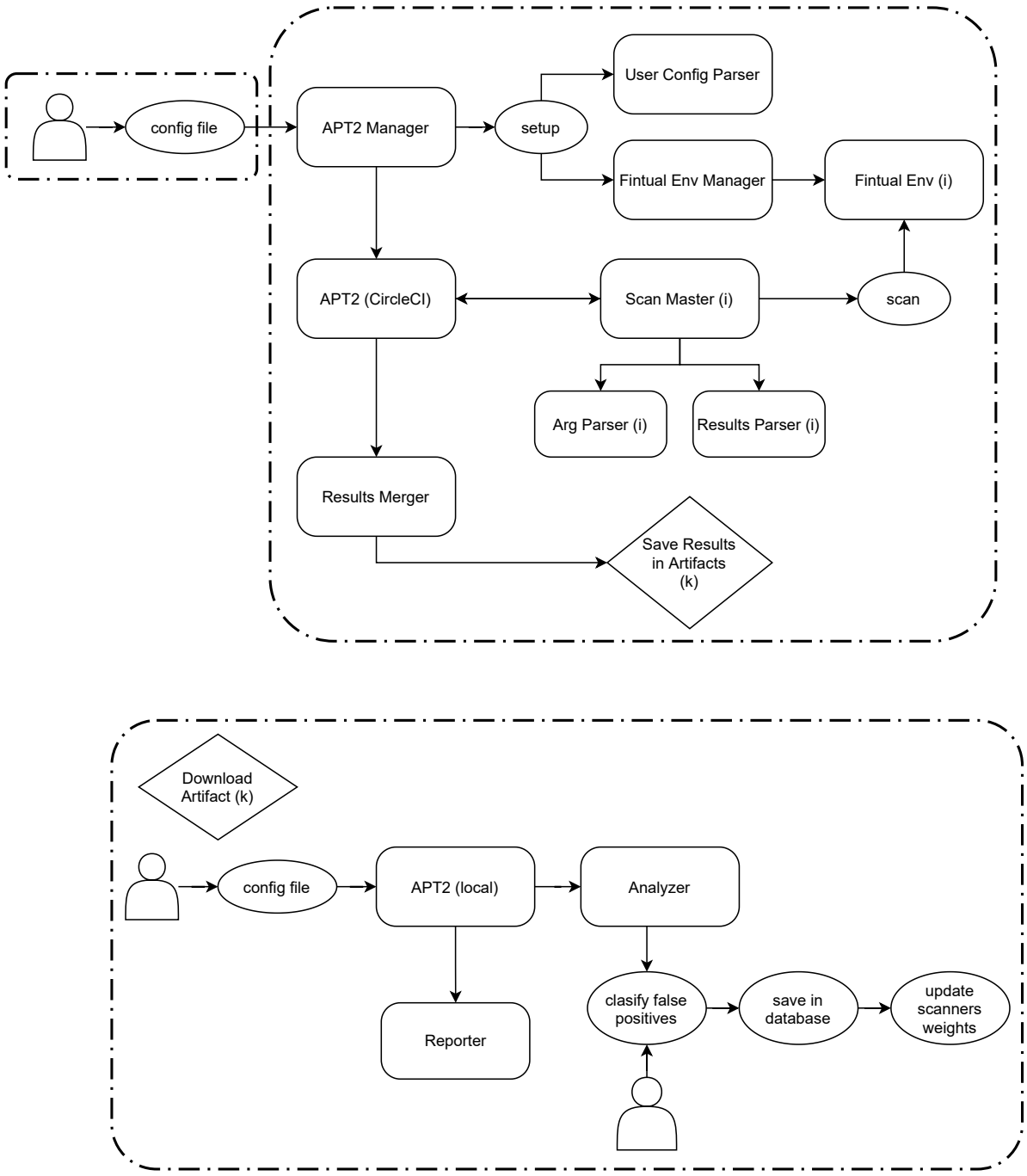

Figura A.1: Diagrama de flujo de la solución en el ambiente de desarrollo de Fintual.

Anexo B: Glosario

1. **Aplicación web**. Es un programa computacional que utiliza los navegadores web y tecnologías web para realizar tareas en la Internet. Las aplicaciones web pueden o no tener un servidor web. Las aplicaciones web estudiadas en este trabajo de memoria son aquellas que utilizan un servidor web.
2. **Aplicación web vulnerable**. Son aplicaciones web diseñadas con vulnerabilidades a propósito con un fin educacional. Las personas que poseen interés de aprender de seguridad pueden hacerlo en estos ambientes vulnerables donde podrán identificar las vulnerabilidades clásicas de las aplicaciones web en un ambiente controlado.
3. **Automated Penetration Testing Tool (APT2)**. Es el nombre de la herramienta desarrollada en este trabajo de memoria que tiene como objetivo simplificar y facilitar la realización de pruebas de penetración en aplicaciones web.
4. **Contenedor**. Es un tipo de sistema operativo virtualizado que tiene archivos ejecutables, el código binario, librerías y archivos de configuración. En comparación a un servidor o una máquina virtualizada, un contenedor no posee las imágenes del sistema operativo, lo que lo hace más liviano y más rápido para levantar el ambiente.
5. **Desarrollo continuo (CD)**. Es una metodología de desarrollo de software ágil en donde en vez de realizar grandes cambios a los sistemas, se realizan pequeños cambios en forma continua. Esta metodología genera avances constantes en el tiempo, por lo que existen pilas de pruebas que se ejecutan para revisar por errores en el código agregado y validar su funcionamiento.
6. ***Dumpster diving* (sumergirse en basureros)**. En el contexto de ciberseguridad, es la acción de buscar información de relevancia en los basureros de un objetivo, ya sea una organización o una persona natural, con el propósito de utilizar esa información para generar un vector de ataque.
7. **Fuzzing**. Es una técnica para poner a prueba un software en la cual se le entregan datos inválidos o casos de borde de tal forma de entender y solucionar los comportamientos anómalos que podrían generar estos datos. Esta técnica es relevante debido a que es imposible que una persona pueda poner a prueba un software situándose en todos los casos posibles que un atacante podría detectar para vulnerar el sistema.
8. **Infraestructura web**. Es el hardware físico y el software utilizados para interconectar los computadores y usuarios a la Internet. Entre otras componentes, la infraestructura web incluye los servidores de Internet, los servidores web y el hardware de redes.
9. **Payload (carga útil)**. En el contexto de la computación, es la información que se transmite mediante la comunicación de distintos sistemas tecnológicos. En el rubro de

la ciberseguridad, un payload, hace referencia a un código malicioso que se transmite con el propósito de explotar una vulnerabilidad.

10. **Pentester**. Es el profesional que demuestra y documenta la existencia de vulnerabilidades en los sistemas, buscando hacerlo antes que un tercero malicioso las pueda explotar, y entrega recomendaciones de mitigaciones y adaptaciones a las políticas de seguridad de los sistemas, aportándole argumentos al CISO para asegurar presupuestos adecuados para el área de ciberseguridad.
11. **Penetration Testing/Pentesting/Hackeo Ético**. Es la práctica de probar y auditar sistemas computacionales o aplicaciones web para encontrar sus vulnerabilidades de seguridad, las que podrían ser utilizadas por un atacante externo para fines maliciosos.
12. ***Pipeline* (tubería)**. Se utiliza en el contexto de desarrollo continuo como la serie de pasos que se deben realizar para el desarrollo de una nueva versión de un software. Si bien, estos pasos se pueden realizar de manera manual, el valor entregado de un *pipeline* en este ámbito es que estos procesos se realicen de manera automática.
13. ***Scanner***. Es una herramienta utilizada para detectar vulnerabilidades en un sistema. Un *scanner* generalmente está parcialmente automatizado, requiriendo que el usuario deba configurar los parámetros para que se ejecute correctamente y, luego, la ejecución misma se realiza de forma automática. En el caso de un *scanner* de vulnerabilidades de aplicaciones web, el *scanner* posee una base de datos de payloads maliciosos que envía a través de peticiones mediante el protocolo HTTP a la aplicación y captura la respuesta de éste. Dependiendo de la respuesta obtenida, se puede inferir si la vulnerabilidad está presente en el sistema, aunque su validación requiere del analista. El verbo *escanear* se utilizará en este documento como el acto de ejecutar un *scanner* sobre una aplicación.
14. ***Tailgating***. Es la acción de ir perseguir a alguien físicamente para obtener información y/o acceso de las facilidades que se encuentra entrando, con la intención de que suceda como si fuese una situación normal y no ser descubierto.
15. **Vulnerabilidad**. Es una debilidad en un sistema de información, en los procedimientos de seguridad de un sistema, en los controles internos o en una implementación que puede ser explotada por una fuente de amenaza.

Anexo C: Dashboard

La recepción del *Dashboard* por parte de Fintual fue positiva debido a que se verificó la utilidad de visualizar los resultados obtenidos de los análisis de vulnerabilidades. Se consideró valioso tener la capacidad para filtrar y desplegar el detalle de cada vulnerabilidad detectada. Además, la facultad para visualizar los resultados de los análisis históricos permiten ver el progreso en el tiempo de la aplicación web con respecto a la seguridad que presenta.

Figura 2: Vista del Dashboard con las vulnerabilidades detectadas de la última ejecución de la plataforma.

Figura 3: Vista del Dashboard con las vulnerabilidades detectadas de la última ejecución de la plataforma con filtros por riesgo y *scanner*.

Figura 4: Vista del Dashboard con con el detalle de las URL's de las vulnerabilidades detectadas de la última ejecución de la plataforma.

Figura 5: Vista del Dashboard con las vulnerabilidades detectadas históricamente de la plataforma con filtros por riesgo y *scanner*.

Figura 6: Vista del Dashboard con las vulnerabilidades de riesgo alto históricas detectadas por la plataforma.

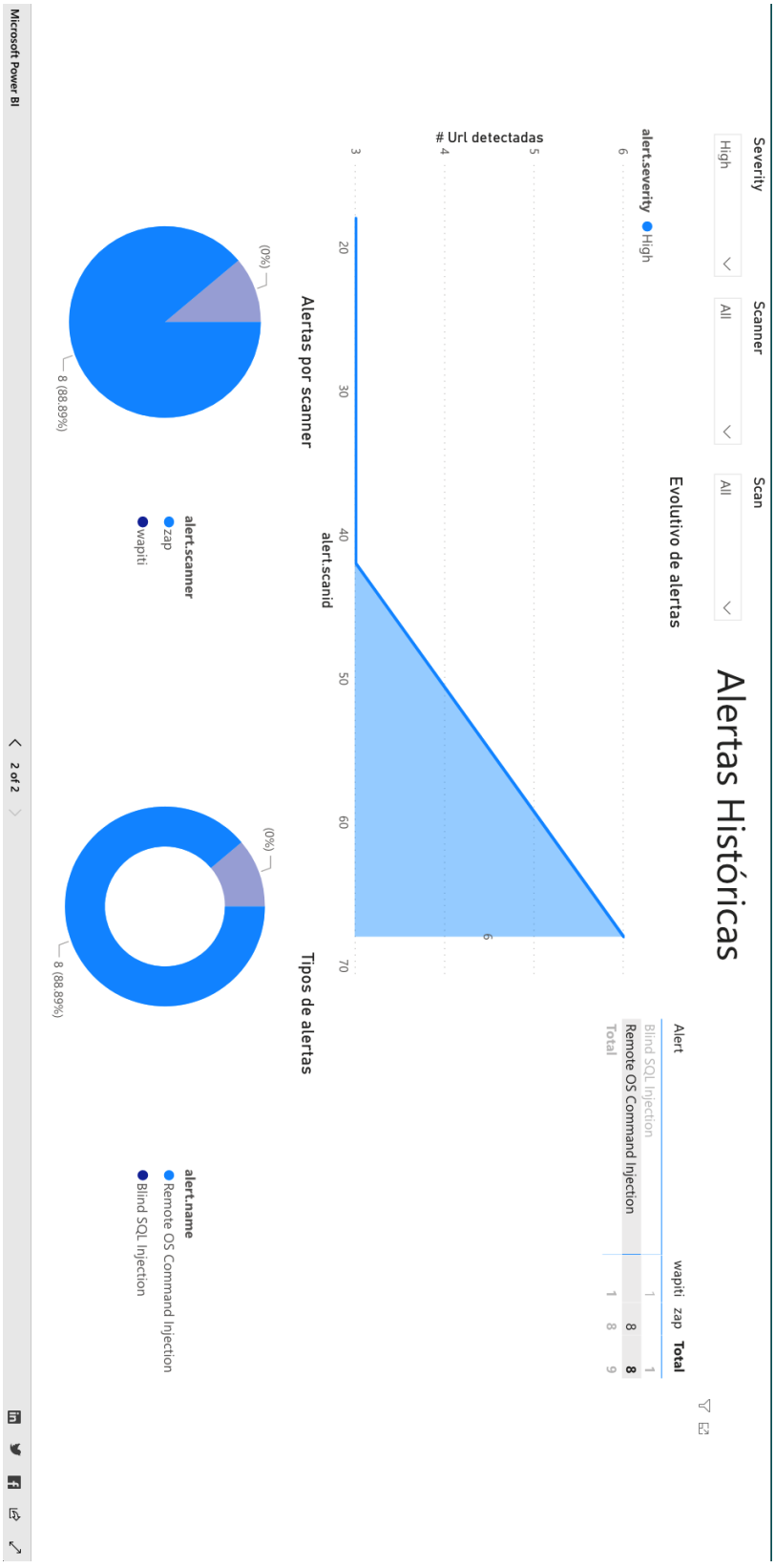

Figura 7: Vista del Dashboard con la visualización de las detecciones históricas de la vulnerabilidad *Remote OS Command Injection* detectadas por la plataforma.

Alert	w3af	wapiti	zap	**Total**
Application Error Disclosure			92	**92**
Blind SQL Injection		7		**7**
Cross Site Scripting		7		**7**
Cross Site Scripting (Persistent)			5	**5**
Cross Site Scripting (Reflected)			45	**45**
Cross site scripting vulnerability	14			**14**
Cross-Domain Misconfiguration			90	**90**
Directory Browsing			27	**27**
External Redirect			3	**3**
Format String Error			2	**2**
Insecure file upload	1			**1**
Local file inclusion vulnerability	2			**2**
Memcache injection vulnerability	6			**6**
Path Traversal			20	**20**
Remote File Inclusion			1	**1**
Remote OS Command Injection			28	**28**
Response splitting vulnerability	10			**10**
Server Side Include			5	**5**
SQL injection	2	10	7	**19**
SQL Injection - Authentication Bypass			1	**1**
SQL Injection - MySQL			1	**1**
Unhandled error in web application	6			**6**
Vulnerable JS Library			4	**4**
X-Frame-Options Header Not Set			380	**380**
Total	**32**	**23**	**561**	**615**

Figura 8: Cantidad de detecciones únicas de los distintos tipos de vulnerabilidades por cada *scanner*.